Uma Ferramenta Baseada na Reutilização de Transformações para Geração de Interface em Sistemas Orientados a Modelo

Luciane Amorim Pereira

Universidade Federal do Rio de Janeiro
Instituto de Matemática
Dissertação de Mestrado

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

Rio de Janeiro

2006

Uma Ferramenta Baseada na Reutilização de Transformações para Geração de Interface em Sistemas Orientados a Modelo

Luciane Amorim Pereira

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. Carlo Emmanoel Tolla de Oliveira - Orientador

Ph. D., University College, 1993

_____

Prof.ª Maria Luiza Machado Campos

Ph. D., University of East Anglia, 1993

_____

Prof. Paulo Figueiredo Pires

D. Sc., COPPE/UFRJ, 2002

_____

Prof. Alexandre Sztajnberg

D. Sc., COPPE/UFRJ, 2002

Rio de Janeiro
2006

Pereira, Luciane Amorim.

Uma Ferramenta Baseada na Reutilização de Transformações para Geração de Interface em Sistemas Orientados a Modelo – Rio de Janeiro, 2006.

x, 71 p.; il.

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

Orientador: Carlo Emmanoel Tolla de Oliveira

1. MDA. 2. UML. 3. Projeto Hércules. 4. Ferramenta Megara. I. Oliveira, Carlo Emmanoel Tolla de (Orient.). II. Universidade Federal do Rio de Janeiro. Instituto de Matemática / Núcleo de Computação Eletrônica. III. Título.

Agradecimentos

Agradeço a Deus pela minha vida.

Agradeço aos meus pais por terem me dado todo o apoio necessário para que eu pudesse chegar até aqui e ter condições para realizar este trabalho.

Agradeço a minha irmã e ao meu marido pelo incentivo e pela paciência que tiveram em me escutar todas às vezes que precisei.

Agradeço ao Carlo e Ana Paula pela amizade e por tudo que me ensinaram. Agradeço principalmente ao Carlo por seu apoio e pela dedicação durante a realização deste trabalho.

Agradeço as amigas Paula e Mariana pela torcida e pelo companheirismo. Agradeço também aos amigos Griner e Denílson pelas tardes em que me liberaram para que eu pudesse me dedicar a essa dissertação.

Obrigada!

Resumo

Pereira, Luciane Amorim. **Uma Ferramenta Baseada na Reutilização de Transformações para Geração de Interface em Sistemas Orientados a Modelo.** Orientador: Carlo Emmanoel Tolla de Oliveira. Rio de Janeiro: UFRJ/IM-NCE, 2006. Dissertação (Mestrado em Informática).

A complexidade crescente dos sistemas de informação obriga os desenvolvedores a utilizarem automação para atenderem as demandas do mercado. A Arquitetura Dirigida pelo Modelo (MDA – Model Driven Architecture) propõe que o processo de automação seja baseado em modelos independentes de plataforma. No entanto, as regras de transformação não seguem a prerrogativa de serem descritas de maneira abstrata. Apesar disso, a MDA cumpre o seu papel de obter uma arquitetura dirigida pelo modelo. Este trabalho propõe que o processo de desenvolvimento seja dirigido pelo modelo, através da inclusão das definições de transformação no modelo do sistema. Essas definições representam os requisitos que governam o sistema. O processo proposto se concentra na parte de interface gráfica. Desta forma, as regras de transformações representam os requisitos que governam a interação do usuário com o sistema. Como resultado, o processo permite a representação e a reutilização do conhecimento arquitetônico inerente à interface com o usuário.

Abstract

Pereira, Luciane Amorim. **Megara: A Tool Based on the Reuse of Transformations for Interface Generation in Systems Oriented for Model.** Advisor: Carlo Emmanoel Tolla de Oliveira. Rio De Janeiro: UFRJ/IM-NCE, 2006. Master in Computer Science.

The increasing complexity of the information systems compels the programmers to use automation to keep up with the demands of the market. The MDA considers that the automation process is based on platform independent models. However, the transformation rules are mostly hardcoded and can not be described in abstract way. This work considers the model driven development process through the inclusion of the transformation rules represented in the model of the system. These rules represent the requirements that govern the system. This works applies this idea to GUI development, as a proof of concept. In this context, the transformation rules represent the requirements that govern the interaction of the user with the system. As result, not only the solution for a particular problem is derived in the implementation process, but also a more general architectural definition, that can be reused in other designs.

Lista de Abreviaturas

API - Application Program Interface

CORBA - Common Object Request Broker Architecture

DOM – Document Object Model

DTD - Document Type Definitions

JMI – Java Metadata Interchange

MDA - Model Driven Architecture

MOF - Meta-Object Facility

MVC – Model View Controller

OCL - Object Constraint Language

OMG - Object Managemant Group

PIM - Platform Independent Model

PSM - Platform Specific Model

QVT – Query, View and Transformation

SiGA - Sistema de Gestão Acadêmica

UFRJ – Universidade Federal do Rio de Janeiro

UML - Unified Modeling Language

XMI - XML Metadata Interchange

XML - eXtensible Markup Language

XUL – eXtensible User-interface Language

Índice de Figuras

Índice de Tabelas

Sumário

1. Introdução

O crescimento do número de linhas de código e da complexidade dos sistemas torna cada vez mais difícil o atendimento da demanda do mercado com a qualidade requerida e com o prazo determinado. Adicionalmente, o número crescente de usuários introduz novos requisitos como a escalabilidade do sistema e a personalização do serviço.

Com o aumento da complexidade dos sistemas, há uma quantidade maior de informação que deve ser compartilhada entre os desenvolvedores. A melhor forma de compartilhamento dessas informações é através de representações abstratas que escondam detalhes irrelevantes em um determinado contexto. Entretanto, os curtos prazos impostos pelo mercado obrigam o desenvolvedor a se concentrar na etapa de codificação. Mesmo se a etapa de modelagem é realizada, geralmente os modelos perdem o valor quando a etapa de codificação começa, pois quanto mais esta etapa avança maior a distância entre o código e o modelo, principalmente se o projeto estiver atrasado. A distância pode ser diminuída com a atualização dos diagramas, no entanto o valor adicionado com a atualização é questionável já que qualquer nova mudança se inicia no código [WARMER, KLEPPE, BAST, 2003a].

A geração automática de artefatos a partir de representações abstratas da especificação do sistema não somente eleva o nível de abstração do desenvolvimento como também valoriza a etapa de modelagem. Isto ocorre porque os modelos passam a contribuir de forma mais efetiva na construção do sistema.

Para gerar um sistema automaticamente é necessário que este obedeça a uma arquitetura bem definida. Esta arquitetura pode ser imposta através do uso de arcabouços (*frameworks*), que são criados para servirem como base para programação. Eles possuem soluções prontas para vários requisitos não-funcionais do sistema, logo evitam que as equipes de desenvolvimento tenham que se preocupar com esses aspectos a cada nova implementação.

Com a imposição da arquitetura feita através da utilização de um arcabouço, é possível gerar automaticamente os artefatos para execução de um sistema para uma plataforma específica. A geração torna-se trivial porque os modelos são impregnados de informações da plataforma escolhida. Por outro lado, a modelagem torna-se mais

detalhada dificultando a sua reutilização no caso de mudanças no ambiente computacional, mesmo que os requisitos do sistema sejam preservados, comprometendo a longevidade do sistema. Além disso, a utilização dos diagramas para a programação do sistema em outro ambiente computacional também fica comprometida.

A MDA (Model Driven Architecture) define uma abordagem que separa a especificação da funcionalidade da especificação da implementação destas mesmas funcionalidades em uma plataforma tecnológica específica. Através de transformações sucessivas, os modelos originais, independentes de tecnologia, são refinados com detalhes relativos à implementação, eventualmente gerando o código fonte do programa. O principal diferencial entre uma ferramenta MDA e uma outra ferramenta de geração de código é que naquela, as regras para as transformações podem ser criadas e alteradas pelo usuário, permitindo maior domínio sobre o código fonte gerado.

Observando algumas das ferramentas de MDA existentes, é possível verificar que para possibilitar a geração automática do modelo específico, os modelos independentes precisam ser anotados. Essas anotações tornam o modelo menos independente, pois símbolos, que não pertencem ao negócio, são incluídos para que a ferramenta possa reconhecer o padrão e executar a geração de forma coerente.

Além disso, como seu próprio nome diz o objetivo da MDA é manter a arquitetura do sistema descrita através de modelos. Seria interessante que não somente a arquitetura fosse representada através de modelos, mas que as regras de transformação, que definem esta arquitetura, também o fossem.

Este trabalho propõe um processo de desenvolvimento baseado em modelos, que inclui um modelo representativo das definições das transformações. A descrição das regras em diagramas UML aumenta o nível de abstração no processo de desenvolvimento de software, tornando mais fácil para arquiteto do sistema a modificação da transformação e trazendo mais flexibilidade para os artefatos gerados.

Para o estudo de caso relacionado ao processo proposto, foi utilizado o arcabouço Hércules [NCE-UFRJ, 2006], que é uma plataforma para desenvolvimento de sistemas de informação. Este arcabouço é utilizado atualmente no desenvolvimento de um sistema de gestão acadêmica da Universidade Federal do Rio de Janeiro.

> “Esta implementação está em produção há mais de um ano em regime de “24 horas x 7 dias”, atendendo um universo de cerca de 50000 usuários em potencial. O sistema Hércules, portanto foi provado em um ambiente de produção real, demonstrando ser um sistema confiável e robusto.” (PAIS, 2004, p. 112)

As críticas referentes ao Hércules são em relação aos inúmeros detalhes que devem ser conhecidos para que seja possível criar um módulo que execute sobre a plataforma. Entre as alternativas levantadas para resolução desse problema está a geração automática dos artefatos necessários para execução do módulo a partir de especificações abstratas do sistema.

Conceitos relativos à MDA são utilizados para que mudanças do arcabouço sejam mais facilmente incorporadas ao sistema gerado. Esses conceitos são implementados através da ferramenta Megara, que a partir de um modelo independente de plataforma gera um modelo especifico para a plataforma Hércules, utilizando regras de transformação descritas em diagramas UML.

As regras de transformação representam a arquitetura da interface gráfica do sistema. Logo, através delas, o arquiteto pode manter e reutilizar o conhecimento arquitetônico inerente à interface com o usuário. Além disso, um padrão de interface para todo o sistema pode ser imposto pelo arquiteto, tornando o sistema mais fácil de utilizar.

Este trabalho está organizado em oito capítulos. O segundo capítulo descreve alguns princípios da Arquitetura Dirigida pelo Modelo (MDA). Além disso, mostra como a linguagem de modelagem UML pode ser útil no desenvolvimento com MDA.

O terceiro capítulo apresenta alguns trabalhos relacionados, incluindo algumas linguagens e ferramentas baseadas na MDA. O capítulo quatro apresenta alguns conceitos de arquitetura do Hércules e mostra como os artefatos de um sistema desenvolvido para essa plataforma devem ser construídos.

O quinto capítulo apresenta todos os passos necessários para aplicação do processo proposto na implementação de um módulo para plataforma Hércules. A plataforma é utilizada como estudo de caso para a geração automática de artefatos com a utilização do processo.

O capítulo seis explica detalhes sobre a implementação do aplicativo Megara,

que é responsável pela tradução do modelo independente de plataforma em um modelo com detalhes do arcabouço Hércules, através da leitura de um modelo de transformação.

O sétimo capítulo mostra um caso de teste da utilização do processo proposto através da criação de modelos e da utilização do aplicativo Megara para um caso de uso do sistema de registro acadêmico da UFRJ.

2. Arquitetura Dirigida pelo Modelo

A utilização de ferramentas para a geração automática de código retira do desenvolvedor uma lista de tarefas repetitivas e possibilita que se concentre nas regras de negócio do sistema. A possibilidade de diminuição do tempo e do custo do projeto torna os desenvolvedores menos resistentes a idéia de construírem diagramas, aumentando a aceitação da fase de modelagem.

O problema da automatização é que exige que os modelos contenham muitos detalhes para que seja possível a geração de artefatos. O excesso de informação dificulta a recuperação dos requisitos e torna os modelos descartáveis no caso de uma mudança na tecnologia adotada.

A MDA (Model Driven Architecture) [WARMER, KLEPPE, BAST, 2003a] propõe a separação entre a especificação das operações do sistema e os detalhes das funcionalidades de uma plataforma[1] específica. Com isso, a MDA pretende evitar que o investimento em um sistema seja totalmente perdido caso ocorram mudanças na tecnologia ou caso uma implementação em outra plataforma seja necessária.

2.1. Conceitos básicos

A separação dos conceitos é garantida através da especificação do PIM (Platform Independent Model) e do PSM (Platform Specific Model). O PIM é um modelo independente de plataforma que contém os conceitos e o comportamento do sistema abstraindo os detalhes de uma plataforma específica. Os detalhes da plataforma são adicionados a um outro modelo chamado PSM. Um PSM combina as especificações do PIM com detalhes que descrevem como o sistema usa um tipo particular de plataforma. Os conceitos e o comportamento da plataforma são definidos em um modelo da plataforma, que especifica os elementos que podem ser usados na especificação do uso da plataforma por uma aplicação.

Um modelo é uma representação de um sistema (ou parte dele) através da utilização de abstrações descritas em uma determinada linguagem. Modelo em MDA é sinônimo para modelo formal, isto significa que a representação deve ser descrita através de uma linguagem com sintaxe e semântica bem definidas para que possa ser interpretado automaticamente por um computador. A geração automática de código

[1] O termo plataforma pode ser definido como uma tecnologia específica de software ou hardware que serve de alicerce para a construção de um sistema.

somente será possível com a construção de modelos formais, porque estes são precisos e computacionalmente completos.

A transformação de um modelo independente de plataforma em um modelo específico para uma plataforma é feita através da execução de definições de transformação descritas em uma linguagem de transformação. Uma definição de transformação é um conjunto de regras de mapeamento que descrevem como um modelo pode ser mapeado para um modelo alvo. Estes modelos podem estar especificados na mesma linguagem ou em linguagens distintas. Uma regra de mapeamento é uma descrição de como uma ou mais estruturas definidas em uma linguagem origem podem ser transformadas para uma ou mais construções na linguagem alvo. Uma ferramenta de transformação executa uma definição de transformação e gera um PSM a partir de um PIM. A Figura 1 mostra como os modelos, linguagens e ferramentas estão relacionados.

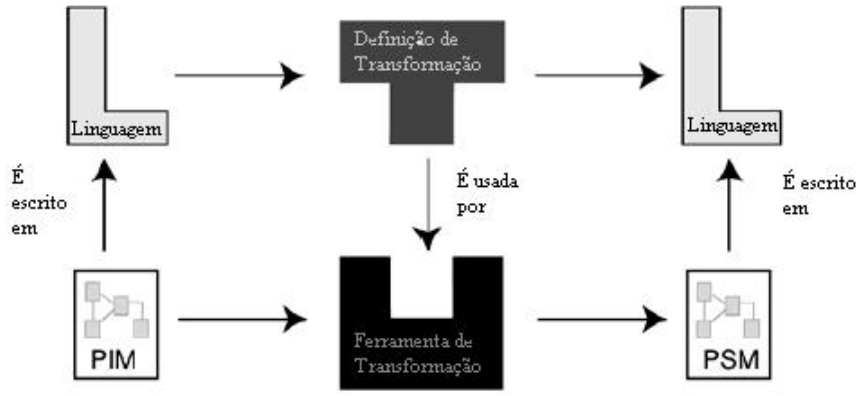


Figura 1 - Relação entre modelos, linguagens e ferramentas de transformação.
Fonte: WARMER, KLEPPE, BAST, 2003 traduzida pela autora.

Analogamente, um PSM pode ser mapeado para código fonte utilizando uma ferramenta de transformação. Neste caso, é possível também a utilização de uma ferramenta de automação que não seja guiada por transformações. O mapeamento é trivial porque o modelo específico para uma plataforma possui toda informação necessária para a geração do código.

Os sistemas são projetados com um nível de abstração onde detalhes específicos de plataformas são escondidos. O projeto abstrato permite mais flexibilidade e agilidade caso mudanças de tecnologias ocorram, já que os requisitos do sistema são descritos de maneira formal e por isso são facilmente recuperados.

A separação entre a arquitetura do sistema e a implementação protege o

investimento realizado no sistema caso ocorram mudanças na tecnologia. A inserção de detalhes tecnológicos é feita através da transformação de um modelo independente de plataforma para um modelo dependente de plataforma, essa transformação pode ser feita tanto manualmente como automaticamente.

As ferramentas de geração automática devem observar os elementos do modelo e aplicar a eles boas práticas de programação, como padrões de projeto [GAMMA *et Al*, 1995]. A aplicação dessas boas práticas garante a qualidade do artefato gerado. Outros requisitos para essas ferramentas são a possibilidade de modificação do artefato gerado e a permanência do código inserido pelo desenvolvedor. Assim, essas ferramentas evitarão que o modelo seja abandonado durante a fase de desenvolvimento do sistema devido a problemas relacionados ao prazo de entrega do projeto.

2.2. A utilização da UML na descrição dos modelos

A única observação sobre uma linguagem de descrição do modelo é que esta deve ser uma linguagem bem definida. Os modelos fonte e alvo podem ser especificados na mesma linguagem ou em linguagens diferentes. Um modelo UML (Unified Modeling Language) [FOWLER, SCOTT, 2000], por exemplo, pode ser transformado em outro modelo UML, em um modelo ER (Entidade-Relacionamento) ou em qualquer outro modelo formal.

A UML é uma linguagem adequada para a especificação de modelos, pois provê mecanismos para criação de projetos precisos suficiente para dirigir a geração de código e de projetos abstratos suficiente para permanecer independente de plataforma. Além disso, a UML tem sido largamente utilizada e aceita como linguagem padrão para modelagem de sistemas.

A UML é independente de tecnologia, entretanto a representação de detalhes dependentes de plataforma pode ser feita através da utilização de um Perfil UML [WARMER, KLEPPE, BAST, 2003]. Perfis são conjuntos de extensões para UML, que especializam um modelo para uso em um domínio particular. Os mecanismos de extensão mais utilizados são os estereótipos e os valores etiquetados (*tagged values)*.

Estereótipos definem novos tipos de elementos de modelo ampliando a semântica dos tipos existentes ou de classes em um meta-modelo UML. A notação consiste do nome do estereótipo escrito entre os sinais de maior e menor, por exemplo,

<<nomeDoEstereótipo>>. A notação é anexada a um elemento do modelo, que ganha o significado descrito pelo estereótipo. Anexar um valor etiquetado a um elemento do modelo é uma maneira de explicitamente definir uma propriedade para esse elemento. Um valor etiquetado é um par nome-valor. A notação correspondente é *{nome=valor}* com o valor sendo atribuído ao nome. Esses mecanismos são utilizados para marcar os elementos do PSM e tornam o modelo específico para plataforma.

Os elementos de uma determinada plataforma e a maneira como eles se relacionam são descritos no perfil, facilitando a modelagem e permitindo checagem do modelo. A UML 2.0 permite, através da criação de perfis, que vendedores ou usuários da extremidade configurem suas ferramentas de UML 2.0 para suportar explicitamente vários estilos de arquiteturas. Além disso, nesta versão da UML passou a existir uma maior integração com o MOF [OMG,2006].

O MOF possibilita a definição de linguagens de modelagem e permite a construção de ferramentas para a definição dessas linguagens. Adicionalmente, o MOF especifica um conjunto padrão de mecanismos de acesso e gerenciamento de modelos que foram criados utilizando qualquer linguagem descrita em um modelo MOF. A UML 2.0 define um padrão para o acesso e o gerenciamento de modelos UML baseados no meta-modelo MOF para UML.

Estes mecanismos de MOF são genéricos e podem ser realizados de maneiras diferentes. Atualmente, o OMG padronizou mapeamentos em XML, Java, e CORBA®. O mapeamento XML é chamado XML Metadata Interchange (XMI), e é usado tipicamente para transferência de metadados entre ferramentas *MOF-compliant*. Os mapeamentos Java e CORBA resultam em APIs que são úteis para trocar a informação entre ferramentas de uma maneira mais granular.

A existência de mecanismos-padrão possibilita a integração entre as ferramentas MDA, permitindo ao usuário o intercâmbio entre ferramentas. Antes do MOF, as ferramentas de cada vendedor usavam mecanismos proprietários, este era um problema tanto para os usuários, que gostariam de utilizar várias ferramentas no processo de desenvolvimento, como também para os fabricantes, porque se esforçavam para adicionar ferramentas novas aos jogos de ferramentas existentes, para integrar suas ferramentas com as de um outro vendedor, ou para integrar os tipos diferentes de meta-

dados usados em um projeto. Através dos seus mapeamentos, o MOF soluciona estes problemas, permitindo que ferramentas compartilhem informação sobre modelos UML.

2.3. Mapeamento entre modelos

Uma transformação é o mapeamento de um modelo origem para um modelo alvo seguindo as regras especificadas na definição da transformação. As definições são descritas em uma linguagem de transformação e são executadas por ferramentas apropriadas.

Na fase de geração automática dos artefatos, o modelo origem é mais abstrato do que o modelo alvo. Na fase de criação de modelos a partir dos artefatos existentes, o modelo origem é mais refinado do que o modelo alvo. Abstração pode ser definida por supressão de detalhes irrelevantes para o contexto e refinamento é um antônimo para abstração.

Detalhes podem ser adicionados ao longo do desenvolvimento, através da criação de novos modelos cada vez mais refinados [OMG, 2003]. Dessa forma o modelo alvo de uma transformação pode se tornar o modelo origem para uma outra, isto porque, o termo independência de plataforma pode ter significado diferente dependendo do ponto de vista. Exemplo, um modelo de componentes distribuídos gera um modelo de componentes CORBA, que em um segundo momento serve de entrada para uma transformação que gera um modelo de componentes CORBA específicos de uma determinada linguagem de programação. Este comportamento pode ser observado na Figura 2.

