Lucas Villela Neder Issa

Desenvolvimento de Interface com Usuário Dirigido por Modelos e Geração Automática de Código

Belo Horizonte
Novembro de 2006

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Desenvolvimento de Interface com Usuário Dirigido por Modelos e Geração Automática de Código

Dissertação apresentada ao Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Lucas Villela Neder Issa

Belo Horizonte
Novembro de 2006

Universidade Federal de Minas Gerais

FOLHA DE APROVAÇÃO

Desenvolvimento de Interface com Usuário Dirigido por Modelos e Geração Automática de Código

Lucas Villela Neder Issa

Dissertação defendida e aprovada pela banca examinadora constituída por:

Ph. D. Clarindo Isaías Pereira da Silva e Pádua – Orientador
Universidade Federal de Minas Gerais

Ph. D. Rodolfo Sérgio Ferreira Resende
Universidade Federal de Minas Gerais

Ph. D. André Luiz Zambalde
Universidade Federal de Lavras / MG

Belo Horizonte, 28 de novembro de 2006

Para minha noiva, Fernanda.

Agradecimentos

Agradecimentos especiais a meus pais que não mediram esforços para que eu chegasse aqui. A meus irmãos, pelo apoio. A Raquel e a minha mãe, pela ajuda nas revisões do texto.

A minha noiva, Fernanda, pelo amor, incentivo, compreensão, paciência e apoio durante todo esse tempo.

A minha família e amigos, por compreenderem minhas ausências.

Ao Professor Clarindo, pela confiança, disponibilidade, paciência e pela excelente orientação.

Ao Synergia, pelas oportunidades de aprendizado que propiciaram a experiência e a motivação necessárias para a realização deste trabalho. A todos os companheiros e amigos do Synergia que sempre proporcionaram um excelente ambiente de trabalho. Aos professores Robson, Wilson e Clarindo, por acreditarem e apoiarem meu trabalho dentro do Synergia. Ao Professor Wilson, pela ajuda no estudo e no entendimento da Engenharia de *Software*.

Ao Professor Rodolfo, pelos ensinamentos e pelas orientações.

Ao Stênio, pela grande ajuda, apoio e motivação durante o mestrado. Ao Gorgulho, pela confiança e pela compreensão nos momentos difíceis, em que foi necessária dedicação além do previsto no mestrado. Ao Benício, pela força e ajuda ao entrar no mestrado. Ao Pedro, pela ajuda e pelos ensinamentos. A todos os amigos que, de alguma forma, contribuíram para conclusão deste trabalho, entre eles Daniel Câmara, Piti, Habib, Wagner, André, Corélio, Bruno, Alysson, Wolber, Samyra, Gislaine, Rafael Barra, Diego, Samuel, Vilaça, Tiquinho e aos desmaiados.

Enfim, agradeço ao programa de iniciativa acadêmica da IBM por oferecer ferramentas e ambiente de desenvolvimento que foram de grande valia para realização deste trabalho.

Sumário

Lista de figuras

Lista de tabelas

Resumo

Atualmente, modelos UML são muito usados e importantes na Engenharia de *Software*, no entanto, é normal que depois que o sistema começa a ser codificado tais modelos percam sua importância e fiquem desatualizados à medida que o desenvolvimento avança. Neste trabalho, essa questão é abordado por meio da aplicação de técnicas de MDA (*Model Driven Architecture*) que visam a focar o desenvolvimento de software na criação de modelos usados na geração automática de código. Baseado no trabalho aqui apresentado, foi proposto um método para geração de interfaces (MDGI - Método de Desenvolvimento e Geração de Interfaces) que permite gerar modelos de padrões recorrentes em sistemas. Esses padrões são referentes à interface com o usuário e são gerados a partir do modelo de domínio da aplicação. O MDGI segue a filosofia do MDA e, portanto, é totalmente compatível com ele. Como resultado final deste trabalho, além do MDGI, foi formalizado um modelo para interface com o usuário independente de tecnologia e plataforma, denominado PIM UI, que pode ser usado na execução do MDGI. Um caso de uso foi implementado seguindo o método proposto e analisado o resultado comparando com desenvolvimento feito manualmente. Os resultados preliminares obtidos nessa análise mostram que é possível ter ganho de produtividade ao usar o MDGI, na situação analisada.. Nessa análise também é apontado que o MDGI permite nível maior de reutilização, pois ele, em conjunto com o MDA, encapsula conhecimento de transformação de um modelo em outro modelo mais detalhado.