Tipos de Mapeamento

Um mapeamento MDA provê especificações para transformação do PIM para o PSM de uma plataforma particular. O modelo da plataforma determina a natureza do mapeamento. Os tipos de mapeamento da MDA estão descritos abaixo [OMG, 2003]:

- Mapeamento baseado em tipos de elementos: especifica um mapeamento de um modelo construído usando tipos especificados no PIM para um modelo expressado usando tipos de um PSM. Esses mapeamentos podem também especificar regras em termos de valores de atributos encontrados no modelo.
- Mapeamento baseado em instâncias de elementos: identifica elementos no PIM que devem ser transformados de uma maneira particular dado uma plataforma

alvo específica para PSM. Uma maneira de identificar esses elementos é através de anotações, que indicam como um elemento do PIM deve ser transformado. A colocação de anotações sobre o PIM dá origem a um novo modelo para o sistema chamado PIM anotado.

- Combinação dos dois modelos: A maioria dos mapeamentos consiste de combinação dos dois mapeamentos descritos acima.

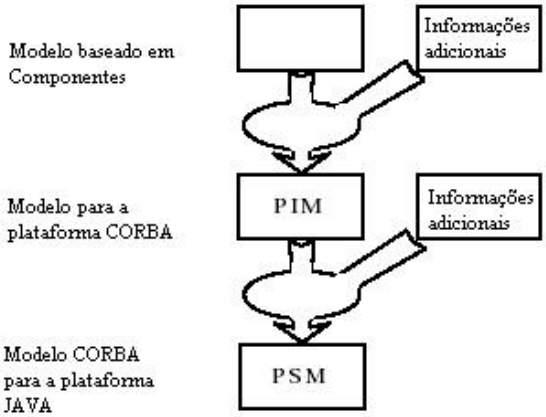


Figura 2 – A aplicação do padrão MDA
Fonte: WARMER, KLEPPE, BAST, 2003 traduzida pela autora.

3. Linguagens e Ferramentas de Transformação

O OMG (Object Managemant Group) deseja padronizar uma linguagem QVT (Query, View, Transformation) e aceitou submissões de trabalho sobre o tema. Em [OMG 2004] são descritas as características desejáveis em um processo de transformação. O mínimo necessário é permitir a consulta a modelos MOF 2.0, a criação de vistas sobre modelos e a definição de transformação entre eles.

Outros requisitos importantes são: cada aplicação deve ter a possibilidade de parametrizar o mapeamento sem modificar as definições de transformação; deve ser possível atualizar PIMs a partir de mudanças em áreas geradas do PSMs; informações adicionadas ao modelo alvo devem ser mantidas caso ocorra nova geração e as transformações devem ser bidirecionais, ou seja, elas devem ser aplicadas modelo fonte para alvo e vice-versa.

Algumas das propostas que se candidataram à linguagem QVT padrão foram: [THALES], [DSTC], [COMPUWARE], [INTERACTIVE] e [QVT-PARTNERS]. Independente da tentativa de padronização, várias linguagens foram publicadas na literatura e implementadas em ferramentas com código aberto e/ou comerciais. Cada uma destas linguagens possui características distintas. Em relação a consultas sobre o modelo, a maioria utiliza OCL pura ou alguma extensão. Outras opções adotadas foram a utilização de XQuery [W3C, 2004a] e Action Semantics [WILKIE *et Al*, 2004] para a construção de consultas. Algumas propostas devolvem como resultados somente elementos existentes no modelo e são chamadas de seletivas. Outras, chamadas construtivas, retornam elementos definidos em um meta-modelo, que podem não estar contidos no modelo consultado. Além disso, as consultas podem ser definidas como declarativas ou imperativas.

> "Declarativa é um termo geral para uma linguagem relacional ou funcional, como oposição a linguagem imperativa. Linguagens imperativas (ou procedurais) especificam uma seqüência explícita de passos que seguidos levam a produção de um resultado, enquanto linguagens declarativas descrevem relações entre variáveis em termos de funções e regras de inferência e o executor da linguagem (interpretador ou compilador) aplica um algoritmo fixo sobre essas relações com a finalidade de produção de resultado." (GARDNER, 2004, p.3)

Em relação às visões, a maioria considera uma visão como o resultado de uma transformação. Poucas a trataram como resultado de uma consulta. As propostas

divergem em relação a possibilidade de persistência e atualização da visão.

Considerando o aspecto das transformações, muitas diferenças são encontradas, entre elas estão: umas linguagens são imperativas, algumas são declarativas e outras combinam as duas abordagens; os tipos de regras variam entre os tipos unidirecional, bidirecional e misto; a maioria das propostas considera a atualização de um modelo como um caso padrão de transformação e não implementam soluções específicas para esse tipo de problema.

A ordenação das regras de transformação também é tratada de maneira diferente pelas propostas. A ordenação pode ser implícita ou explícita. Ordenação Implícita é aquela em que o usuário não tem controle explícito sob a ordem em que a execução das regras ocorre. Nesse caso, a única maneira do usuário garantir a ordem de execução é através da lógica, por exemplo, uma regra que recebe como entrada o resultado de uma outra, nesse caso esta é necessariamente executada antes daquela.

Czarnecki e Helsen (2004) classificam as transformações em Modelo-Para-Modelo e Modelo-Para-Código. Podemos considerar a transformação de Modelo-Para-Código como um caso especial de transformação Modelo-Para-Modelo; somente é necessário prover um meta-modelo para a linguagem de programação alvo. Entretanto, o código é frequentemente gerado simplesmente como texto, por isso será considerado em separado.

Na categoria de transformação Modelo-Para-Código, a maioria das ferramentas está reusando tecnologias baseadas em gabaritos (*templates*) como o Velocity [APACHE, 2004] e o XDoclet[XDOCLET, 2004] para promover a geração de código. Essa tecnologia é, em geral, utilizada em conjunto com outra que seja capaz de acessar o modelo fonte.

Um caminho menos seguido é baseado no padrão de projeto Visitor [GAMMA *et al*, 1995]. Utilizando esse padrão é possível percorrer estruturas complexas sem quebrar os princípios da orientação a objeto. Entretanto, essa abordagem é menos flexível porque mudanças nas definições da transformação acarretam em necessidade de re-compilação do código fonte.

Na categoria de transformação Modelo-Para-Modelo, as classificações foram as seguintes [CZARNECKI, HELSEN, 2004]: Manipulação Direta, Relacional, Baseada

em Grafos, Dirigida a Estrutura, Misto e Outros. Essas categorias são sucintamente explicadas nas seções abaixo.

Manipulação Direta

Nessa abordagem, a estrutura de transformação provê uma infra-estrutura mínima para organização das definições. A implementação e ordenação de regras são de responsabilidade do desenvolvedor do sistema.

Relacional

Os elementos pertencentes a esta categoria se baseiam em relações matemáticas. Segundo Czarnecki e Helsen (2003, p.11), “A idéia básica é indicar os tipos dos elementos origem e destino da relação e especificá-la usando regras”. Essa abordagem utiliza linguagem declarativa e a programação lógica é uma boa escolha para implementá-la.

Baseada em Gráficos

Definições são representadas como modelos UML e apresentam a forma declarativa. As regras gráficas de transformação consistem em um padrão gráfico original que é transformado em um padrão gráfico alvo. Esses padrões podem ser definidos de forma concreta utilizando as linguagens origem e alvo ou de forma abstrata usando MOF.

Dirigida a Estrutura

Essa abordagem é dividida em duas etapas: a primeira etapa cria a estrutura hierárquica do modelo alvo e a segunda modifica este modelo através da adição de atributos e referências. A ordem em que as regras serão executas é resolvida implicitamente.

Misto

Algumas propostas combinam diferentes técnicas das categorias anteriores. Um exemplo é a composição das abordagens declarativa e imperativa. Outros tipos de combinações também podem ocorrer.

Outros

Existem outras maneiras de realizar uma transformação, dentre elas as que mais

se destacam são: o CWM (Common Warehouse Metamodel)[WARMER, KLEPPE, BAST, 2003] e o XLST[W3C, 2004]. O CWM, padrão definido pela OMG, possui uma estrutura para transformação. Nessa estrutura existe um mecanismo para ligar os elementos origem e alvo, mas não há uma linguagem implementada para a derivação dos elementos alvo.

Modelos UML são normalmente persistidos no formato XMI (XML Metadata Interchange) [OMG, 2004a]. Sendo o XMI escrito em XML (eXtensible Markup Language)[HAROLD,1999], podemos utilizar o XLST (eXtensible Stylesheet Language Transformations) para realizar transformações entre modelos. Entretanto, o aumento da complexidade do modelo representado acarretará em dificuldade de leitura das transformações e diminuição do desempenho da aplicação.

3.1. Propostas para Linguagens e Ferramentas de Transformação

Nessa seção serão mostradas algumas propostas para linguagens e ferramentas de transformação. Os nomes das seções são nomes dos grupos responsáveis pelas propostas ou nomes das ferramentas.

Compuware Corporation e SUN Microsystems

A linguagem XMOF [COMPUWARE, SUN, 2003], candidata a linguagem padrão de transformação, e a ferramenta OptimalJ [COMPUWARE, 2004] foram propostas pelas empresas Compuware Corporation e SUN Microsystems. A proposta utiliza o MOF para a definição de meta-modelos, a linguagem declarativa OCL na execução de consultas seletivas sobre o modelo e apresenta o XMOF, uma extensão para o MOF, como linguagem para especificação de definições de transformações entre meta-modelos.

Visões são consideradas como a derivação de um modelo obtida a partir de uma transformação. Segundo os autores da proposta, o diferencial das visões é o cenário em que são usadas e não a maneira como a transformação é definida.

A linguagem de transformação permite transformações de Modelo-Para-Modelo e de Modelo-Para-Código. No caso do mapeamento para código, a abordagem de gabaritos é utilizada e no caso de mapeamento entre modelos é classificada como baseada em estruturas. A linguagem OCL é utilizada na etapa da transformação para definir quando um mapeamento é aplicável.

Os mapeamentos são definidos de forma declarativa e as regras podem ser bidirecionais. O controle da ordem de execução é feito implicitamente pela estrutura de transformação, o que por um lado evita que o usuário se preocupe com esse aspecto, mas por outro diminui o controle do usuário sobre a execução das transformações. A reutilização de definições é permitida através da composição e da especialização.

OptimalJ é uma ferramenta comercial de modelagem e desenvolvimento MDA que implementa a maioria dos conceitos propostos na extensão do MOF. A ferramenta serve como prova de conceito, comprovando a eficácia da linguagem. A ferramenta gera código somente para plataforma J2EE. As definições de transformação podem ser modificadas através de uma linguagem proprietária chamada OptimalJ's Template Pattern Language (TPL).

QVT-Partners

A linguagem QVT-Partners [QVT-PARTNERS, 2003], candidata a linguagem padrão de transformação, se enquadra na categoria da abordagem relacional. Existe uma distinção entre a especificação do comportamento do artefato e a implementação. A especificação da relação entre modelos é descrita através de relações e detalhes que caracterizam a maneira como os modelos devem ser traduzidos são descritos através de mapeamentos. Relações e mapeamentos podem ser reutilizados devido aos mecanismos de composição e especialização considerados na proposta.

Essa proposta considera relações e mapeamentos como especializações de transformação. Relações são declarativas, não executáveis, multi-direcionais e são usadas na especificação do sistema ou para checar a validação do mapeamento. Mapeamentos são implementações da transformação, executáveis e podem ser escritos em alguma Linguagem Semântica de Ação (*Action Semantic Language*). Além disso, podem criar ou alterar modelos e refinar qualquer número de relações. Linguagens de transformação baseadas em estruturas podem assumir uma escala melhor do que propostas declarativas [GARDNER, 2003].

Transformações são organizadas dentro de uma seqüência ordenada de passos. Cada passo é especificado por um conjunto de tarefas de transformação, com cada tarefa sendo definida como um conjunto de pares relação-mapeamento. Tarefas de transformação podem ser marcadas como transacionais. O rastreamento (*traceability*) de

uma transformação é conseguido via *logging* de passos da transformação. Consultas e visões são tratadas de maneira análoga a proposta anterior.

AndroMDA

A ferramenta AndroMDA é gratuita e de código aberto. Nesta ferramenta, as regras de transformação são escritas através de cartuchos [OLIVEIRA, CUNHA, 2005]. Cada cartucho permite a transformação para uma determinada plataforma. Os cartuchos são normalmente escritos em Velocity. O AndroMDA possui um conjunto de cartuchos já embutidos, mas novos cartuchos podem ser criados para suportar as plataformas não atendidas.

Os cartuchos recebem como entrada para transformação um modelo UML serializado em um arquivo XMI. Este modelo deve seguir um conjunto de convenções de modelagem para que a ferramenta possa compreendê-lo. Ou seja, o modelo independente precisa ser anotado com estereótipos para que o código possa ser gerado. A ferramenta não gera o PSM, o código é gerado diretamente.

GMT Consortium

O objetivo do consórcio é construir uma ferramenta com código aberto para auxiliar na construção de aplicações cujo desenvolvimento é dirigido pelo modelo. Com esse objetivo o grupo investiu em uma linguagem protótipo, a UMLX [WILLINK, 2003], para definição de transformações de Modelo-Para-Modelo.

A linguagem é um exemplo da categoria de linguagens gráficas. Ela estende a UML através do uso de um diagrama de transformação que definem como um modelo deve ser transformado em outro. Na transformação é mostrado um padrão de entrada, que deve ser encontrado e substituído por um padrão de saída. As regras de transformação se baseiam nos seguintes comandos: *Preservation*, *Evolution* e *Removal.*

UMLX deriva muitos conceitos importantes do XSLT (eXtensible Stylesheet Language Transformations) [W3C, 2004] e a natureza declarativa daquela permite que ela seja vista como uma linguagem de alto nível para esta.

4. O Arcabouço Hércules

O Hércules é uma plataforma para desenvolvimento de aplicações para *web*. A arquitetura do Hércules propõe a divisão do sistema em módulos que representam os casos de usos[2] do sistema [PAIS, 2004]. A tradução de um caso de uso em código fonte é feita através da utilização de um padrão de implementação imposto pelo arcabouço. Dessa forma, todos os desenvolvedores seguirão os mesmos passos para a construção dos módulos e qualquer membro da equipe poderá dar manutenção em qualquer parte do sistema.

Os casos de uso descrevem a interação do usuário com o sistema. Eles geralmente narram o que o sistema deve fazer e não como. Segundo Pais (2004, p. 54), "A lógica contida no cenário de caso de uso determina como o ator solicita a realização de funcionalidades ao sistema, e como o sistema deve apresentar o resultado correspondente". Observando a definição de camada de Vista do padrão MVC no parágrafo seguinte, podemos concluir que a descrição de um caso de uso se concentra nessa camada do sistema.

O MVC (Model View Control) é um padrão que confere qualidade, porque prega mínimo acoplamento entre as três partes de um projeto: Vista, Controle e Modelo. A camada de Vista é responsável por apresentar a informação mantida na camada de domínio e por possibilitar a interação entre o sistema e o usuário. A camada de Modelo é constituída pelos dados e pelas regras de negócio. A camada de Controle não somente estabelece uma ligação entre as outras camadas, como também conduz o fluxo de execução das operações do sistema. O MVC reduz a dependência entre partes disjuntas de um programa, facilita a manutenibilidade, aumenta a extensibilidade e encoraja o reuso.

A posição central do caso de uso em relação à arquitetura do arcabouço e a ligação do caso de uso com a camada de vista leva a uma nova divisão do sistema. O Hércules propõe a aplicação recursiva da tríade MVC dando ênfase à camada de vista. A Vista é dividida em três partes: modelo, vista e controle. A camada de modelo é constituída das informações das entidades que serão mostradas durante o caso de uso (espelho da entidade). A camada de vista é responsável pela apresentação e pela captura

[2] Casos de usos são coleções de cenários que um ator (usuário ou sistema) realiza com o sistema para alcançar um determinado objetivo.

de solicitações do usuário. A camada de controle é responsável pelo recebimento das solicitações vindas da vista e pelo acionamento da execução das regras de negócio correspondentes implementadas no Controle da aplicação.

A arquitetura do arcabouço utiliza os conceitos da arquitetura MVC e a aplicação recursiva da tríade para estabelecer a divisão da lógica em domínio, apresentação e controle. A camada de Domínio é formada pelo Modelo, pelo Controle e pelo modelo da vista, contendo as entidades de domínio, as regras de negócio e sendo responsável por prover o espelho da entidade[3]. A camada de Controle é constituída pelo controlador de caso de uso e a camada de Apresentação é constituída pela vista da Vista. A aplicação do padrão MVC na arquitetura Hércules é mostrada na Figura 3.

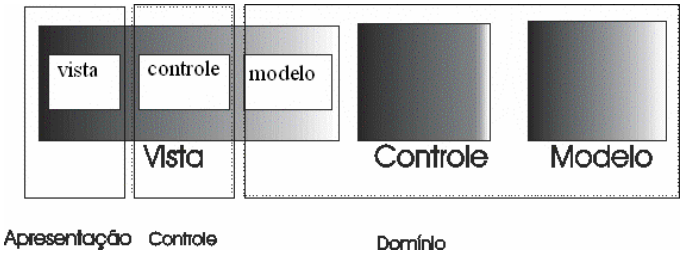


Figura 3 - A aplicação do padrão MVC na arquitetura do Hércules
Fonte: PAIS, 2004

O arcabouço estabelece um conjunto de descrições de alto nível que são utilizadas pelos desenvolvedores para especificar a apresentação, fluxo de execução do caso de uso e os espelhos das entidades. Essas descrições são escritas em um arquivo XML, que é formado por três seções: *view*, *control* e *model*. Existe um arquivo XML para cada caso de uso do sistema. A divisão, mostrada na figura abaixo, foi criada para facilitar a implementação do padrão MVC.

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

Figura 4 - A especificação abstrata do caso de uso

[3] Um espelho da entidade é uma representação de uma entidade de domínio. No espelho, algumas informações da entidade são omitidas. Essa omissão visa uma melhor adequação da entidade a um determinado caso de uso do sistema.

As tags do XML podem possuir atributos, que auxiliam na descrição do elemento representado pela *tag*. Cada elemento possui um atributo, que lhe serve como identificador único.

A seção *view* contém os elementos que compõem a interface do usuário. O nome da *tag* especifica o tipo de componente gráfico que deve ser renderizado. Todos os componentes visuais que exibem uma informação que está armazenada em um elemento do modelo possuem um atributo chamado dataSources e um chamado ref. O atributo dataSources indica o espelho de entidade e o ref indica a propriedade do espelho que deve ser exibida. Além disso, essa seção mostra a maneira como os elementos devem estar organizados na tela. A seguir é mostrada a seção *view* de uma especificação, onde alguns atributos foram omitidos.

```
<view>
  < tabbox>
    <tabs>
      <tab label=”Edição” selected=”true”/>
    </tabs>
    <tabpanels>
      <tabpanel>
        <vbox>
          <hbox>
            <label value=”Código da Turma:”/>
            <textbox datasources=”Turma” ref=”codigo”
            type=”text” multiline = “false”/>
          </hbox>
          <hbox>
            <label value=”Nome da Turma:”/>
            <textfield datasources=”Turma” ref=”nome”
            type=”text” multiline = “false”/>
          </hbox>
        </vbox>
      </tabpanel>
    </tabpanels>
  </tabbox>
</view>
```

A partir da especificação abstrata da interface gráfica, o arcabouço Hércules poderá gerar múltiplas apresentações de um sistema, além de garantir a consistência entre elas. O desenvolvedor se restringe apenas a construir, uma única vez, a

especificação da interface gráfica do sistema. A descrição da apresentação é entendida diretamente pelo engenho de execução do arcabouço. Após a leitura da seção *view*, o engenho cria a camada de Apresentação do sistema. Esta camada é responsável por representar o conteúdo e interagir com o mundo externo. Além disto, esta camada dispara as solicitações, para que elas sejam tratadas pela camada de Controle.

A criação da camada de Controle é baseada na seção *control* da descrição abstrata. Esta camada é responsável por interpretar as requisições provenientes da camada de apresentação, comandar a execução das regras de negócio na camada de Domínio, obter o conteúdo que deve ser disponibilizado pela camada de Apresentação e comandar a apresentação do conteúdo na interface gráfica. O controlador do caso de uso é responsável por controlar o fluxo de controle de um caso de uso, ou seja, é responsável por coordenar o fluxo de execução das regras de negócio. O controle é feito através da utilização de uma máquina de estados. Uma máquina de estados é uma representação de alto nível, que precisa ser contextualizada para se obter o entendimento adequado. A interpretação de cada elemento da máquina no contexto da arquitetura do Hércules será descrita a seguir:

- Estado – representa como o sistema se apresenta ao ator do caso de uso. Cada estado corresponde a uma tela apresentada ao usuário.
- Eventos – correspondem a ações solicitadas pelo usuário. São originados da interação do usuário com os elementos visuais, no contexto de uma interface gráfica. A ocorrência de um evento pode acarretar a transição entre estados.
- Transições de estado – causam uma alteração na tela apresentada ao usuário. Uma transição é causada pela ocorrência de um evento, logo uma transição corresponde ao processamento de uma operação requisitada pelo usuário. Esse processamento pode acionar uma regra de negócio.
- Ação – corresponde a uma regra de negócio. As ações ocorrem durante a execução das transições de estado, correspondendo à execução de regras de negócio solicitadas pelo usuário através de um evento.
- Condição de guarda – condiciona a ocorrência de uma determinada transição.

A máquina de estados é acionada através de uma interação do usuário com o sistema, levando a execução de regras de negócio e a apresentação do resultado na

interface com o usuário. O acionamento da máquina de estados implica em traduzir a interação do usuário com a interface em eventos. Assim como, transições de estado devem ser traduzidas na execução de regras de negócio e na produção da resposta para o usuário.

O controlador de caso de uso é o elemento responsável por interagir tanto com a interface com o usuário como com os objetos de negócio, conforme pode ser verificado na Figura 5.

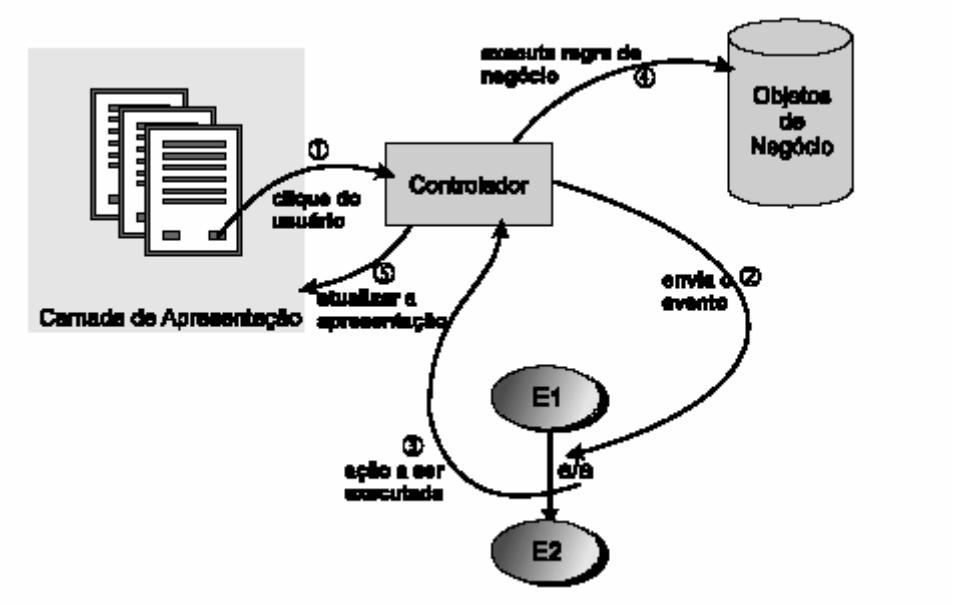


Figura 5 - O controlador de caso de Uso
Fonte: PAIS, 2004

A seguir é mostrada a seção *control* de uma especificação, onde alguns atributos foram omitidos.

```
<control>
      <initial-state>PesquisandoTurmas</initial-state>
      <state name=" PesquisandoTurmas"/>
      <state name="SelecionandoTurma"/>
      <event name="consultar"/>
      <transition name=" transicaoConsultar">
            <source> PesquisandoTurmas </source>
            <target> SelecionandoTurma </target>
            <trigger>consultar</trigger>
            <effect>mudaTela</effect>
      </transition>
</control>
```

A máquina de estados referente à especificação acima é mostrada na figura abaixo através de um diagrama de estados da UML.

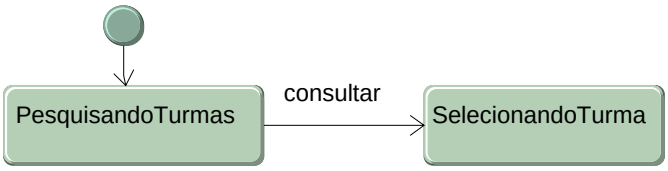


Figura 6 - Diagrama de estados referente a descrição

A seção *model* contém a descrição dos espelhos das entidades envolvidas no caso de uso que está sendo especificado. Os valores dos elementos que devem ser mostrados na tela são armazenados nesta seção. O elemento *JavaBean* descreve o espelho de uma entidade de domínio, enquanto que o elemento property descreve um atributo da entidade. A seguir é mostrada a seção *model* de uma especificação, onde alguns atributos foram omitidos.

```
<model>
   <JavaBean class="Turma">
       <properties>
            <property name="codigo" type="java.lang.String"
            value="" />
            <property name="nome" type="java.lang.String"
            value="" />
       </properties>
   </JavaBean>
</model>
```

As entidades de domínio consistem em uma representação da informação que é apropriada para a execução de regras de negócio, e que se aproxima do problema no mundo real. No entanto, esta representação pode não ser suficientemente apropriada para ser fornecida ao mundo externo. Por este motivo, a camada de domínio provê uma apresentação das entidades compreensível para o mundo externo, que constitui o espelho dessas entidades (modelo da Vista). Existe um conjunto de espelhos específico para cada caso de uso. Este espelho permite que o conteúdo das entidades seja mais adequadamente exibido pela interface gráfica, por um relatório, ou por qualquer outra forma possível de apresentação.

Além de prover os espelhos das entidades, a camada de Domínio é responsável por persistir os dados e manter as regras de negócio da aplicação. Além disso, qualquer

solicitação de modificação da informação é respondida por esta camada. Essas outras atribuições da camada de Domínio não são tratadas pela plataforma Hércules.

O projeto Hércules possui três subprojetos encarregados pela criação de ferramentas que agregam funcionalidades ao projeto principal, são eles: O projeto Lion é responsável pela ferramenta que traduz a descrição em XML do controle e transforma em uma classe JAVA; o projeto Chiron pesquisa formas de persistência e de representações das entidades de domínio. O projeto Megara, do qual este trabalho faz parte, objetiva a geração automática, a partir de diagramas da UML, das especificações abstratas necessárias para execução do Hércules.

5. Proposição do processo

A utilização da ferramenta Megara implicará na adoção de um processo de desenvolvimento dirigido pelo modelo. O arquiteto do sistema torna-se responsável por produzir as regras de transformação que guiarão a construção do modelo específico. Na linguagem proposta, diferentemente das outras linguagens de transformação, as regras são escritas na linguagem UML original, sem extensões. A utilização da UML facilita o aprendizado da linguagem e torna a representação das regras mais abstrata.

A linguagem utilizada para definição da transformação é baseada em gráficos, imperativa, unidirecional, considera a atualização de um modelo como um caso padrão de transformação e não implementa soluções específicas para esse tipo de problema.

Os desenvolvedores modelam as regras de negócio do caso de uso sem que detalhes de uma plataforma específica sejam considerados (PIM). A ferramenta Megara gera o modelo específico (PSM) baseado no modelo da plataforma e nas regras de transformação. O desenvolvedor poderá alterar os modelos gerados antes de solicitar que a ferramenta gere os artefatos necessários para execução de um caso de uso na plataforma Hércules. A Figura 7 mostra as etapas do processo. A seguir, estas etapas são descritas com mais detalhes.

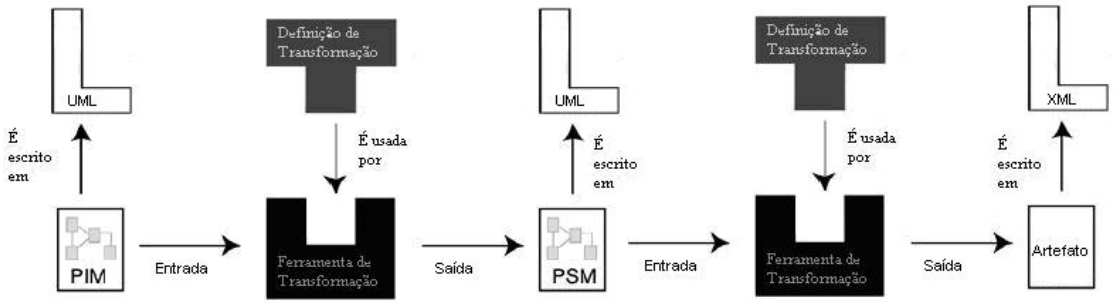


Figura 7- As etapas do processo de desenvolvimento

O Modelo Independente de Plataforma

O modelo independente de plataforma deve descrever a estrutura e o comportamento do caso de uso. Este modelo é representado através de um diagrama de classes, onde existem elementos marcados com diferentes estereótipos. Essas marcações se referem à aplicação da arquitetura MVC.

As entidades do sistema são marcadas com o estereótipo <<*model*>>. No

diagrama são representadas apenas as entidades e os atributos utilizados no caso de uso que está sendo modelado. A abstração de informações não relevantes para um caso de uso específico facilita o entendimento da equipe e dos clientes. Na plataforma Hércules, essas entidades modificadas são chamadas de espelho da entidade ou modelo da Vista.

O controle faz a ligação entre a vista da Vista e o modelo da Vista. A classe que representa o controle é anotada com o estereótipo <<*binder*>>. A classe com o estereótipo <<*view*>> representada uma ou várias telas do caso de uso. Para um melhor entendimento do processo, este será aplicado com o auxílio de um exemplo. A Figura 8 mostra o exemplo de diagrama independente de plataforma do sistema Chronos.

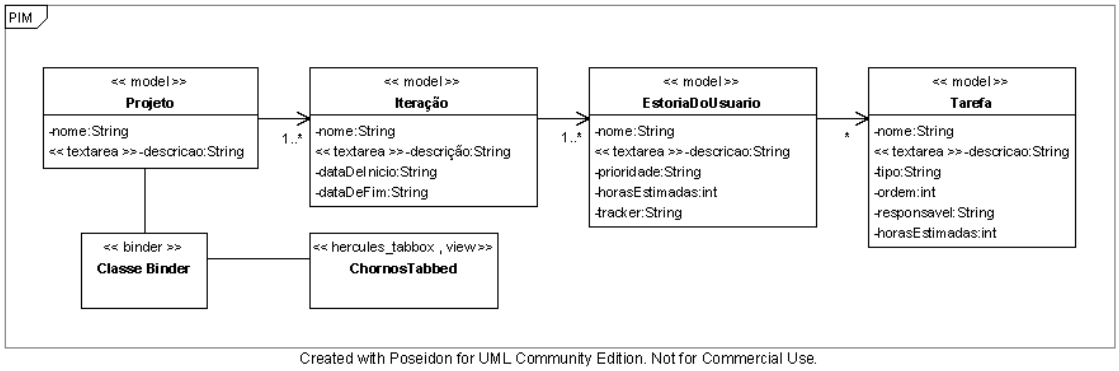


Figura 8 – Modelo independente de plataforma para o sistema Chronos

O diagrama da figura representa o modelo do sistema Chronos, sem que detalhes da plataforma sejam levadas em conta. O sistema Chronos serve como um organizador para projetos que utilizam o XP (Extremming Programming) como processo de desenvolvimento. Quando surge uma demanda para um novo projeto de software, este projeto é criado e dividido em iterações. Novas iterações podem surgir ao longo do projeto. Uma iteração é um período de tempo em que algumas estórias do usuário são implementadas. Para que uma estória seja implementada, ela é dividida em várias tarefas. Um desenvolvedor se candidata ou é indicado como responsável por uma determinada tarefa.

O Modelo de Plataforma

Um modelo de plataforma provê um conjunto de conceitos técnicos, representando as partes e os serviços de uma plataforma. O modelo também provê conceitos que representam os diferentes tipos de elementos que podem ser usados na especificação do uso da plataforma por uma aplicação.

Durante a fase de desenvolvimento, o arquiteto escolhe uma plataforma onde o sistema será implementado. O arquiteto utiliza um modelo da plataforma para programar o sistema. Esse modelo geralmente está na forma de manuais ou apenas na cabeça do arquiteto. Na MDA, os modelos devem ser detalhados, devendo estar descritos em uma linguagem com sintaxe e semântica bem definidas.

Diagramas de classe e de estado da UML foram utilizados para descrever o modelo de um sistema que utilize o processo de desenvolvimento dirigido pelo modelo. O modelo do sistema deve conter os pacotes que representam cada caso de uso da aplicação, o pacote base e o pacote architecture. Para cada pacote de caso de uso, devem existir três sub-pacotes: o pacote PIM contém o diagrama que representa o modelo independente de plataforma e que foi descrito na seção anterior, o pacote TM contém o diagrama que representa o modelo de transformação e o pacote PSM contém três sub-pacotes. Os sub-pacotes do pacote PSM são: o modelo que contém o espelho das entidades; o controle que representa o controlador do caso de uso e a vista que contém a apresentação do conjunto de telas do caso de uso. Esses sub-pacotes representam cada uma das seções da especificação abstrata do Hércules, que foi descrita no capítulo quatro. A Figura 9 mostra a estrutura criada na ferramenta Poseidon.

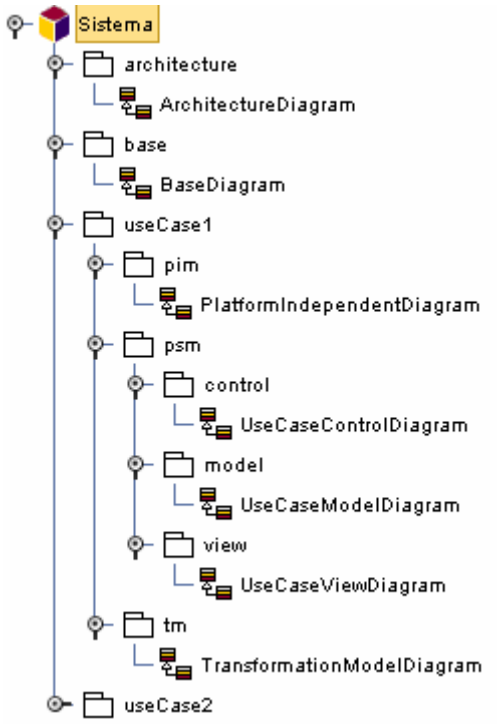


Figura 9 – Estrutura para modelagem de sistemas na plataforma Hércules

Os pacotes *base* e *architecture* servem para facilitar a reutilização. Esses pacotes e o pacote do modelo de transformação serão explicados detalhadamente nas próximas seções. Os sub-pacotes do pacote PSM serão descritos à seguir.

O diagrama de controle apresenta o controle do caso de uso através de uma máquina de estados. Cada estado representa uma tela do caso de uso e os eventos são as ações que o usuário pode realizar em uma determinada tela. Esse diagrama não é gerado automaticamente a partir do PIM, devendo ser criado pelo usuário.

O modelo do caso de uso é exibido em um diagrama de classes, onde as classes representam os espelhos das entidades. Este diagrama é formado pelas classes do PIM que possuem o estereótipo <<*model*>>.

A apresentação da tela é esquematizada em um diagrama de classes. Este diagrama é obtido através da aplicação das regras de transformação sobre o diagrama independente de plataforma. As classes que participam deste diagrama são anotadas com estereótipos definidos em um perfil UML.

O perfil UML da plataforma Hércules é definido por um conjunto de estereótipos que representam os componentes visuais disponíveis nesta plataforma. A estes componentes, são acrescidos classes e estereótipos inerentes ao Megara que permitem a construção de um modelo de transformação. A Figura 10 mostra a caixa de diálogo da ferramenta Poseidon que permite a aplicação dos estereótipos disponíveis no perfil.

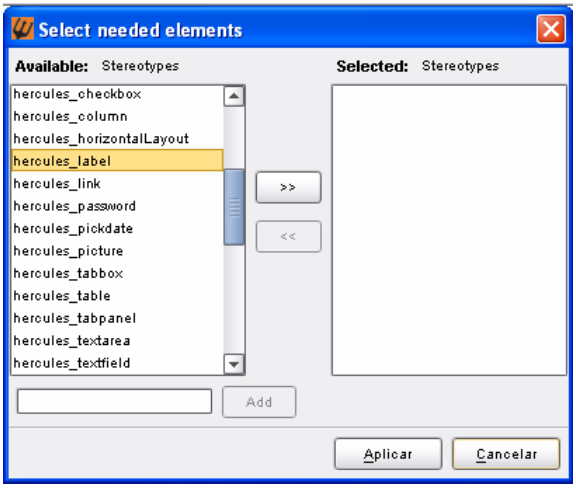


Figura 10 – Subconjunto dos estereótipos disponíveis no perfil UML

Os componentes visuais podem ser classificados em primitivos ou derivados. Os elementos derivados são formados pela composição de elementos primitivos e/ou derivados. Esses elementos são utilizados como um estoque de componentes pré-fabricados. A utilização desse tipo de elemento permite a reutilização de elementos visuais, diminuindo a repetição de tarefas durante a fase de desenvolvimento do projeto. Os elementos derivados podem ser criados no pacote *base* ou no pacote *architecture* para tornar mais fácil a identificação e a reutilização dos elementos. Estes elementos podem ser posteriormente agregados ao perfil UML. O meta-modelo parcial deste diagrama é apresentado na Figura 11.

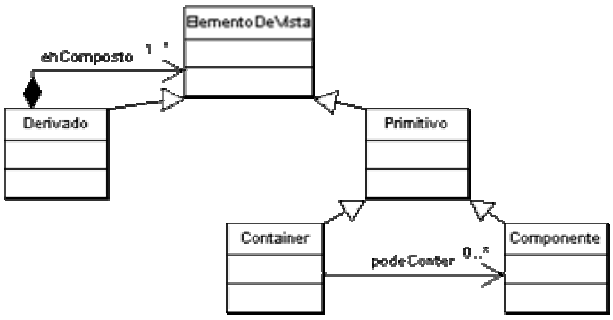


Figura 11 - Meta-modelo parcial dos elementos de vista
Nota: As sub-classes das classes Contêiner e Componente não estão representadas na figura.

O Modelo de Transformação

Este trabalho propõe o uso da UML para a definição do modelo de transformação. Como prova de conceito, a transformação será feita para a plataforma de apresentação Hércules. O modelo de transformação é representado por um diagrama de classes que descreve como deverão ficar as telas do sistema. Esse diagrama contém simultaneamente o padrão de reconhecimento e o conjunto de regras de transformação, que são responsáveis pela tradução do PIM para PSM. O padrão de reconhecimento é descrito através de valores etiquetados e as regras de transformação são descritas através de classes que possuem estereótipos que representam a plataforma final de saída.

Considerando o Hércules a plataforma de saída, o diagrama do modelo de transformação é formado, essencialmente, com classes que possuem estereótipos que representam elementos visuais primitivos ou derivados do Hércules. Essas classes representam as regras de transformação, indicando o que deve ser gerado no PSM. Existem associações que fazem a ligação entre essas classes e que possuem valores etiquetados. Estes valores etiquetados indicam o padrão que deve ser procurado no PIM.

Após o batimento do padrão de reconhecimento contra o PIM, as classes ou os atributos que obedecem ao padrão devem ser selecionados para que o comando de transformação possa ser executado sobre eles. Os comandos de transformação são sempre executados sobre os elementos correntemente selecionados. Neste texto, esses elementos selecionados são chamados de entidades ativas. Alguns dos valores etiquetados utilizados para descrever padrões de reconhecimento estão listados abaixo:

- *Axis* (eixo), essa marca modifica a entidade que está correntemente ativa na transformação. Os tipos de eixos são: *association*, seleciona as entidades que possuem associação com a entidade ativa; *attribute*, seleciona os atributos da entidade ativa; e *self_or association*, seleciona a entidade ativa e as entidades que possuem associação com a entidade ativa. Os elementos resultantes da seleção são as novas entidades ativas. O conceito de eixos foi inspirado no mesmo conceito existente na linguagem XSLT.

- *level* (nível), essa marca indica até que nível de associação, os elementos devem ser selecionados. Ela deve ser utilizada em combinação com a marca *{axis = association}* ou *{axis = self_or_association}*. Por exemplo, se a ferramenta está percorrendo o PIM descrito na Figura 8 e a entidade ativa é o Projeto: caso o valor atribuído a *level* seja 1 e o valor atribuído a *axis* seja *self_or_association*, as entidades Projeto e Iteração serão selecionadas e os próximos comandos serão executados sobre elas; caso o valor atribuído a *level* seja dois, as entidades selecionadas serão Projeto, Iteração e EstoriaDoUsuario. Para que os comandos sejam executados até encontrar um elemento que não tenha associação com nenhum outro, o usuário pode colocar um valor alto para o número de níveis de associações do sistema dele, por exemplo, 1000. Não haverá processamento adicional pelo número ser grande, o processamento apenas dependerá do número de níveis. No entanto**,** o desenvolvedor deve se preocupar com a existência de ciclos.

- *stereotype* e *type*, essas marcas condicionam a execução de um determinado comando de transformação. Essas marcas devem ser utilizadas em combinação com a marca *{axis = attribute}*. A marca *stereotype*, deve ser utilizada quando a condição para que uma regra seja executada está ligada com o estereótipo que os atributos ativos possuem; e *type*, deve ser utilizado quando a condição para que uma regra seja executada está ligada com o tipo dos atributos ativos. Por exemplo, se no

padrão de reconhecimento existir um valor etiquetado *{type=String}*, somente os atributos que possuem o tipo String serão selecionados no PIM, logo os próximos comandos de transformação serão executados somente sobre eles.

- *order*, essa marca descreve em que ordem os elementos visuais devem ser desenhados. No caso da transformação de PIM para PSM, esse valor etiquetado é apenas copiado para o PSM. No caso da transformação de PSM para artefato, esta marca altera a ordem de execução das regras de transformação. Nesse caso, se a marca está presente na modelagem, a ordem de execução das regras de transformação é explícita, caso contrário é implícita.

É importante ressaltar que as regras possuem uma natureza semântica e não sintática, ou seja, as regras são aplicadas em conformidade com o tipo do elemento e não com relação a uma determinada instância. Exemplificando a discussão, o padrão não procura um atributo *name*, ao invés disso, procura todos os atributos que tenham um determinado tipo ou estereótipo. Essa utilização potencializa a abstração e o reuso do modelo de transformação.

A Figura 12 representa o modelo de transformação para criação do sistema Chronos na plataforma Hércules. Na figura, os valores etiquetados são representados no diagrama através de comentários. Podemos observar no diagrama, que estão representados tanto os elementos que servem para definir o padrão a ser encontrado quanto os comandos a serem executados sobre o padrão. Os padrões de reconhecimento estão representados na cor escura e os comandos de transformação na cor clara. A Tabela 1 mostra a seqüência inicial de comandos realizados pela ferramenta devido à leitura do diagrama.

As regras de transformação representam a arquitetura da interface gráfica do sistema. Através dessas regras, o arquiteto pode manter o conhecimento sobre a interface do sistema e pode garantir a homogeneidade da apresentação, impondo um padrão de interface para todo o sistema.

O resultado da aplicação das regras de transformações descritas no modelo representado na Figura 12 sobre o modelo independente da Figura 8 é o PSM do sistema. Os passos da transformação serão descritos nos parágrafos a seguir.