Abstract

UML models are frequently used and are important in software engineering, however, it is normal that after the system codification starts these models tend to lose their importance and become outdated as the development advances. In this work, this problem is treated through the application of MDA techniques (Model Driven Architecture) which aim at software development through the creation of models that are used in the automatic generation of code. Based on the presented work, we propose a method for interface generation (MDGI - Method of Development and Generation of Interfaces) that allows generating models of recurrent patterns in systems. These patterns refer to the user interface and are generated from the domain model of the application. The MDGI follows the MDA philosophy; therefore it is totally compatible with it. As a final result of this work, in addition to the MDGI, a technology and platform independent model for user interaction called PIM UI was created, and can be used in the MDGI execution. A use case was implemented following the proposed method and the result was analyzed comparing with manual development. The results observed in this analysis show that by using MDGI it is possible to increase productivity, in the analised situation. In this analysis, it is also pointed out that the application of MDGI allows better reuse level because it encapsulates knowledge for the transformation from a model into another more detailed model.

Capítulo 1

Introdução

1.1 Contextualização e motivação

Um dos objetivos da Engenharia de *Software* é construir sistemas extensíveis e de qualidade. Além disso, a busca da redução de custos e do aumento da produtividade na construção do *software* é constante. No entanto, com o passar do tempo, a complexidade do *software* produzido tem aumentado em escala, dificultando ainda mais o alcance desses objetivos. Neste contexto, surgiu a linguagem de modelagem *Unified Modeling Language* (UML) [25] [46], com o objetivo de permitir a modelagem, com visualização gráfica, em mais alto nível de abstração dos sistemas ou até mesmo do problema que o sistema se propõe resolver ou automatizar.

Um dos marcos na história da evolução da Engenharia de *Software* foi o desenvolvimento da metodologia de orientação a objetos, utilizada na programação de computadores. Essa metodologia facilitou a reutilização de componentes já existentes, visando à melhoria da produtividade e introduziu um modo de programar em que os elementos de programação são associados a elementos e a conceitos do mundo real. Com a crescente complexidade de aplicações que o *hardware* permitia, os metodologistas da área de Engenharia de *Software* buscaram novas abordagens às questões de análise e projeto. Reconhecia-se a grande importância de se modelar os complexos problemas cuja solução se buscava em sistemas de *software*. Não era mais razoável, às vezes era até impossível, desenvolver sistemas sem um estudo e uma definição do problema por meio de técnicas de modelagem.

Assim, surgiram pesquisas visando ao desenvolvimento de metodologias para a modelagem necessária à análise e ao projeto de *software*. A abordagem de orientação a objetos, além de ser usada na programação do código-fonte, passou a ser utilizada também para a modelagem na atividade de análise. Três dos mais conhecidos metodologistas, Grady Booch, James Rumbaugh e Ivar Jacobson, resolveram unificar as metodologias que propunham. Essa foi a origem do Processo Unificado para o desenvolvimento de *software* [14]. Para a criação desse processo, era necessário também criar um padrão para a notação utilizada na modelagem. Assim nasceu a UML. A responsabilidade pelo trabalho de padronização associado à UML foi entregue ao consórcio OMG (*Object Management Group*) [40], que reúne, virtualmente, todas as grandes empresas da indústria mundial de computação e centenas de companhias pequenas. O OMG é uma

sociedade aberta, sem fins lucrativos, que produz e mantém especificações que padronizam as necessidades da indústria de *software*. O Processo Unificado da Rational [12] é um exemplo de processo comercial e maduro que é baseado nesses padrões.