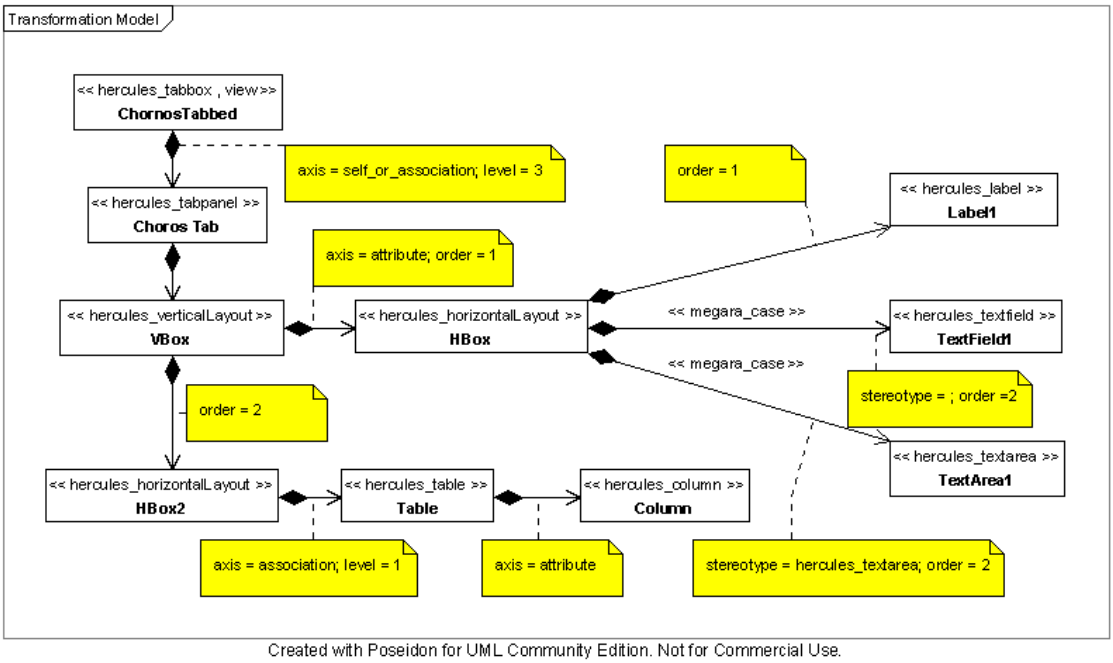
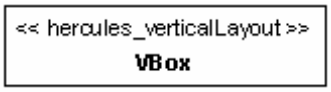
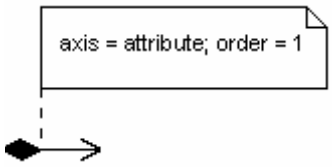
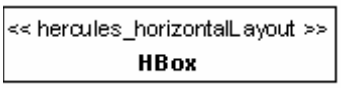


Figura 12 - O modelo de transformação do Hércules para o sistema Chronos

Tabela 1 - Sequência de execução das regras de transformação

Passo	Comando de Transformação (CT) / Padrão de Reconhecimento (PR)	Descrição
Passo 1 CT	<< hercules_tabbox , view >> **ChornosTabbed**	Criar no PSM um elemento com o estereótipo *hercules_tabbox*.
Passo 2 PR	axis = self_or_association; level = 3	Selecionar a classe ativa e as classes que estão nos três próximos níveis de associação. As classes selecionadas se tornaram ativas e o próximo comando será executado sobre elas.
Passo 3 CT	<< hercules_tabpanel >> **Choros Tab**	Criar no PSM um elemento com o estereótipo *hercules_tabpanel* para cada uma das classes selecionadas. Associar com o elemento criado no Passo 1.

Passo 4 PR		Mantém o mesmo conjunto de entidades ativas.
Passo 5 CT		Criar no PSM um elemento com o estereótipo *hercules_verticalLayout* para cada uma das classes selecionadas. Associar com os elementos criados no Passo 3.
Passo 6 PR		Para cada classe ativa, selecionar os atributos.
Passo 7 CT		Criar no PSM um elemento com o estereótipo *hercules_horizontalLayout* para cada um dos atributos selecionados. Associar com os elementos criados no Passo 5. O valor etiquetado order encontrado no passo anterior será copiado para essas associações e será utilizado na transformação de PSM para artefato.

Durante a transformação, a ferramenta navega simultaneamente pelos diagramas do modelo de transformação e do modelo independente de plataforma. No início do processamento, a vista ativa é a classe ChronosTabbed e a entidade ativa é a classe Projeto. A vista ativa é sempre uma das classes do diagrama do modelo de transformação e a entidade ativa é sempre uma das classes do diagrama independente de plataforma. A ferramenta cria no PSM uma classe que representa a vista ativa e possui o mesmo estereótipo que esta. A ferramenta lê os valores das marcas *axis* e *level*, e entende que os comandos posteriores devem ser aplicados para entidade ativa e para os três níveis de associações seguintes. No exemplo, a ferramenta cria quatro classes com o estereótipo <<*hercules_tabpanel*>>, essas classes representaram a vista das entidades *Projeto*, *Iteração*, *EstoriaDoUsuario* e *Tarefa.* Como a associação seguinte não possui

nenhuma marca, a ferramenta apenas cria uma classe com o estereótipo *<<hercules_verticalLayout>>* associada a cada uma das classes que foram criadas no passo anterior. Com os passos descritos até então, a ferramenta gera o diagrama da Figura 13.

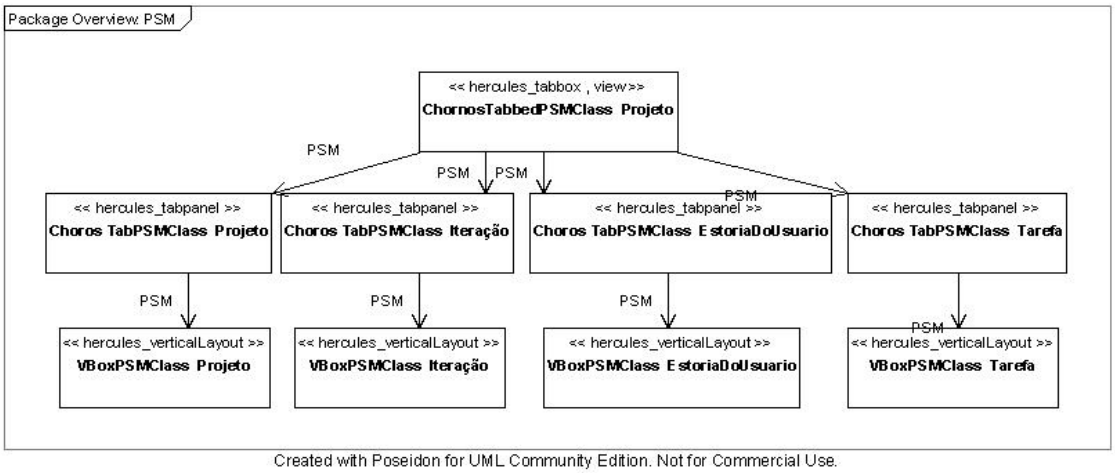


Figura 13 - PSM gerado pela ferramenta Megara – Parte 1

Continuando o processamento, a ferramenta seguirá as associações da classe VBox. A primeira associação leva a classe HBox e possui as marcas *axis* e *order*. A marca *axis* tem o valor *attribute*, o que significa que os próximos comandos serão aplicados para os atributos das entidades ativas. Cada uma das classes com o estereótipo *<<hercules_verticalLayout>>*, possui um valor diferente para entidade ativa. Na classe VBoxPSMClassProjeto, por exemplo, a entidade ativa é Projeto. Nas classes VBoxPSMClassIteração, VBoxPSMClassEstoriaDoUsuario e VBoxPSMClassTarefa, as entidades ativas são Iteração, EstoriaDoUsuario e Tarefa, respectivamente. A marca *order* será apenas copiada para o PSM, para que seja levada em consideração no momento da tradução de PSM para artefato.

Para cada um dos atributos das entidades ativas será criada no PSM, uma classe com o estereótipo *<<hercules_horizontalLayout>>*, que estará associada a uma classe com o estereótipo *<<hercules_label>>* e a uma outra classe. Esta última classe poderá possuir um estereótipo *<<hercules_textarea>>*, se o atributo que ela representa estiver marcado no PIM com este estereótipo. Caso nenhuma marca seja encontrada neste atributo, será criada uma classe com o estereótipo *<<hercules_textfield>>*.

As associações representadas no modelo de transfomação que ligam a classe HBox com as classes TextField1 e TextArea1 possuem o estereótipo

<<*megara_case*>>. Este estereótipo visa apenas mostrar para o usuário de forma gráfica que apenas uma das classes será levada para o PSM. Sem o estereótipo o usuário teria que selecionar a associação e observar seus valores etiquetados. Atualmente, os valores etiquetados não possuem representação gráfica nas ferramentas UML. A não visualização dos valores etiquetados prejudica a percepção da arquitetura pelo usuário da ferramenta. A Figura 14 mostra o PSM gerado após as transformações descritas nesta parte do texto. Na figura é mostrada apenas a geração para a entidade Projeto, as outras entidades serão transformadas de maneira análoga.

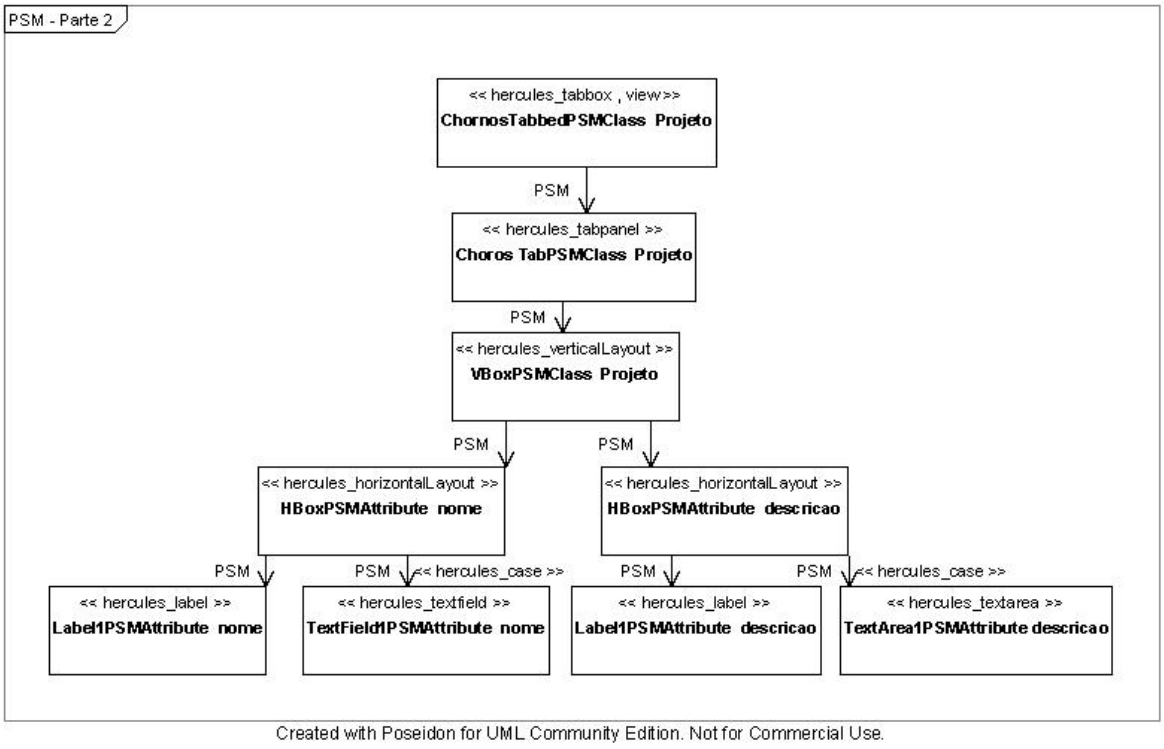


Figura 14 - PSM gerado pela ferramenta Megara - Parte 2

Seguindo a associação que liga as classes VBox e HBox2 no diagrama do modelo de transformação, a ferramenta encontrará apenas a marca *order* que será copiada para o PSM. Uma classe com o estereótipo <<hercules_horizontalLayout>> será criada. Na próxima associação do modelo de transformação, a ferramenta encontra duas marcas: *axis* e *level*. A ferramenta lê os valores das marcas *axis* e *level*, e entende que uma classe com o estereótipo <<*hercules_table*>> deve ser criada para o nível de associação seguinte. No caso da entidade ativa ser Projeto, uma tabela será construída para representar a entidade Iteração. A entidade Iteração passa a ser a entidade ativa. A próxima associação indica que deve ser criada uma classe com o estereótipo

<<*hercules_column*>> para cada um dos atributos da entidade ativa. A Figura 15 mostra o PSM gerado após as transformações descritas nesta parte do texto. Na figura é mostrada apenas a geração da entidade Projeto, as outras entidades serão transformadas de maneira análoga.

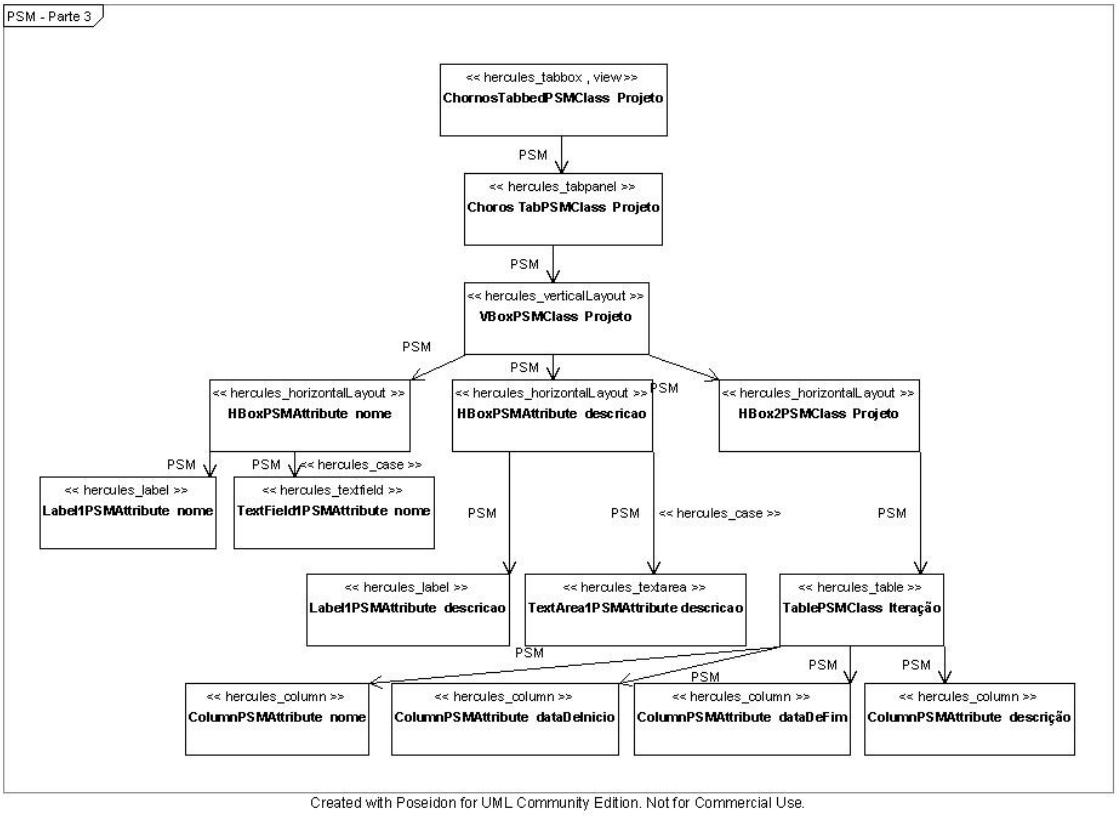


Figura 15 - PSM gerado pela ferramenta Megara - Parte 3

Caso o PSM gerado não satisfaça o usuário, ele pode alterar o diagrama conforme sua vontade. No caso de geração de interfaces gráficas, dificilmente a geração automática atenderá a todos os requisitos do usuário. No exemplo da Figura 15, para que o atributo descrição não apareça como coluna da tabela TablePSMClass Iteração, basta apagar a classe ColumnPSMAttribute descrição.

O PSM gerado pela ferramenta possui classes e alguns atributos. Os atributos gerados são o *datasource* e o *ref*, descritos no capítulo quatro. Esses atributos fazem a ligação entre o componente visual e o espelho de entidade que armazena a informação mostrada pelo componente. Nenhum outro atributo é gerado. Em uma etapa posterior, o diagrama do PSM é lido pela ferramenta e a descrição abstrata necessária para execução de um caso de uso sistema na plataforma Hércules é gerada. O formato da especificação do Hércules é um XML representando a linguagem XUL (eXtensible

User-interface Language). Neste passo, para cada classe presente no PSM, uma *tag* XML-XUL é criada.

A ferramenta oferece a opção de gerar o PSM com classes que possuam atributos. Neste caso, a ferramenta lê o arquivo DTD (Document Type Definitions) [HAROLD, 1999] que dita as regras para a formação do XML de especificação do Hércules procurando os atributos relativos a cada tag. O DTD da especificação do Hércules está no Anexo I. Apenas os atributos requeridos são criados. Se o atributo descrito no DTD possui um valor padrão, este valor é atribuído ao valor inicial do atributo da classe. Após a geração, o usuário pode alterar os detalhes que não ficaram conforme desejado. A Figura 16 mostra a tela obtida através da leitura pelo Hércules do arquivo XML gerado pela ferramenta Megara.

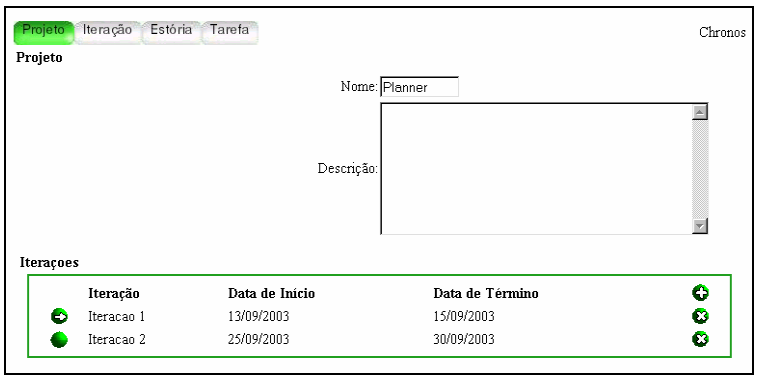


Figura 16 - Tela de edição da entidade Projeto

Pacotes Base e Architecture

Conforme dito anteriormente, os pacotes base e architecture poderão conter elementos derivados que são reutilizados em diagramas que representam um modelo de transformação. Supondo que o pacote base estivesse armazenando o elemento Linha mostrado na Figura 17, este elemento poderia ser usado no modelo de transformação mostrado na Figura 12. O modelo alterado é apresentado na Figura 18.

O pacote base deve ser utilizado quando o usuário desejar reusar os componentes derivados em outros sistemas. Para permitir a reutilização, o usuário deverá agregar o pacote base ao perfil UML existente. Quando os componentes derivados são específicos para o sistema, eles acabam por definir os requisitos que governam a interação do usuário com o sistema e devem ser criados no pacote architecture.

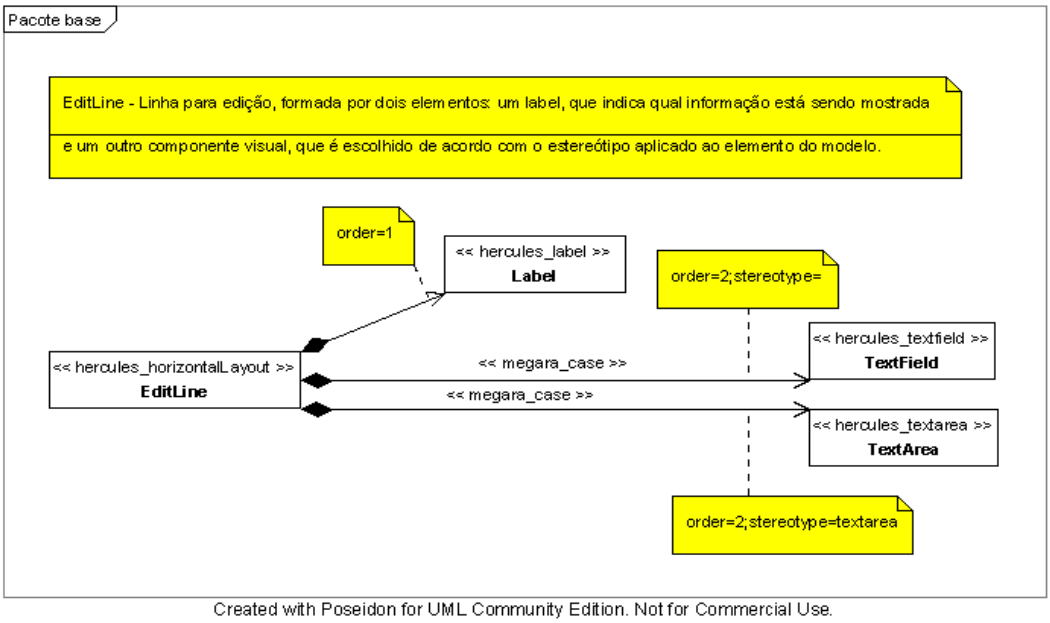


Figura 17 - Diagrama com exemplo de pacote base

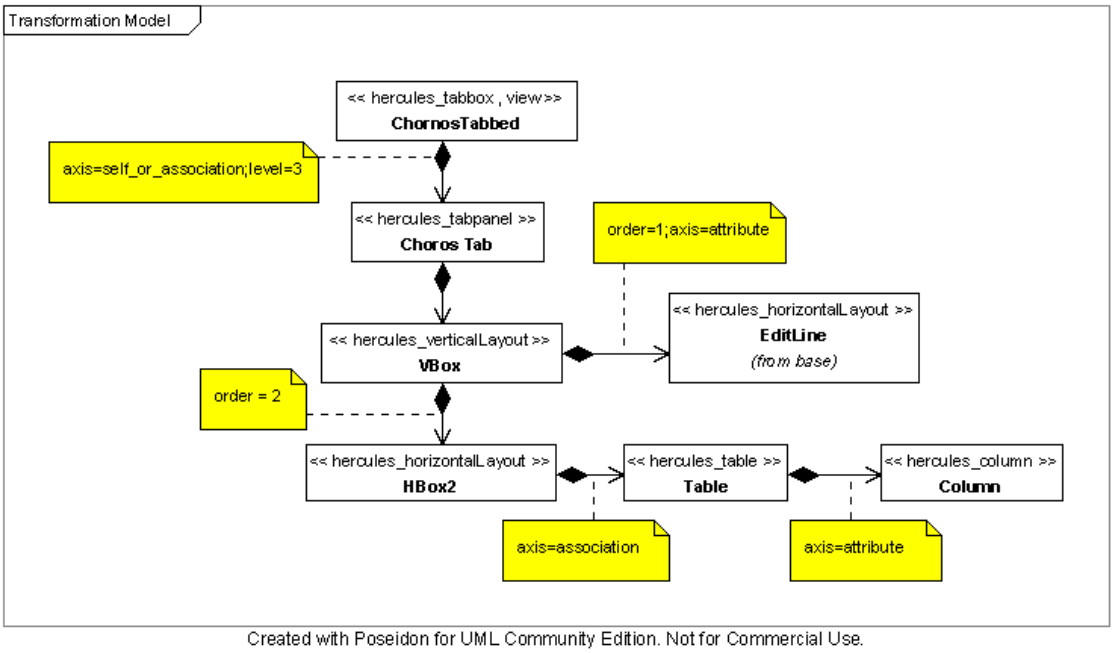


Figura 18 - Modelo de transformação utilizando elemento contido no pacote base

Passos do Processo

A equipe de desenvolvimento constrói o modelo independente de plataforma em uma ferramenta de modelagem. Esta pesquisa adotou a ferramenta Poseidon, por ter uma versão gratuita e ter o código aberto. Após a criação do PIM e do modelo de transformação, a ferramenta gera: toda a estrutura de pacotes, os diagramas que representam o modelo e a apresentação do caso de uso. O diagrama da camada de

Controle é criado pelo usuário.

O modelo gerado é incrementado pelo projetista. Esse modelo contém os detalhes necessários para a geração automática dos artefatos do Hércules. Quanto mais padronizada for a apresentação do sistema, menos trabalho a equipe terá em modificar os diagramas de telas. Com o modelo pronto, a ferramenta é utilizada para gerar os artefatos necessários para a execução do módulo sobre a plataforma Hércules.

Todos os passos da ferramenta de geração seguem o mesmo padrão: o modelo é construído na ferramenta de modelagem e salvo em um arquivo XMI (XML Metadata Interchange) [OMG, 2004]. Uma biblioteca utilizada pela ferramenta carrega o arquivo XMI em uma estrutura de classes. Enquanto essa estrutura é percorrida, uma nova estrutura é gerada. Essa nova estrutura dará origem ao modelo alvo. A Figura 19 mostra a utilização da ferramenta Megara na transformação de modelos em artefatos. Os artefatos gerados podem ser outros modelos, arquivos XML ou classes Java.

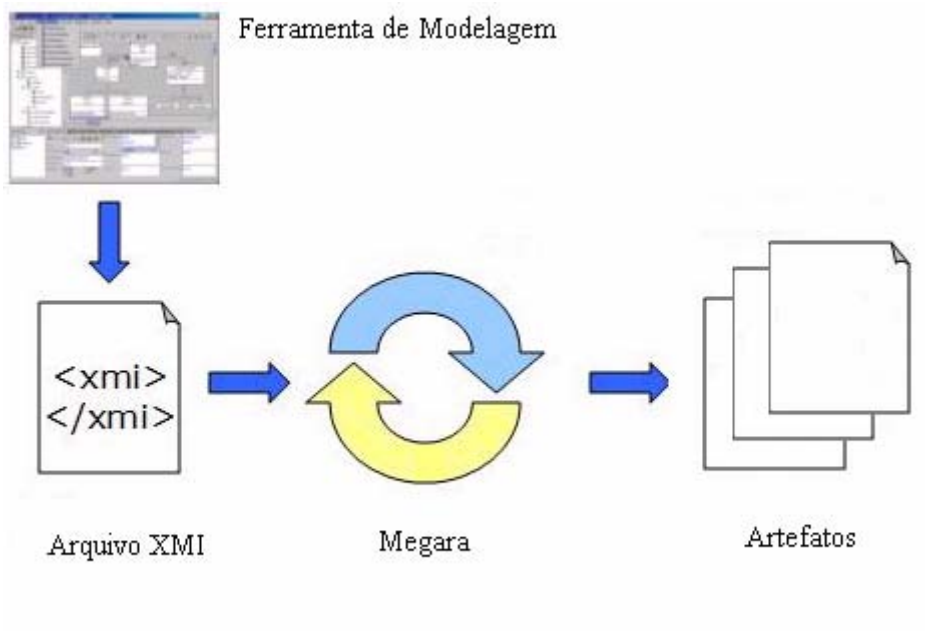


Figura 19- Geração de Artefatos através da ferramenta Megara

6. Implementação

A implementação da ferramenta foi baseada no paradigma orientado a objetos. Este paradigma oferece benefícios como o encapsulamento, partição natural do problema em classes e abstração. Outra grande promessa desse paradigma é a garantia de reutilização, mas essa promessa freqüentemente não é realizada [AMBLER, 1998]. A reutilização é obtida não somente através da utilização da orientação a objetos, mas também pela utilização de técnicas de desenvolvimento que garantem o reuso. O desenvolvimento baseado em componentes e o uso de padrões de projeto são exemplos de técnicas que promovem o reuso.

A reutilização de componentes refere-se ao uso de componentes pré-construídos e totalmente encapsulados no desenvolvimento de suas aplicações. Componentes são tipicamente auto-suficientes e encapsulam um único conceito. Um desenvolvedor pode utilizar componentes criados em outros projetos ou por outros desenvolvedores. A utilização de componentes facilita a manutenibilidade porque a implementação de um componente pode ser modificada sem interferir no funcionamento do sistema, onde ele está inserido.

A reutilização de padrões refere-se ao uso de abordagens publicamente documentadas para solução de problemas comuns. A documentação mostra como desenvolvedores experientes resolveram um determinado problema. A aplicação de padrões provê uma reutilização de alto nível, onde é a idéia do desenvolvedor que é reutilizada. Padrões de projeto evitam que o projetista perca tempo resolvendo um problema que já têm uma boa solução conhecida.

6.1. Estrutura componentizada e Aplicação de padrões

O usuário do processo proposto deve utilizar a ferramenta Poseidon na fase de modelagem. Esta ferramenta possui uma versão grátis que pode ser utilizada para fins não comerciais. Após a criação do modelo independente, ele deve ser salvo em um arquivo XMI. XMI é o mapeamento de MOF para XML e é usado para transferência de metadados entre ferramentas Poseidon e Megara.

A ferramenta Megara utiliza componentes (bibliotecas) pré-existentes seguindo a prática da reutilização de componentes. Essa reutilização garante uma diminuição no tempo de desenvolvimento, pois classes que teriam que ser implementadas, revisadas e testadas são substituídas por outras que já passaram por todo o processo.

A Megara utiliza a biblioteca MDR (Metadata Repository) [NETBEANS.ORG, 2006] para ler as informações do arquivo XMI e criar uma estrutura de objetos que corresponda a informação contida neste arquivo. A biblioteca possui uma API(Application Program Interface), que implementa a especificação JMI de mapeamento MOF para Java, para acessar e manipular os dados da estrutura.

Megara constrói sua própria estrutura de objetos baseada na estrutura criada pelo MDR. A estrutura é formada por instâncias das classes do pacote *ceryneia*, que contém um conjunto de classes que representam o meta-modelo UML. A Figura 20 mostra o diagrama de classes do componente Ceryneia. A nova estrutura é criada de forma que esta aceite a visita de uma classe que implementa o padrão de projeto Visitor[GAMMA *et Al*,1995]. A interação entre a ferramenta Megara e os componentes MDR e Ceryneia é mostrada na Figura 21.

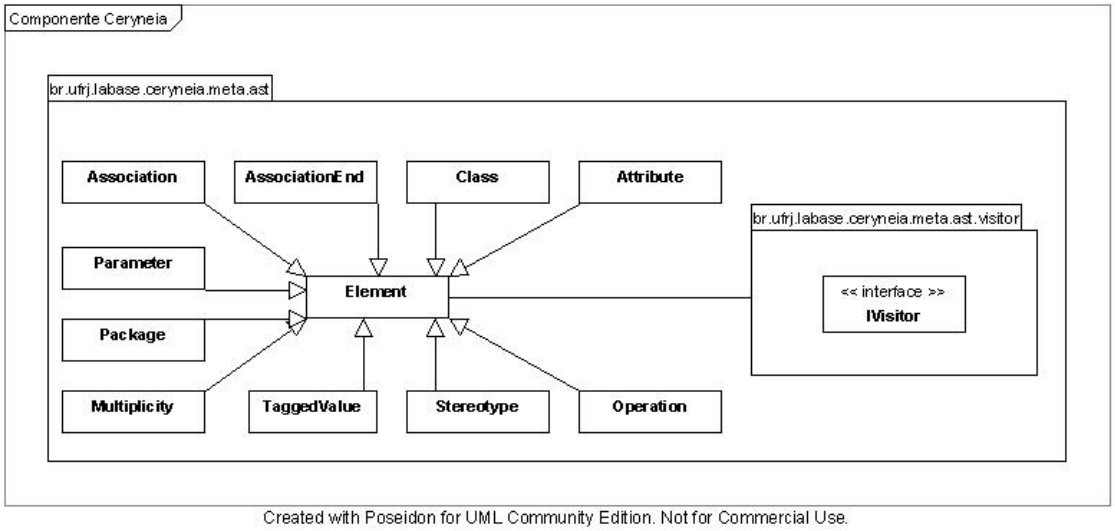


Figura 20 - Diagrama de classes do componente Ceryneia

Essa nova estrutura contém as classes do modelo independente de plataforma, as classes do modelo de transformação e conterá, após a geração realizada pela Megara, as classes do modelo específico para a plataforma Hércules. As classes representadas no PIM com o estereótipo <<*binder*>> fazem a ligação entre o PIM e o modelo de transformação.

A estrutura é percorrida através da classe MatchPatternVisitor, uma implementação do padrão de projeto Visitor. O MatchPatternVisitor encontra classes que obedecem a um determinado padrão. Por exemplo, pode encontrar classes que herdem de classes que possuem um determinado estereótipo. No caso da estrutura, procura

classes que possuam o estereótipo <<*binder*>> e estejam ligadas a uma e somente uma classe com o estereótipo <<*model*>> e a uma e somente uma classe com o estereótipo <<*view*>>.

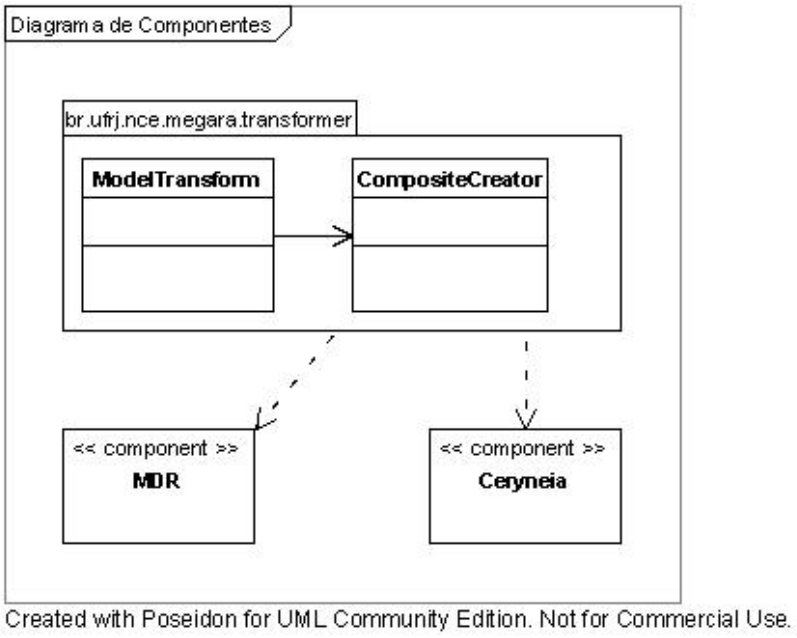


Figura 21 - Diagrama de componentes mostrando interação entre Megara, Ceryneia e MDR

Para cada uma das classes com o estereótipo <<*binder*>> encontrada, uma tela inteira, parte de uma tela ou um conjunto de telas é criado. O padrão Visitor foi novamente utilizado na etapa de construção do diagrama de interface gráfica, mas desta vez o padrão foi levemente alterado. O padrão percorre ao mesmo tempo as estruturas do PIM, que contém uma representação das entidades, e do modelo de transformação, que descreve a vista. O Visitor vai percorrendo o modelo de transformação e criando os elementos da vista baseado na entidade ativa. Para que o Visitor possa percorrer corretamente os diagramas, é importante que a navegabilidade entre as classes do diagrama esteja descrita de forma correta.

A entidade de modelo ativa é alterada quando um valor etiquetado com o nome *axis* é encontrado. Dependendo do valor atribuído a esse valor etiquetado, uma lista diferente de comandos deve ser executada. Para facilitar a identificação dos comandos que devem ser executados, o padrão *Command*[GAMMA *et Al*,1995] foi implementado. A utilização do padrão *Command* evita uma cadeia de comandos *if* para identificar o valor atribuído.

Um objeto *Command* é criado para cada uma das combinações de nome e valor, esse objeto armazena o conjunto de instruções necessárias à execução da tarefa. Os objetos *Commands* implementam uma *interface*, essa *interface* define o método que quando invocado executa o conjunto de instruções apropriado para uma determinada combinação de nome e valor. Quando o valor etiquetado *axis* é encontrado, a ferramenta seleciona o respectivo *Command* e executa o método definido na *interface*. A única alteração necessária para a inclusão de novas combinações é a criação de novos *Commands*. A Figura 22 mostra a utilização do padrão de projeto *Command*.

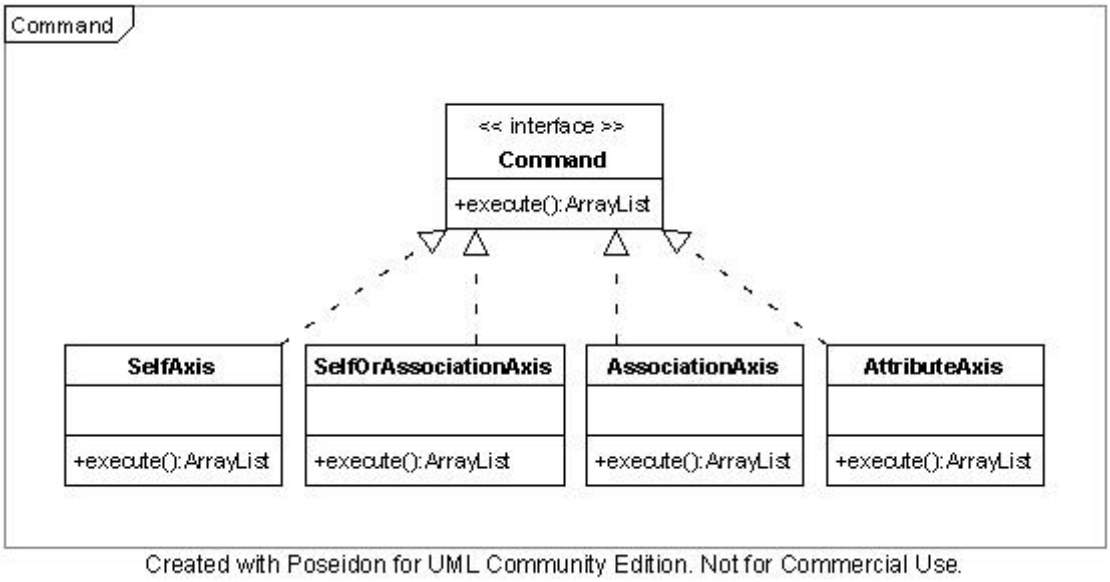


Figura 22 - Implementação do padrão de projeto Command

Ao final da criação dos novos objetos na estrutura da Megara, a estrutura é convertida novamente para a estrutura do MDR. Essa conversão é baseada no padrão Builder[GAMMA *et Al*,1995]. Em seguida, todas as informações são armazenadas em um arquivo XMI. Este arquivo pode ser aberto e alterado na ferramenta Poseidon. O engenho de transformação é mostrado na Figura 23.

Se o usuário escolhe preencher o PSM com os atributos da classe, após terminar a transformação, a ferramenta chama a biblioteca DTDParser[WUTKA, 2003]. Esta biblioteca é responsável por encontrar os atributos pertencentes a um determinado elemento. O DTDParser lê o arquivo DTD e retorna um objeto DTD. Esse objeto é similar ao objeto Document no modelo DOM(Document Object Model). O objeto DTD possui estrutura de árvore e métodos que facilitam a busca de um determinado elemento na árvore.

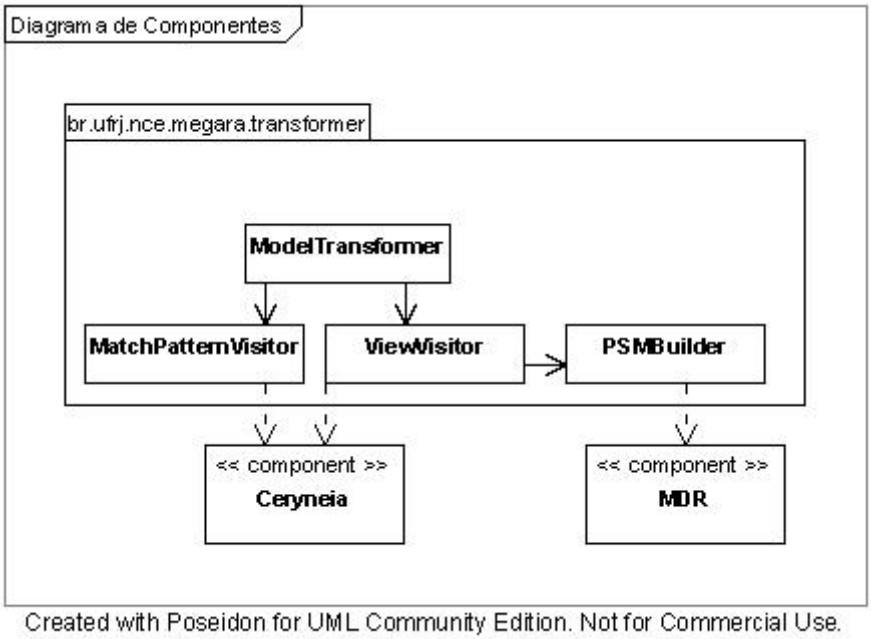


Figura 23 - Diagrama de componentes representando o engenho de transformação

Na transformação de PSM para artefato, o componente MDR é novamente utilizado para leitura dos componentes visuais representados no PSM. O padrão de projeto Visitor foi utilizado para percorrer a estrutura de classes e o padrão Builder foi utilizado para a criação do arquivo XML.

7. Caso de Teste

O caso de teste foi baseado nos casos de uso do Sistema de Gestão Acadêmica (SiGA). O SiGA é um sistema desenvolvido com a finalidade de integrar os sistemas de registro acadêmico da UFRJ. A principal função do sistema é atender a todos os serviços prestados pela Divisão de Registro de Estudantes.

Alguns dos serviços disponíveis no sistema são: matrícula de alunos, inscrição em disciplinas, previsão de turmas, lançamento de notas, emissão de boletim e histórico escolar. Alunos, professores e funcionários têm acesso a esses serviços através da rede. As opções disponíveis a um usuário dependem do seu perfil no sistema. Por exemplo, a opção de lançamento de notas não está disponível a um usuário que possua apenas o perfil de aluno. O serviço de previsão de turmas servirá como caso de teste para o processo de desenvolvimento proposto. Devido a complexidade do caso de uso, algumas simplificações foram feitas.

7.1. A previsão de turmas

A previsão de turmas é um dos serviços prestados pelo SIGA. O objetivo principal desse serviço é o cadastramento e a validação de turmas da UFRJ. Esse serviço está disponível somente para funcionários da UFRJ que possuam o perfil adequado.

Caso de uso

O caso de uso Previsão de turmas é composto por quatro telas principais. A primeira tela oferece ao usuário a possibilidade de consultar as turmas já criadas. Alguns parâmetros devem ser passados para que a consulta seja realizada. O resultado da consulta é a lista de turmas que preenche os requisitos definidos pelos parâmetros. Esse resultado é apresentado na segunda tela do caso de uso, onde uma das turmas pode ser selecionada para edição. As informações sobre a turma selecionada podem ser alteradas na tela de edição. Quando alguma das informações é modificada, os novos dados são validados através das regras de negócios. O resultado da validação é uma lista de mensagens mostrada em uma nova tela.

Análise

Uma turma é representada por uma entidade de domínio, que possui alguns atributos simples e algumas coleções formadas de outras entidades. Alguns dos atributos simples são:

- situacaoTurma – Esse atributo descreve a situação da turma. Uma turma está em uma das seguintes situações: candidata, pendente, ativa ou fechada.
- segmentacao – O atributo segmentação indica para qual segmentação a turma foi aberta. A UFRJ adota dois tipos de segmentação: a segmentação por blocos e a segmentação por período. O formato dessa informação é ano-período-bloco.
- turno – Esse atributo define o período do dia em que as aulas são ministradas. O atributo pode assumir os seguintes valores: Manhã, Tarde, Noite, Parcial ou Integral.