Desde então, a UML tem sido muito utilizada no desenvolvimento de sistemas. Entretanto, ela ainda pode ser melhor aproveitada em técnicas mais sofisticadas. A seguir, esse trabalho irá destacar algumas formas de utilização. Para cada uma, identificaremos alguns problemas que dificultam a utilização de modelos durante o desenvolvimento. Por isso, esse trabalho pretende estudar, aplicar e analisar metodologias que ajudam a tornar a utilização da modelagem de *software* mais efetiva.

A utilização da UML tem se dado de várias formas e com objetivos diversos. Entre estes, podemos destacar três utilizações:

1. Documentar os requisitos do usuário, ou seja, modelar o problema a ser automatizado.
2. Gerar a estrutura do código a ser codificado, ou seja, gerar o código-fonte automaticamente das classes com seus relacionamentos e com seus métodos vazios, sem qualquer lógica definida.
3. Documentar o código produzido na implementação do sistema.

Para cada tipo de aplicação apresentada acima, pode-se identificar problemas ou dificuldades que ocorrem durante o processo de desenvolvimento de *software*. Na primeira aplicação, à medida que o sistema é desenvolvido, os requisitos vão ficando mais claros tanto para o cliente, ou para o usuário final, quanto para os desenvolvedores. Com isso, é normal que ocorram alterações de requisitos, alterações nas interações entre os objetos ou ainda alterações nos próprios objetos que foram levantados no início do desenvolvimento. Essas alterações nem sempre são documentadas, pois o custo desse processo é alto, uma vez que, depois de iniciada a implementação, a maior parte dos esforços é concentrada na produção e na documentação do código-fonte.

No segundo tipo de aplicação, a geração da estrutura do código agiliza bastante a produção do código-fonte. Entretanto, o modelo usado para gerar a estrutura do código-fonte se torna obsoleto rapidamente, principalmente mediante as várias técnicas de refatoração [9] que são utilizadas atualmente. A utilização dessas técnicas é facilitada por várias ferramentas existentes no mercado que as aplica. Essas podem ser encontradas tanto sob licenças sem custo para o desenvolvedor quanto sob licenças pagas.

No terceiro tipo de aplicação, a documentação em UML do código produzido é facilitada por ferramentas que fazem engenharia reversa do código-fonte [48], gerando automaticamente as classes e seus relacionamentos na notação UML. No entanto, ainda é necessário um esforço considerável para organizar os diagramas e para corrigir alguns detalhes que a engenharia reversa não é capaz de gerar automaticamente. Devido a esse esforço, existe uma tendência de a

documentação UML ficar desatualizada sempre que o código-fonte é alterado. Durante o desenvolvimento do sistema é praticamente inviável manter essa documentação consistente pelos mesmos motivos descritos no item anterior. Neste caso, fazê-la no final do desenvolvimento pode ser a solução, apesar de que muitos detalhes podem ser deixados para trás, já que não foram realizados no ato do desenvolvimento. Além disso, toda alteração de correção ou manutenção do sistema poderá deixar a documentação desatualizada e/ou inconsistente.

Complicando esse cenário, com o passar do tempo são criadas novas tecnologias. Migrar sistemas existentes para as novas tecnologias tem sido um trabalho caro e árduo para as grandes empresas, principalmente quando é necessário fazer sistemas com tecnologias diferentes "conversarem" entre si. Nesse contexto é que a metodologia conhecida como *Model Driven Architecture* (MDA) [44] pode ajudar a atingir os objetivos citados na utilização da UML.

O MDA é uma arquitetura que, em primeiro lugar, tem seu foco na funcionalidade e no comportamento da aplicação ou sistema, sem distorções causadas por idiossincrasias da tecnologia ou por tecnologias que seriam usadas na implementação. O MDA desacopla detalhes de implementação das funções de negócio. Nele, a modelagem do negócio é totalmente separada da modelagem da solução adotada. Além disso, ele provê uma arquitetura que permite reutilizar conceitos de negócio e conceitos de solução independentemente. Isso tanto no nível de modelagem, quanto no nível de código-fonte.