Os atributos que representam coleções são:

- aAsTurma – Uma disciplina nada mais é do que uma ementa. A turma representa a oferta da disciplina em uma determinada segmentação. Uma turma pode ser criada para atender a mais de uma disciplina, que possuam ementas equivalentes. O atributo aAsTurma representa o conjunto de disciplinas equivalentes que a turma oferece.
- docentesTurma – Um ou mais professores ministram aulas para uma turma, a lista de professores é armazenada no atributo docentesTurma.
- horariosTurma – As aulas de uma turma podem ser ministradas em vários horários. Uma turma de Física para Informática possui uma carga horária semanal de seis horas que pode ser dividida, por exemplo, em três horários de aula: segunda-feira de 8 às 10, quarta-feira de 8 às 10 e sexta-feira de 8 às 10. Essa coleção de horários é armazenada na coleção horariosTurma.
    - horariosLocais - As aulas podem ser ministradas em locais diferentes em cada horário, ou podem ser ministradas em mais de um local no mesmo horário. Cada horarioTurma possui um atributo chamado horarioLocal que armazena os locais de aulas alocados pela turma.
        - participacoesTurma – Em cada um dos locais de aula pode existir um ou mais professores lecionando. A coleção de participaçõesTurma armazena a informação sobre quais professores lecionaram em um determinado lugar e o período em que isso ocorreu.

A figura a seguir mostra o diagrama de classes do caso de uso de Previsão de Turmas.

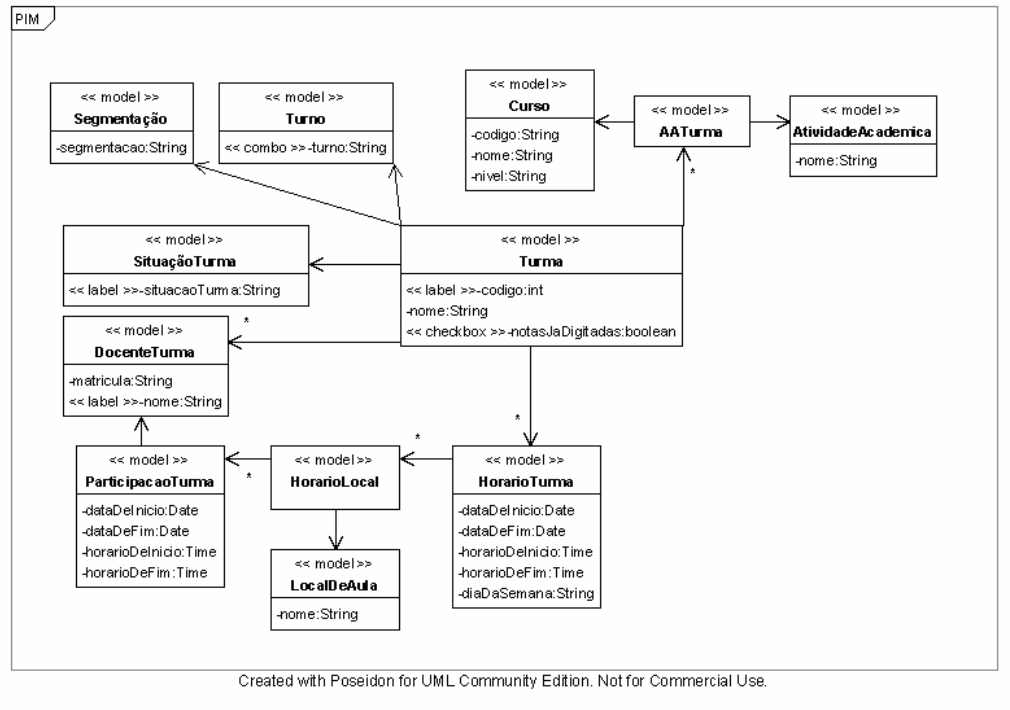


Figura 24 – Diagrama de classes do caso de uso Previsão de Turma

Projeto

A primeira tela do caso de uso é a tela que recebe os parâmetros da consulta. Qualquer subconjunto de parâmetros pode ser utilizado na consulta. O campo opção de busca define os parâmetros que participam da consulta. Os parâmetros que não participam possuem esse campo preenchido com o valor "*qualquer*". Caso o usuário deseje que um determinado parâmetro faça parte da consulta, ele deve preencher o campo opção com um valor adequado. O usuário pode, por exemplo, querer saber quais as turmas ativas da segmentação 2003-1-0. A tela que representa esta consulta pode ser visualizada na figura abaixo.

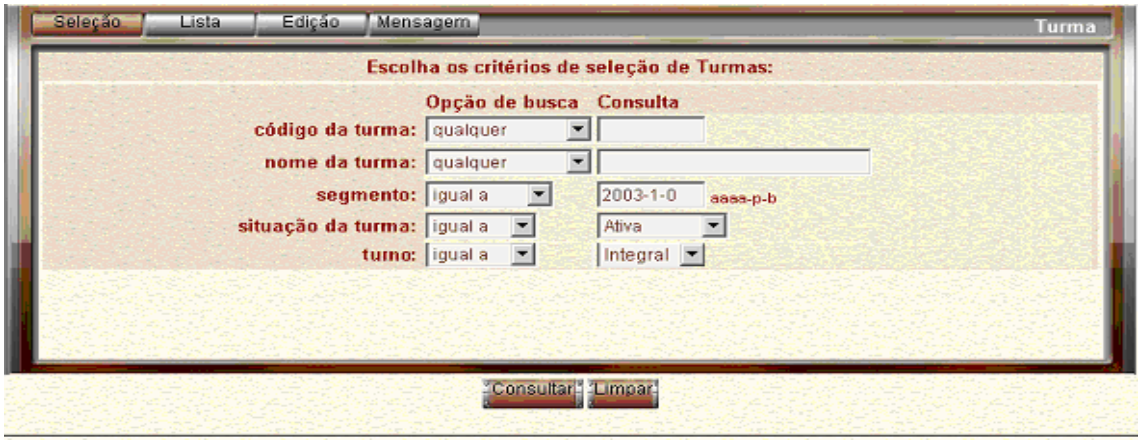


Figura 25 - A tela de consulta da Previsão de Turma

A tela de consulta apresenta os botões “consultar” e “limpar”. O botão “limpar” faz com que todos os campos de opção de busca passem para o valor “*qualquer*” e limpa todos os valores de consulta. Enquanto que o botão “consultar” é utilizado para efetuar a consulta. O sistema mostra a tela de lista quando o usuário clica o botão consultar. Essa tela contém a lista de turmas que preenchem os requisitos da consulta.

Figura 26 - Tela de Lista

O usuário pode escolher uma turma da lista e editar os dados dessa turma. A figura abaixo mostra a tela de edição para a turma de código 14800. A tela de edição possui várias outras partes que não estão sendo mostradas, são elas: horários, locais, participações, docentes e disciplinas.

Figura 27 - Tela de edição de turma

O usuário pode modificar dados sobre a turma e clicar no botão “alterar”. A turma passa por uma etapa de validação, mensagens são mostradas na tela de mensagens caso existam erros nessa etapa. Se o usuário clica no botão “incluir”, uma nova turma é criada à partir das informações obtidas da tela. Ao clicar nos botões “excluir” e “limpar” são realizadas, respectivamente, as tarefas de exclusão da turma e limpeza dos campos da tela de edição.

Os passos para que as telas do caso de uso sejam criadas são: criação do PIM; criação do modelo de transformação para o caso de uso; geração do PSM através da ferramenta Megara; alteração pelo desenvolvedor dos diagramas gerados, para que atendam a todas as expectativas do usuário final; e geração da descrição em XML do Hércules.

A criação do PIM é baseada no diagrama de classes do caso de uso mostrado na Figura 24. A alteração necessária no diagrama é a adição das classes com estereótipos <<*binder*>> e <<*view*>>. Uma primeira possibilidade para criação do PIM é a adição de uma classe <<*binder*>> e uma classe <<*view*>> para cada uma das telas do caso de uso, conforme diagrama mostrado na Figura 28.

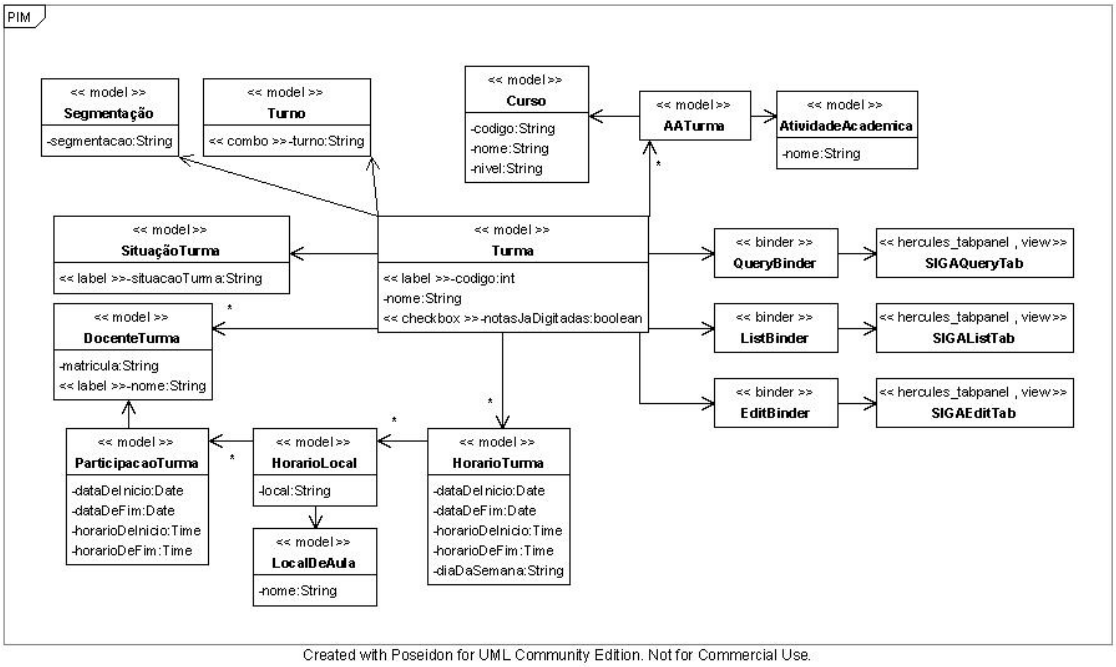


Figura 28 - Diagrama do modelo independente de plataforma para o caso de uso Previsão de Turma – Primeira proposta.

Uma melhor representação é obtida se for incluída somente uma classe com estereótipo <<*view*>> que contenha as três telas, como mostrado na Figura 29. Com

essa representação, apenas duas classes são incluídas no modelo independente, diminuindo a diferença entre o diagrama de classes do caso de uso e o PIM. Como a ferramenta é capaz de gerar o PSM para as duas propostas de PIM, a segunda proposta foi escolhida.

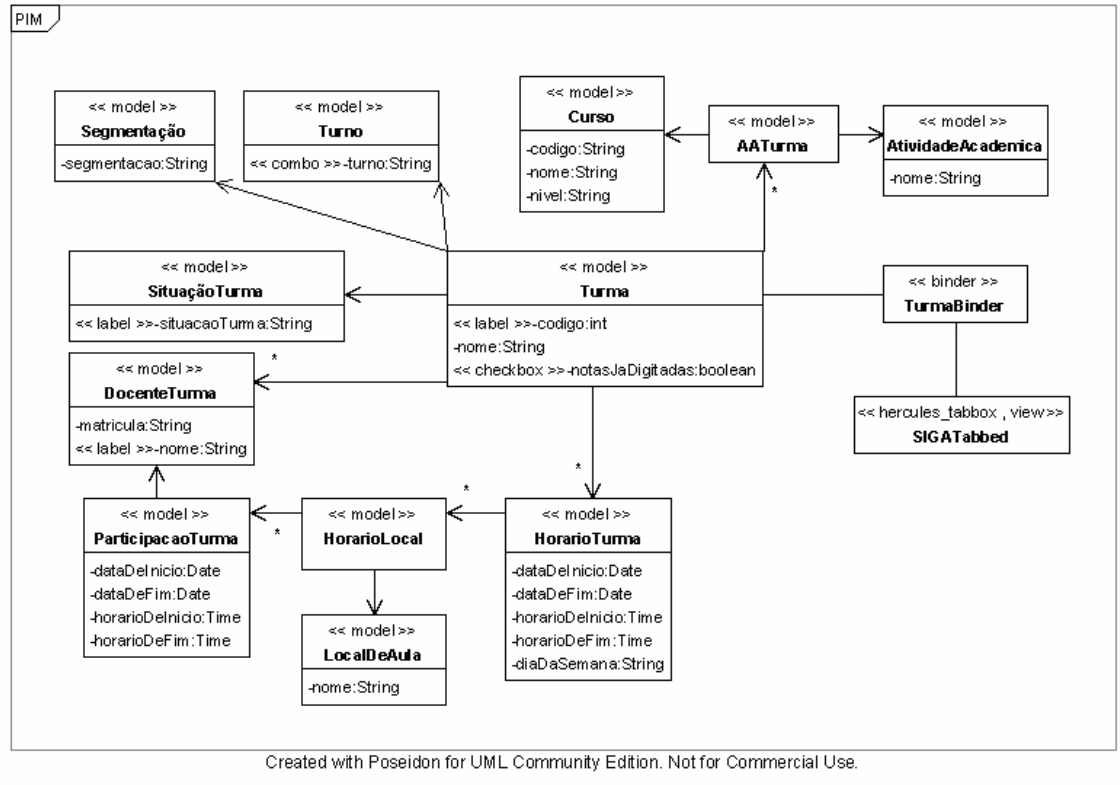


Figura 29 - Diagrama do modelo independente de plataforma para o caso de uso de Previsão de Turma– Segunda Proposta

O diagrama do modelo de transformação, mostrado na Figura 30, define como as entidades representadas no PIM são apresentadas para o usuário do sistema. Este diagrama contém elementos que representam componentes visuais, sendo que alguns deles foram especificados pelo arquiteto nos pacotes *base* e *arquitecture*.

O trabalho que o desenvolvedor terá ao criar o modelo de transformação para o caso de uso dependerá da qualidade dos elementos criados pelo arquiteto. Se o arquiteto padroniza a interface gráfica do sistema e cria componentes para diversos tipos de situação, existe a possibilidade do desenvolvedor associar a classe que possui o estereótipo <<*binder*>> diretamente a um elemento descrito no pacote *architecture*, sem ser necessária a criação de um modelo de transformação específico para o caso de uso.

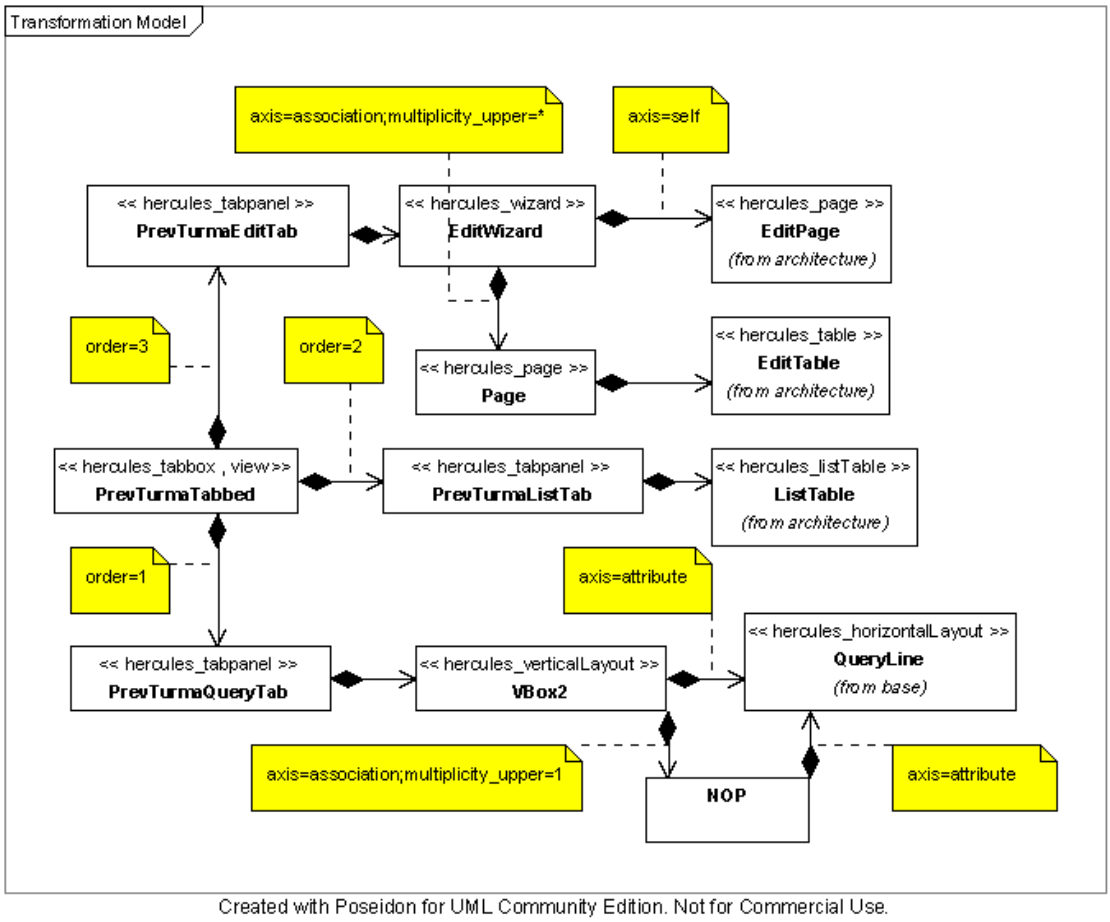


Figura 30 - Diagrama do modelo de transformação para o caso de uso Previsão de Turma

A ferramenta começa a percorrer o diagrama acima, a partir do elemento PrevTurmaTabbed, uma classe com o estereótipo <<hercules_tabbox>> é criada no PSM. Em seguida, a ferramenta segue a associação com o valor etiquetado *order* igual a um, e começa a criar as classes que darão origem a tela de consulta mostrada na Figura 25. A ferramenta cria as classes com o estereótipo <<hercules_tabpanel>> e <<hercules_verticalLayout>> no PSM.

Nesse momento, a ferramenta está na classe VBox2, seguindo pela associação que tem o valor etiquetado *axis* igual a *attribute*, a ferramenta cria um elemento *QueryLine* para cada um dos atributos ativos, que neste caso são os atributos da classe Turma. Um *QueryLine* é um elemento derivado que está definido no pacote base. O pacote base vai sendo atualizado a cada projeto, de forma que cada vez possua mais elementos reutilizáveis. O pacote base pode ser visualizado na Figura 31. A maior oferta de elementos facilita a tarefa de criação do modelo de transformação do caso de uso.

Na tela de consulta requerida pelo usuário, os atributos das entidades que

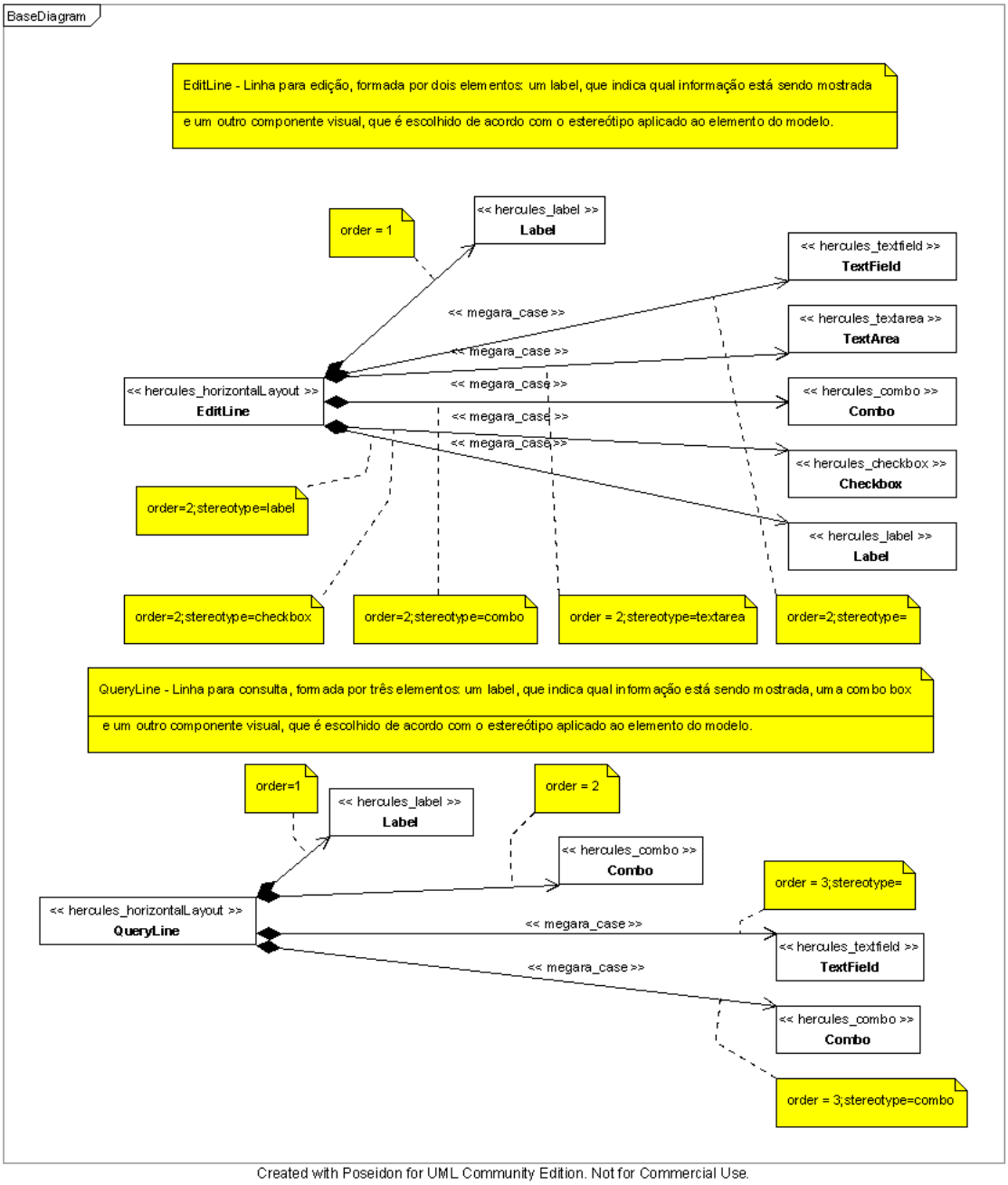


Figura 31 - Diagrama do pacote base

estão associadas a classe Turma e que possuem multiplicidade máxima igual a um, também devem ser utilizadas como parâmetros da consulta. Para possibilitar a inclusão desses atributos na tela de consulta, dois novos elementos foram criados: um novo padrão de reconhecimento chamado *multiplicity_upper* e um novo comando de transformação chamado NOP. O novo padrão de reconhecimento somente deve ser utilizado em combinação com a marca *axis* preeenchida com os valores *association* ou *self_or_association*. O padrão serve como um filtro para as entidades a serem tratadas

na transformação, selecionando apenas as classes que tiverem multiplicidade máxima igual a indicada no valor etiquetado. O comando NOP não faz nada, ele apenas é utilizado porque dois comandos axis não podem ser utilizados na mesma associação, pois existe a possibilidade deles não serem executados na ordem desejada.

Voltando para a classe VBox2 e seguindo pela associação que tem a marca *axis* com valor igual a *association* e a marca *multiplicity_upper* igual a um, as classes que possuem associação de multiplicidade um para um com a entidade ativa serão selecionadas. Em seguida, o comando NOP será encontrado e nada ocorrerá. O próximo passo será a criação de um elemento *QueryLine* para cada um dos atributos pertencentes as classes atualmente selecionadas. Como resultado da execução dos passos mencionados, a representação da tela de consulta no PSM foi criada.

As telas de resultado da consulta e de edição também são criadas em decorrência da aplicação das regras descritas no modelo de transformação sobre o modelo independente de plataforma. Nessas telas são usados elementos derivados descritos no pacote *architecture*. Esse pacote é mostrado na Figura 32. O PSM gerado pela ferramenta para o caso de uso da Previsão de Turma encontra-se no Anexo II e foi dividido em três partes, onde cada parte representa uma tela do caso de uso, para permitir uma melhor visualização.

A utilização do pacote architecture permite a representação e a reutilização da arquitetura da interface gráfica do sistema. De forma análoga ao pacote base, o pacote architecture também irá se aperfeiçoando ao longo do tempo, sendo também aperfeiçoado o processo como conseqüência.

O caso de uso Cadastrar Aluno foi utilizado para mostrar a reutilização dos elementos definidos pelo arquiteto. A Figura 33 mostra o modelo independente de plataforma do caso de uso e a Figura 34 mostra o modelo de transformação. Com a aplicação do modelo de transformação sobre o PIM, a ferramenta cria o PSM para as telas apresentadas na Figura 35, na Figura 36 e na Figura 37.

O processo proposto foi seguido para os casos de teste e a especificação abstrata do caso de uso foi gerada. Após a geração do arquivo XML, o desenvolvedor apenas teve que arrumar os detalhes relativos a aparência, como cores e figuras. Todos os outros componentes de tela foram gerados como requerido. Isso ocorreu porque após a

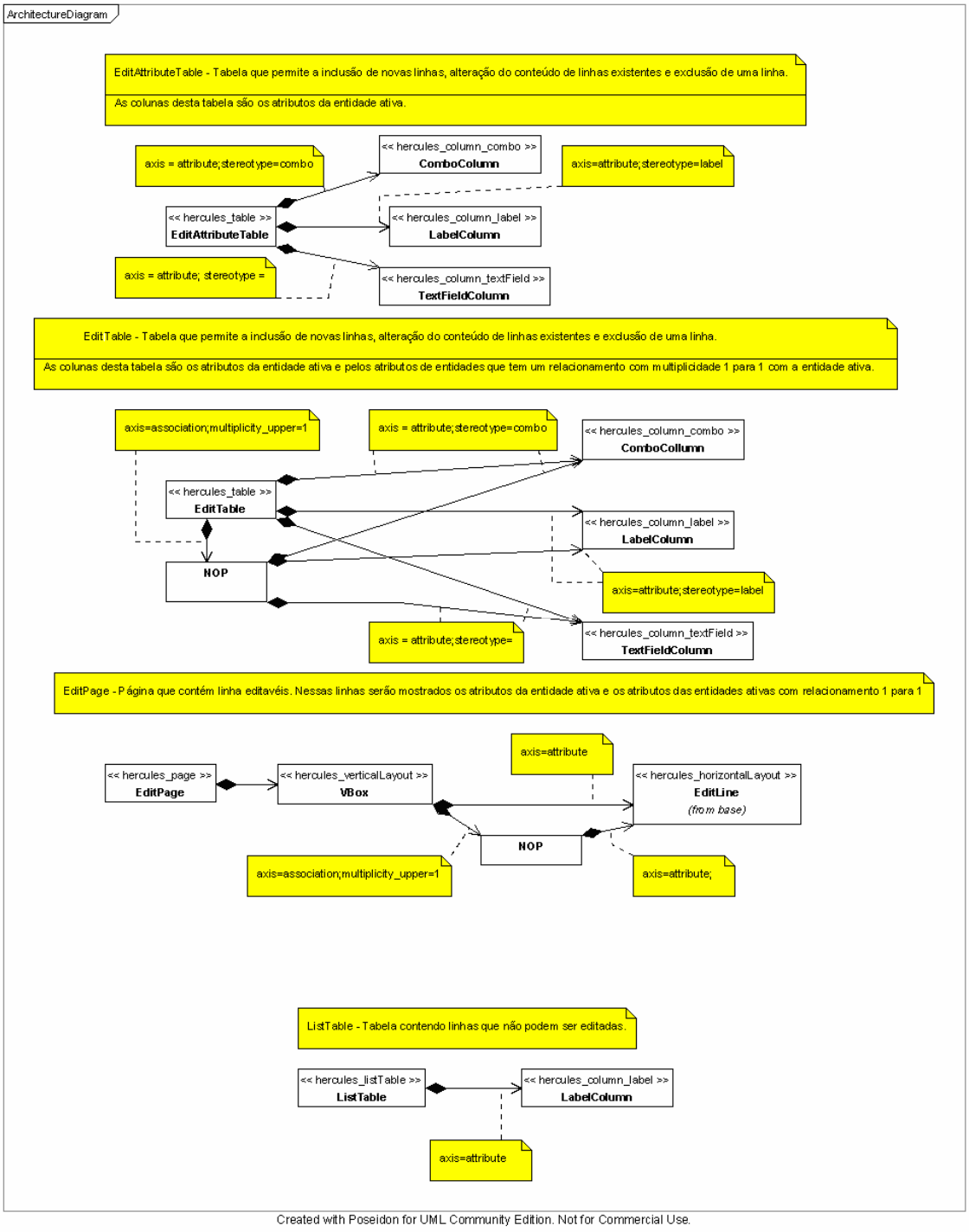


Figura 32 - Diagrama de arquitetura do sistema SiGA

geração do diagrama específico, o usuário pode interferir nesse diagrama, proporcionando uma entrada aperfeiçoada para a fase seguinte do processo.

A extensibilidade da linguagem permite que o PSM gerado se aproxime cada vez mais da tela requisitada pelo cliente. Com o continuo aperfeiçoamento da

linguagem, o número de alterações no PSM tende a diminuir, concentrando o desenvolvimento na fase de criação do modelo independente de plataforma.

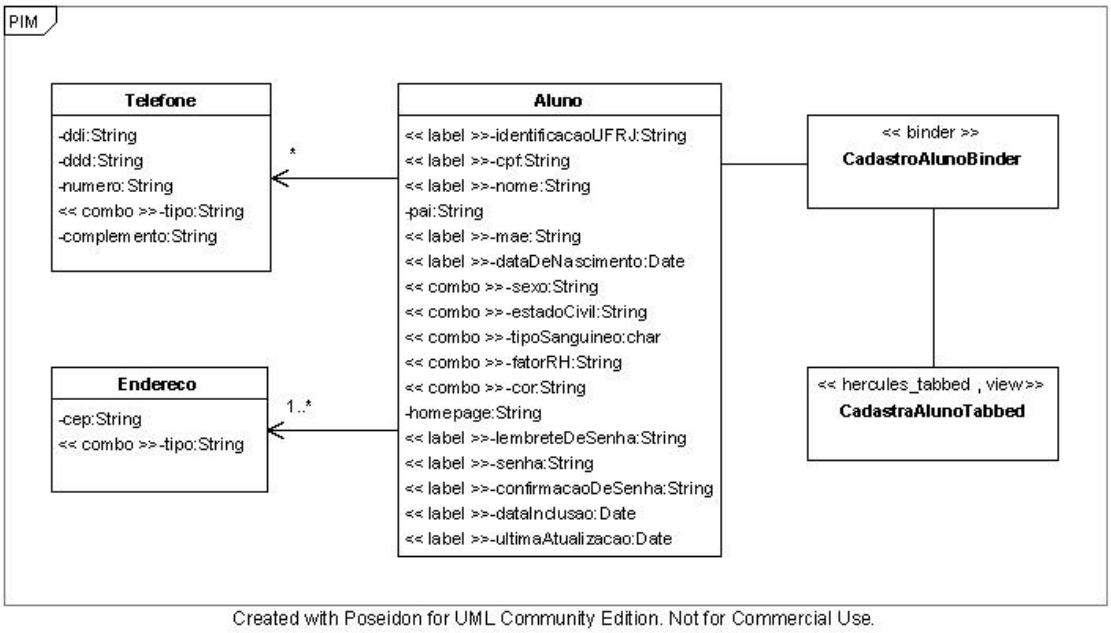


Figura 33 -Modelo independente de plataforma do caso de uso Cadastrar Aluno

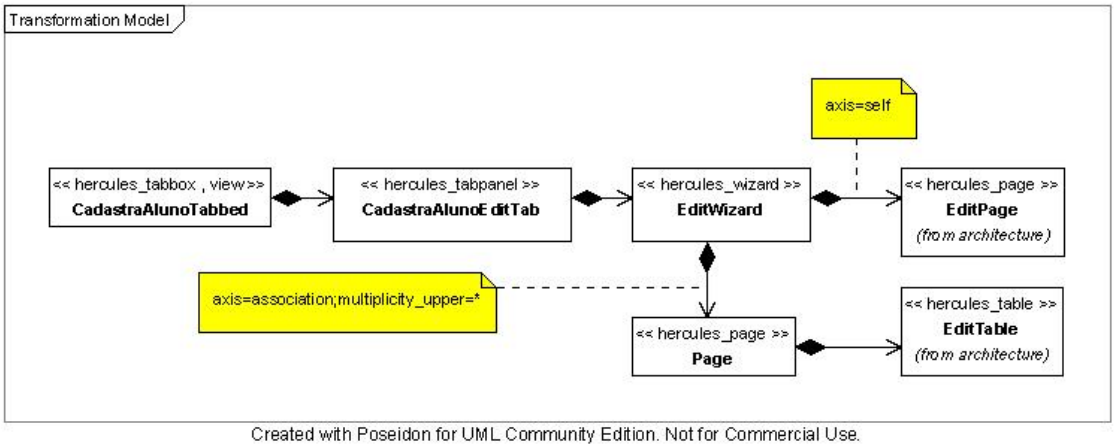


Figura 34 - Modelo de transformação para o caso de uso Cadastrar Aluno

Figura 35 - Tela do caso de uso Cadastrar Aluno
Fonte: Sistema de Registro Acadêmico da UFRJ

Figura 36 - Tela do caso de uso Cadastrar Aluno
Fonte: Sistema de Registro Acadêmico da UFRJ

Figura 37 - Tela do caso de uso Cadastrar Aluno
Fonte: Sistema de Registro Acadêmico da UFRJ

8. Avaliação e Conclusão

O processo apresentado nesta dissertação complementa a proposta de arquitetura dirigida pelo modelo. Além de representar a arquitetura do sistema na forma de modelos, o processo propõem que as regras de transformação também sejam representadas desta forma.

As contribuições trazidas com a proposição do processo são: Aumento do nível de abstração do processo de desenvolvimento; Flexibilidade do código gerado; Representação e reutilização da arquitetura da interface do sistema; Representação semântica das regras de transformação; e Extensibilidade da linguagem de transformação.

A utilização da ferramenta elevou o nível de abstração, pois ao invés de escrever um arquivo XML, os desenvolvedores apenas projetaram diagramas UML. Baseada no diagrama PSM, a ferramenta gerou os artefatos necessários para execução de um caso de uso do sistema na plataforma Hércules.

Um problema da geração automática é que o código gerado nem sempre é como o desenvolvedor gostaria. A utilização de regras de transformação descritas pelo desenvolvedor garante a flexibilidade de geração do código, pois o desenvolvedor define como o modelo será transformado.

As regras de transformação foram descritas na linguagem UML. A descrição em UML proporcionou a representação da arquitetura da interface do sistema e tornou possível a reutilização tanto das transformações como dos elementos da interface gráfica.

O reuso é garantido para todo o sistema, pois basta utilizar a mesma classe em um outro lugar do mesmo sistema ou de um sistema diferente. A criação de classes, que serão reutilizadas, dentro dos pacotes *base* ou *architecture* facilita a tarefa de encontrar um conjunto de classes existente que satisfaça uma nova demanda. Os elementos reutilizáveis ficam armazenados em um lugar conhecido e não ficam espalhados pelo modelo do sistema. No caso do pacote *base*, a representação em um pacote isolado facilita o reuso dos elementos em outros sistemas, pois é fácil identificar os elementos que devem ser adicionados ao perfil UML. No caso do pacote *architecture*, é permitida a representação do conhecimento arquitetônico da interface do sistema.

A reutilização das regras também é facilitada devido a sua representação semântica. As regras não são dependentes de nomes de classes e de atributos, elas se baseiam em estereótipos e na função do elemento dentro do meta-modelo da UML. O reuso diminui a complexidade do modelo de transformação do caso de uso, pois ao invés do projetista ter que criar uma nova transformação, ele apenas utiliza uma transformação definida pelo arquiteto do sistema. A definição das regras de transformação pelo arquiteto garante a padronização da interface do sistema. Entidades que obedecem ao mesmo formato são renderizadas sempre da mesma maneira, impondo um padrão para o sistema.

Durante a etapa de definição das regras de transformação, o arquiteto poderá perceber que um novo comando ou um novo padrão de reconhecimento facilitará a tarefa de construção das regras. A linguagem e a ferramenta de transformação são facilmente extensíveis, logo o arquiteto poderá incorporar esses novos elementos a linguagem.

Na linguagem, novos eixos podem ser criados, proporcionando uma geração cada vez mais próxima da necessidade real do usuário. Na implementação da ferramenta, foram utilizados padrões de projetos, componentes e testes funcionais. A qualidade da arquitetura da ferramenta facilita a incorporação de possíveis extensões na linguagem de transformação. A utilização do padrão de projeto Command isola a parte do código do sistema que deve ser alterada, facilitando a manutenção. Para que um novo eixo seja tratado, basta que uma nova classe Command seja criada.

Como estudo de caso, utilizamos a plataforma de desenvolvimento Hércules e como caso de testes, foi utilizado o sistema de gestão acadêmica da UFRJ. A escolha do estudo de caso e do caso de testes se deve a possível utilização da ferramenta pela equipe do SIGA, visando uma melhor documentação e uma maior abstração no processo de desenvolvimento do sistema. Logo, o processo visa atender a uma demanda de um sistema existente.

Algumas fragilidades encontradas no processo e que poderão ser solucionadas com trabalhos futuros são: o diagrama da camada de Controle não é gerado automaticamente; o modelo de transformação é dependente da plataforma; a ferramenta é unidirecional; o modelo de transformação não é validado pela ferramenta; e a camada de Domínio não é contemplada pelo processo.

Os diagramas de vista e modelo de caso de uso, que participam do modelo específico para a plataforma Hércules, são gerados através de transformações. No entanto, com o modelo independente atual não é possível gerar o diagrama de controle. O modelo independente de plataforma precisa ser incrementado de forma que seja possível a geração completa do PSM do sistema.

Nenhum empecilho foi identificado para que outras plataformas possam se utilizar deste mesmo processo. A adaptação deve ocorrer sem problemas para qualquer plataforma gráfica que utilize a arquitetura MVC. Um exemplo de plataforma, para qual a ferramenta pode ser facilmente adaptada, é a plataforma gráfica SWING. Para que a ferramenta possa gerar o PSM para essa plataforma, deverão ocorrer apenas modificações na fase de modelagem, onde as classes deverão ter estereótipos com elementos visuais pertencentes a esta plataforma. No entanto, testes foram realizados somente para a plataforma Hércules. Um próximo trabalho poderia propor um diagrama de transformação em que elementos independente de plataforma fossem utilizados.

A ferramenta é unidirecional, uma proposta para trabalhos futuros é tornar a ferramenta bidirecional. Assim, não somente os artefatos seriam gerados a partir de diagramas como também os diagramas poderiam ser gerados a partir dos artefatos. Modificações realizadas nos artefatos após a geração também poderiam ser refletidas no modelo e o mesmo acontecendo com modificações feitas no modelo.

Um arquivo DTD foi utilizado para criar os atributos das classes no PSM. A utilização do DTD facilita o trabalho do usuário, pois os atributos requeridos são gerados no XML de saída e seus valores já estão preenchidos com o valor padrão encontrado na descrição do atributo dentro do DTD. Como uma complementação do trabalho, o DTD poderia ser utilizado na validação do modelo de transformação e do PSM, já que este possui a informação de qual elemento pode estar contido dentro de outro.

O projeto Hércules e conseqüentemente a ferramenta de geração não contemplam a implementação do domínio da aplicação. Uma proposta para trabalhos futuros seria a incorporação dessas funcionalidades. Assim o sistema todo poderia ser gerado através de diagramas UML.

Referências Bibliográficas

AMBLER, S. **Uma Visão Realística da Reutilização em Orientação a Objetos**. Janeiro, 1998

APACHE JAKARTA PROJECT. **Velocity**. Disponível em http://jakarta.apache.org/velocity. Acesso em: mai. 2004.

COMPUWARE CORPORATION. **OptimalJ 3.0 User's Guide. Disponível em http://www.compuware.com/products/optimalJ**. Acesso em mai. 2004.

COMPUWARE CORPORATION AND SUN MICROSYSTEMS. **MOF Query/Views/ Transformations, Revised Submission**. OMG Document: ad/03-08-07. 2003. Disponível em http://www.omg.org/cgi-bin/apps/doc?ad/03-08-07.pdf. Acesso em mar. 2004.

CZARNECKI, K; HELSEN, S. **Classification of Model Transformation Approaches.** In: OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003, California. Disponível em http://www.swen.uwaterloo.ca/~kczarnec/ECE750T7/czarnecki_helsen.pdf. Acesso em: mai. 2004.

DSTC, CBOP AND IBM. **MOF Query/Views/Transformations, Revised Submission**. OMG Document: ad/03-08-03. 2003. Disponível em http://www.omg.org/cgi-bin/apps/doc?ad/04-01-06.pdf. Acesso em mai. 2004.

FRANKEL, D. S. **Model Driven Architecture: Applying MDA to Enterprise Computing**. Indianapolis. OMG Press, 2003. -

FOWLER, M.; SCOTT, K. **UML Essencial: Um breve guia para a linguagem-padrão de modelagem de objetos**. Porto Alegre. Bookman, 2 ed., 2000.

GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, Reading, MA, 1995.

GARDNER T.; GRIFFIN, C.; KOEHLER, J.; HAUSER, R. **A review of OMG MOF 2.0 Query / Views /Transformations Submissions and Recommendations towards the final Standard**. Disponível em http://www.zurich.ibm.com/pdf/ebizz/gardner-etal.pdf. Acesso em: mai. 2004.

HAROLD, E. R. **XML BIBLE.** Foster City, CA. IDG Books Worldwide, Inc. 1999.

INTERACTIVE OBJECTS AND PROJECT TECHNOLOGY. **MOF Query/Views/Transformations, Revised Submission**. OMG Document: ad/03-08-11. 2003. Disponível em http://www.omg.org/docs/ad/03-08-11.pdf. Acesso em mai. 2004.

JUDSON, S. R., FRANCE, R. B., CARVER, D. L. **Specifying Model Transformations at the Metamodel Level.** Disponível em http://www.metamodel.com/wisme-2003/19.pdf. Acesso em: jan. 2006.

KENNEDY CARTER. **Supporting Model Driven Architecture with eXecutable UML**. 2002. Relatório Técnico – Ref: CTN 80 v2.2. Disponível em http://www.kc.com. Acesso em: mai. 2004.

NCE-UFRJ. **Hércules – Apresentação.** Disponível em http://labase.nce.ufrj.br/hercules. Acesso em: mar, 2006.

NETBEANS.ORG. **Metadata Repository Home.** Disponível em http://mdr.netbeans.org. Acesso em: jan. 2006.

OLIVEIRA, G. B. M., CUNHA, M. F**. Experiência prática na construção de aplicações utilizando Model Driven Architecture.** 2005. Projeto Final de Curso (Bacharelado em Informática) – DCC, IM, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

OMG. **MDA Guide Version 1.0.1**. Disponível em http://www.omg.org/docs/omg/03-06-01.pdf. Acesso em: jun. 2003.

OMG. **OMG's MetaObject Facility Home Page.** Disponível em http://www.omg.org/mof. Acesso em mar.2006.

OMG. **OMG MOF 2.0 Query, Views, Transformations request for proposals**. Disponível em http://www.omg.org/techprocess/meetings/schedule/MOF_2.0_Query_View_Transf._RFP.html. Acesso em: mai. 2004.

OMG. **XML Metadata Interchange**, Disponível em http://www.omg.org/technology/documents /formal/xmi.htm. Acesso em: mai. 2004.

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. 116 p. Dissertação (Mestrado em Ciência da Computação) – Mestrado em Informática IM/NCE, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

QVT-PARTNERS, **MOF Query/Views/Transformations, Revised Submission**. OMG Document: ad/2003-08-08.2003. Disponível em http://tratt.net/laurie/research/publications/papers/qvtpartners1.1.pdf. Acesso em mai. 2004.

SUN MICROSYSTEMS. **NetBeans Metadata Repository**. Disponível em http://mdr.netbeans.org.Acesso em: jun. 2003.