No MDA, o foco é centrado no modelo e não no código-fonte. O modelo passa a gerar resultados concretos, ou seja, parte da aplicação é gerada automaticamente a partir dele. Assim, é possível focar, de forma efetiva, no modelo. Nas três formas de utilização da UML citadas, o que gera resultado concreto é a produção do código-fonte. Portanto, os modelos que representam esse código sempre necessitaram de esforço para acompanhar as alterações que são realizadas no código-fonte. É por causa disso que eles acabam sempre ficando desatualizados à medida que o aplicativo evolui. No MDA, alterações podem ser feitas diretamente no modelo que serão transformadas em outros modelos que descrevem a implementação e em código-fonte. Utilizando essa abordagem, a documentação dos requisitos e da implementação estará naturalmente atualizada. Além disso, no MDA é possível automatizar padrões de implementação. Dessa forma, o código-fonte gerado não será apenas estrutural, uma "casca". As lógicas que são comuns em várias aplicações ou em várias partes de uma mesma aplicação poderão ser geradas ou re-geradas automaticamente, sempre que houver necessidade.

A UML vem sendo usada há algum tempo e ainda existem pontos em que sua utilização pode ser melhorada. Por isso, pretende-se estudar, aplicar e analisar metodologias e padrões de modelagem de *software*, como o MDA, que permitam o desenvolvimento de sistemas com alto nível de reuso, documentação, produtividade e qualidade, que abranja desde a concepção até a implementação de sistemas.

1.2 Objetivos

O objetivo deste trabalho é produzir um método de transformação e uma ferramenta de apoio ao desenvolvimento de *software* baseado no paradigma MDA. Tal ferramenta deve permitir a modelagem e a geração de parte de aplicações, incluindo as interfaces com o usuário. É por meio das interfaces que o usuário do sistema tem contato com os requisitos implementados. Para atender aos requisitos de um sistema é necessário modelar a interface do *software* com o usuário. Embora não esteja tradicionalmente no escopo do MDA detalhar como tratar as questões de interface, este é um aspecto que será considerado visando permitir o uso efetivo do conceito de MDA. Portanto, o foco do trabalho será em como tratar a modelagem de interface dentro do conceito MDA.

Para fazer uma validação informal do resultado deste trabalho, é feito uma comparação entre duas abordagens de desenvolvimento da mesma aplicação. A primeira utiliza o método proposto e a ferramenta desenvolvida neste trabalho e a segunda não usa o método e nem ferramenta de automatização do desenvolvimento. A primeira abordagem é desenvolvida no decorrer deste trabalho, a segunda usa os dados de uma aplicação já desenvolvida, empregando o Praxis [20], que é um processo de desenvolvimento de *software* que não utiliza o conceito MDA. O Praxis foi desenhado voltado para uso na educação, mas é usado também em um modelo industrial pelo Synergia[1], em uma versão personalizada, e por outras empresas brasileiras de desenvolvimento de programas.

Um objetivo secundário deste trabalho é fornecer base metodológica com comprovação prática para permitir que o Synergia e empresas interessadas no assunto façam melhor aproveitamento do desenvolvimento dirigido por modelos. É importante ressaltar que o Synergia proporcionou conhecimento empírico que foi de extrema valia para a realização deste estudo.

Como o Praxis não prevê automatizações conforme o MDA, espera-se que a utilização deste possa trazer benefícios, como redução de erros, melhor documentação do sistema, padronização, melhora na qualidade e aumento de produtividade.

[1] O Synergia é o laboratório de Engenharia de *Software* do Departamento de Ciência da Computação da Universidade Federal de Minas Gerais.

1.3 Limites do trabalho

O MDA é um tema complexo, sendo que o desenvolvimento de modelos e ferramentas nesse contexto aborda vários aspectos que, por si só, podem gerar grandes discussões, sendo merecedor de trabalhos específicos. Portanto, alguns aspectos importantes não foram abordados neste trabalho. A seguir, são citados alguns desses aspectos considerados importantes, mas que não ficaram dentro do escopo do trabalho.