THALES, ALCATEL, SOFTEAM, TNI-VALIOSYS, CODAGEN CORPORATION, *et al*. **MOF Query/Views/Transformations, Revised Submission**. OMG Document: ad/03-08-05. 2003. Disponível em http://www.omg.org/cgi-bin/apps/doc?ad/03-08-05.pdf. Acesso em mai. 2004.

XDOCLET. **XDoclet.** Disponível em http://sourceforge.net/projects/xdoclet. Acesso em: mai. 2004.

WARMER, J.; KLEPPE, A.; BAST, W. **OMG Standards and Additional Technologies**. In: ____, MDA Explained: The Model Driven Architecture: Practice and Promise. Boston. Addison Wesley, 2003. Cap. 11, p. 139 – 148.

WARMER, J.; KLEPPE, A.; BAST, W. **MDA Explained: The Model Driven Architecture: Practice and Promise**. Boston. Addison Wesley, 2003a.

WILLINK, E. D. **UMLX: A graphical transformation language for MDA**. 2003. Disponível em http://www.softmetaware.com/oopsla2003/willink.pdf. Acesso em: abr. 2004

W3C. **XSL Transformations (XSLT)**. Disponível em http://www.w3.org/TR/xslt. Acesso em: mai. 2004.

W3C. **XQuery 1.0: An XML Query Language.** Disponível em http://www.w3.org/TR/xquery. Acesso em: mai. 2004a.

WILKIE, I., KING, A., CLARKE, M., WEAVER, C., RASTRICK, C. **UML ASL Reference Guide.** Disponível em http://www.kc.com. Acesso em: mai. 2004.

WUTKA CONSULTING - **DTDParser** - Disponível em: http://www.wutka.com/dtdparser.html. Acessado em: 28, fev. 2003.

Anexo I - DTD da especificação abstrata da plataforma Hércules

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT mvc (view?,model?,control?)>

<!--
   Este DTD corresponde aos elementos suportados pelo Hércules.
   Os elementos/atributos que não estiverem relacionados aqui, não serão renderizados
   pelo sistema. Os atributos relacionados como #REQUIRED devem ser preenchidos
   obrigatóriamente.
 -->

<!ELEMENT view (tabbox|box|hbox|vbox)>

<!ELEMENT tabbox (tabs, tabpanels, caption?)>
<!ATTLIST tabbox
   id                CDATA #REQUIRED>

<!ELEMENT tabs (tab+)>

<!ELEMENT tab EMPTY>
<!ATTLIST tab
   label             CDATA #REQUIRED
   selected          (true|false) "false">

<!ELEMENT tabpanels (tabpanel+)>

<!ELEMENT tabpanel (box|grid|hbox|vbox|wizard)>
<!ATTLIST tabpanel
   id                CDATA #REQUIRED>

<!ELEMENT wizard (wizardpage)+>
<!ATTLIST wizard
   id                CDATA #REQUIRED>

<!ELEMENT wizardpage (box|grid|hbox|vbox|wizard)>
<!ATTLIST wizardpage
   id                CDATA #REQUIRED
```

```
    onpageadvanced          CDATA #IMPLIED
    onpagerewound           CDATA #IMPLIED
    nextButtonVisible       (true|false) "true"
    previousButtonVisible   (true|false) "true">

<!ELEMENT box (button|caption|checkbox|grid|image|label|menulist|mvcWindow|textbox|tree|box|hbox
|vbox|grid|wizard)*>
<!ATTLIST box
    id                  CDATA #REQUIRED
    width               CDATA #IMPLIED
    value               CDATA #IMPLIED
    align               (start|end|center) "start"
    class               (bevel|line|none) "none">

<!ELEMENT hbox(button|caption|checkbox|grid|image|label|menulist|mvcWindow|textbox|tree|box|hbox|
vbox|grid|wizard)*>
<!ATTLIST hbox
    id                  CDATA #REQUIRED
    width               CDATA #IMPLIED
    align               (start|end|center) "start"
    class               (bevel|line|none) "none">

<!ELEMENT vbox(button|caption|checkbox|grid|image|label|menulist|mvcWindow|textbox|tree|box|hbox|
vbox|grid|wizard)*>
<!ATTLIST vbox
    id                  CDATA #REQUIRED
    width               CDATA #IMPLIED
    align               (start|end|center) "start"
    class               (bevel|line|none) "none">

<!ELEMENT grid ((columns)?,rows,(commands)?,(caption)?)>
<!ATTLIST grid
    id                  CDATA #REQUIRED
    datasources         CDATA #IMPLIED
    ref                 CDATA #IMPLIED
    observes            CDATA #IMPLIED
    value               CDATA #IMPLIED
    width               CDATA #IMPLIED
    class               (plainTable|checkTable|listTable|selectTable|selectListTable|gridLayout) "gridLayout">
```

```
<!ELEMENT columns (column)+>

<!ELEMENT column EMPTY>
<!ATTLIST column
   value              CDATA #IMPLIED
   flex               CDATA #IMPLIED
   align              (start|center|end) "start"
   valign             (start|center|end) "start"
   class              (normal|shadow|light) "normal">

<!ELEMENT rows ((row)+|template)>
<!ELEMENT row (button|checkbox|grid|image|label|menulist|mvcWindow|textbox|tree|box|hbox|vbox)
*>

<!ELEMENT template (row|menuitem)>

<!ELEMENT caption EMPTY>
<!ATTLIST caption
   label              CDATA #REQUIRED>

<!ELEMENT commands (command)+>
<!ELEMENT command EMPTY>
<!ATTLIST command
   label              CDATA #IMPLIED
   class                    (EXTERNAL|CHECK_ALL|UNCHECK_ALL|REMOVE_CHECKED_ITEMS|
ADD|REMOVE) #REQUIRED
   oncommand      CDATA #IMPLIED>

<!ELEMENT button EMPTY>
<!ATTLIST button
   id                 CDATA #REQUIRED
   command        CDATA #IMPLIED
   label              CDATA #IMPLIED
   type               (default|eaction|action|lookup) "default"
   value              CDATA #IMPLIED>

<!ELEMENT checkbox EMPTY>
<!ATTLIST checkbox
   id                   CDATA #REQUIRED
```

```
    datasources         CDATA #IMPLIED
    ref                 CDATA #IMPLIED
    observes            CDATA #IMPLIED
    label               CDATA #IMPLIED
    checked              (true|false) "false">

<!ELEMENT image EMPTY>
<!ATTLIST image
    id                  CDATA #REQUIRED
    src                 CDATA #REQUIRED
    datasources         CDATA #IMPLIED
    ref                 CDATA #IMPLIED
    observes            CDATA #IMPLIED
    onclick             CDATA #IMPLIED>

<!ELEMENT label EMPTY>
<!ATTLIST label
    id                  CDATA #REQUIRED
    datasources         CDATA #IMPLIED
    ref                 CDATA #IMPLIED
    observes            CDATA #IMPLIED
    class               (normal|caption|title) "normal"
    value               CDATA #IMPLIED
    onclick             CDATA #IMPLIED>

<!ELEMENT menulist (menupopup)>
<!ATTLIST menulist
    id                  CDATA #REQUIRED
    datasources         CDATA #IMPLIED
    ref                 CDATA #IMPLIED
    observes            CDATA #IMPLIED>

<!ELEMENT menupopup ((menuitem)+|template)>
<!ATTLIST menupopup
    datasources         CDATA #IMPLIED
    ref                 CDATA #IMPLIED>

<!ELEMENT menuitem EMPTY>
<!ATTLIST menuitem
```

```
   value              CDATA #REQUIRED
   label              CDATA #IMPLIED>

<!ELEMENT textbox EMPTY>
<!ATTLIST textbox
   id                 CDATA #REQUIRED
   maxlength          CDATA #IMPLIED
   datasources        CDATA #IMPLIED
   ref                CDATA #IMPLIED
   rows               CDATA #IMPLIED
   cols               CDATA #IMPLIED
   observes           CDATA #IMPLIED
   type               (text|password|pickdate) "text"
   multiline           (true|false) "false">

<!ELEMENT tree (treecols,treechildren)>
<!ATTLIST tree
   id                CDATA #REQUIRED
   value              (true|false) "false">

<!ELEMENT treecols (treecol)>
<!ELEMENT treecol EMPTY>
<!ATTLIST treecol
   label              CDATA #REQUIRED>

<!ELEMENT treechildren (treeitem)>
<!ATTLIST treechildren
   datasources        CDATA #REQUIRED
   ref                CDATA #REQUIRED>

<!ELEMENT treeitem (treerow)>
<!ELEMENT treerow (treecell)>
<!ELEMENT treecell EMPTY>
<!ATTLIST treecell
   datasources        CDATA #REQUIRED
   ref                CDATA #REQUIRED
   cols               CDATA #IMPLIED
   maxlength         CDATA #IMPLIED>
```

```
<!ELEMENT mvcWindow (menulist)>
<!ATTLIST mvcWindow
   id                  CDATA #REQUIRED
   value               CDATA #REQUIRED
   datasources         CDATA #IMPLIED
   ref                 CDATA #IMPLIED>

<!ELEMENT model (EDIT|list|select_|message)*>

<!ELEMENT EDIT (JavaBean)*>

<!ELEMENT JavaBean (properties)>

<!ATTLIST JavaBean
  id              CDATA #IMPLIED
  class           CDATA #REQUIRED
  table           CDATA #IMPLIED
  label           CDATA #IMPLIED
  access          CDATA #IMPLIED
  delete          CDATA #IMPLIED>

<!ELEMENT properties (property)*>

<!ELEMENT property (collection|fk-table)*>
<!ATTLIST property
  name          CDATA #REQUIRED
  type          CDATA #REQUIRED
  value         CDATA #IMPLIED>

<!ELEMENT collection (JavaBean)*>

<!ELEMENT fk-table (#PCDATA)>
<!ATTLIST fk-table
  package         CDATA #REQUIRED>

<!ELEMENT list (JavaBean)*>

<!ELEMENT select_ (JavaBean)*>
```

```
<!ELEMENT message (JavaBean)*>

<!ELEMENT control (state)*>

<!ELEMENT state (tab|action)*>
<!ATTLIST state
  id              CDATA #REQUIRED
  actualState     CDATA #REQUIRED
  active          CDATA #REQUIRED>

<!ELEMENT tab EMPTY>
<!ATTLIST tab
  id              CDATA #REQUIRED
  active          CDATA #IMPLIED>
```

Anexo II – PSM gerado pela ferramenta Megara para o caso de uso de Previsão de Turma

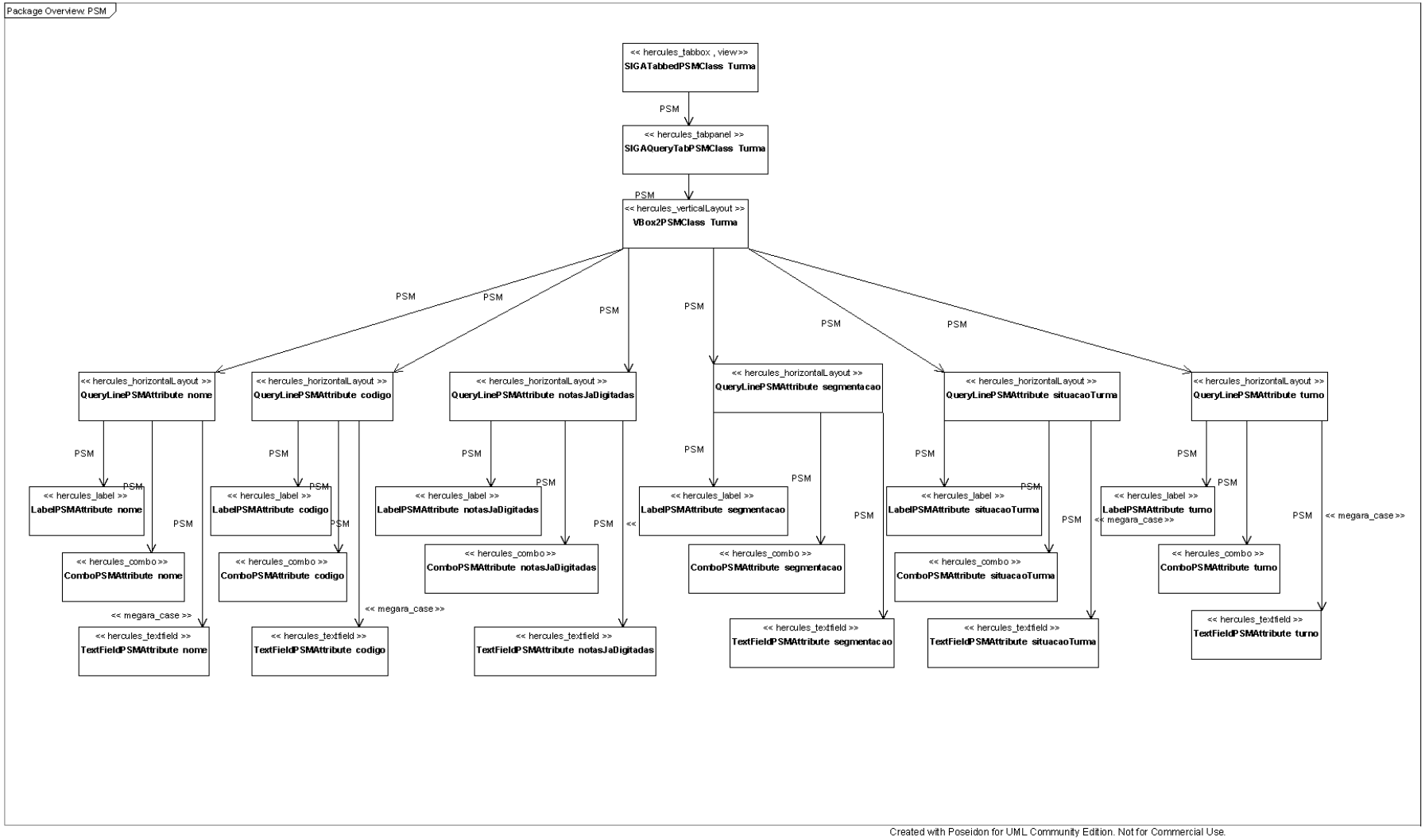


Figura 38 - Parte do PSM gerado pela ferramenta Megara para o caso de uso Previsão de Turma - Tela de consulta

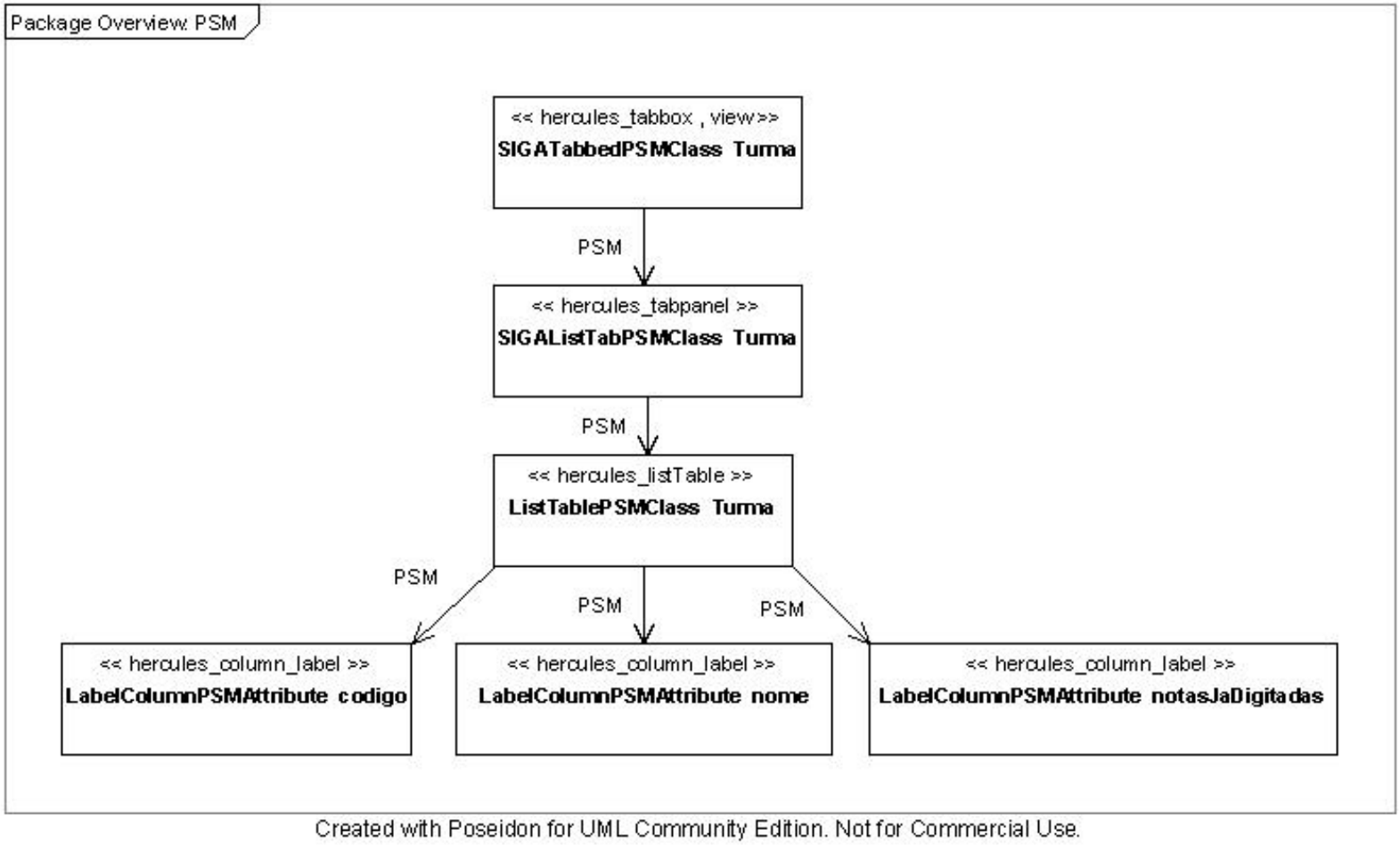


Figura 39 - Parte do PSM gerado pela ferramenta Megara para o caso de uso da Previsão de Turma - Tela de Lista (Resultado da Consulta)

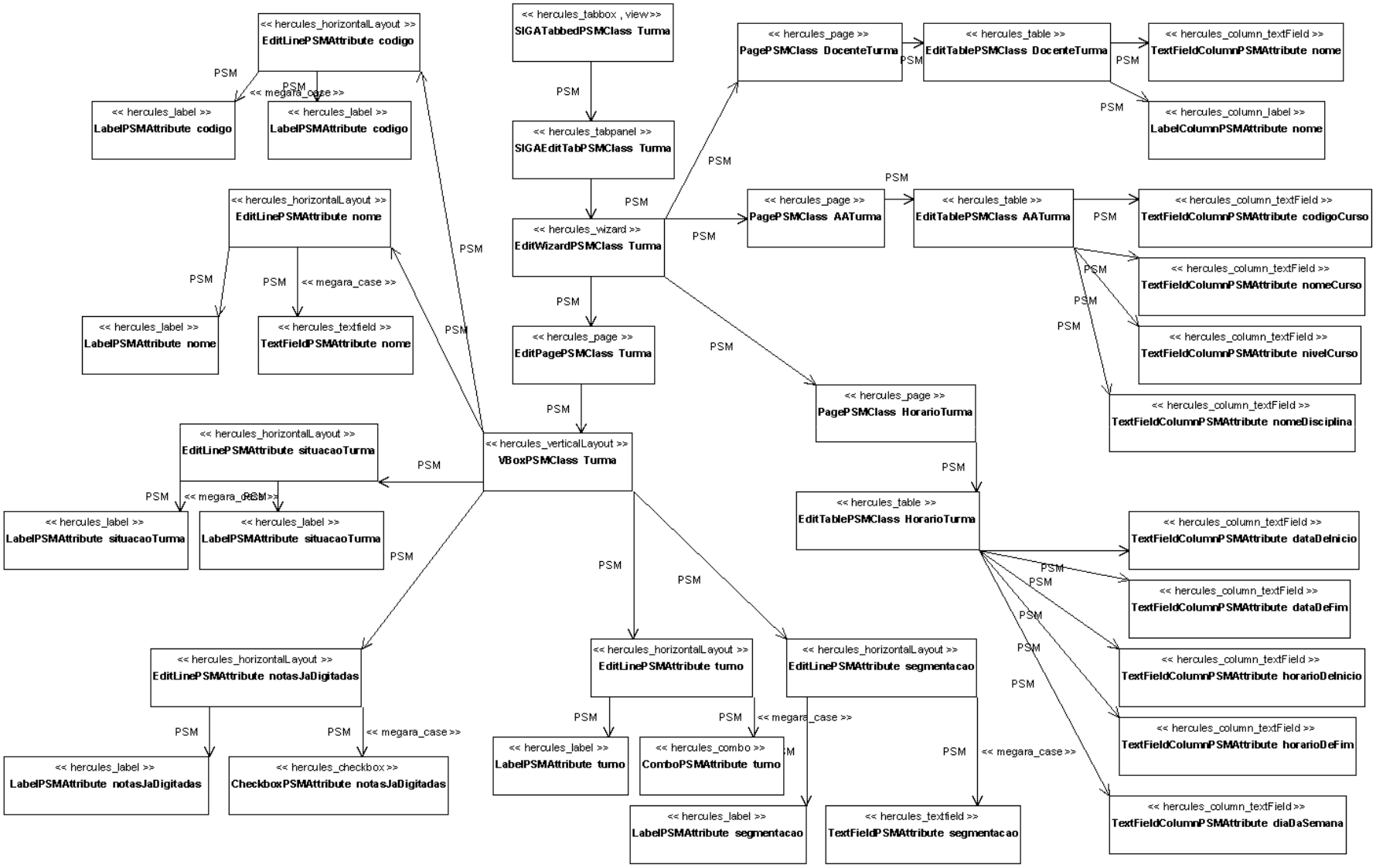


Figura 40 - Parte do PSM gerado pela ferramenta Megara para o caso de uso da Previsão de Turma - Parte da tela de Edição