Para confeccionar os modelos definidos neste trabalho, as atividades de requisitos e análise já devem ter sido executadas. A confecção dos modelos é feita a partir de resultados gerados por essas atividades. Este trabalho não aborda como elas devem ser executadas e nem quais resultados devem gerar.

Neste trabalho, é tratada apenas a geração de modelo e código-fonte a partir de modelo de alto nível construído por meio de artefatos gerados pela atividade de análise. O caminho contrário, que seria gerar os modelos de alto nível a partir do código-fonte e do modelo de baixo nível, não é abordado.

O gerenciamento de alterações no código-fonte e no modelo gerado é uma questão importante para o uso efetivo do MDA, entretanto esse assunto não faz parte de nosso escopo. Este gerenciamento define o que pode ser alterado nos artefatos gerados e garante que tais alterações sejam mantidas ao regerar os artefatos.

Este trabalho preocupa-se em automatizar padrões que são utilizáveis em várias aplicações, independentemente de suas regras de negócio. Tem-se como pressuposto que as regras de negócio específicas da aplicação devem ser codificadas diretamente no código-fonte gerado, preferencialmente em pontos de extensões apropriados. Por isso, não nos preocupamos em modelar tais regras de negócio. Elas podem e devem ser descritas textualmente nos elementos de modelagem, independentemente de plataforma, e quando necessário referenciar o modelo de análise. Neste trabalho, o foco foi automatizar tarefas de codificação repetitivas e que geralmente estão presentes em vários tipos de aplicações.

1.4 Estrutura do trabalho

O trabalho está estruturado da seguinte forma:

No Capítulo 2, é apresentado o referencial teórico relacionado ao tema. Apresenta-se a bibliografia básica, que explica o MDA e outros conceitos que são pré-requisitos para o entendimento e a aplicação do MDA. Em seguida, na bibliografia relacionada, são discutidas, em ordem cronológica, as publicações sobre interface com o usuário.

O Capítulo 3 apresenta a metodologia usada, explicita como o trabalho foi desenvolvido e os passos seguidos no decorrer da pesquisa.

O Capítulo 4 apresenta os resultados e a discussão. São explicados os modelos definidos, em seguida o método proposto e, depois, a discussão dos resultados alcançados com a aplicação da ferramenta desenvolvida.

Por último, no Capítulo 5, são apresentadas as conclusões.

Capítulo 2

Referencial teórico

2.1 Bibliografia básica

Esta seção descreve metodologias e padrões relacionados ao uso de modelos UML de forma efetiva. Aqui são descritos conceitos básicos e importantes para o entendimento e o desenvolvimento do trabalho.

2.1.1 *Model-Driven Software Development* - MDSD

MDSD é um paradigma de desenvolvimento de *software* projetado para projetos com equipes distribuídas envolvendo mais de 20 pessoas. Ele tem raízes na engenharia de linha de produto de *software* (SPL – *Software Product Line*) [36] [37], que é um conjunto de sistemas que compartilham características comuns para satisfazer uma necessidade específica de um segmento particular de mercado e que são desenvolvidos a partir de um núcleo comum fixo. O MDSD combina aspectos que compreendem desde abordagens populares até outras que podem escalar para desenvolvimento de *software* industrializado de larga escala. Ele tem o foco no desenvolvimento de aplicações a partir de modelos de domínio específicos. Análise de domínio, metamodelagem, geração dirigida por modelos, linguagem de *template*[2], projeto de *framework*[3] dirigido pelo domínio e princípios de desenvolvimento de *software* ágeis [35] são a espinha dorsal para esta abordagem.

---

[2] Estilo de linguagem que facilita a geração do código. Ela se baseia na idéia de criar modelos para o código a ser gerado, mas estes são modelos que podem ter trechos dinâmicos.

[3] No desenvolvimento do *software*, um *framework* ou arcabouço é uma estrutura de suporte definida em que um outro projeto do *software* pode ser organizado e desenvolvido. Tipicamente, um *framework* pode incluir programas de apoio, bibliotecas de código, linguagens de *script* e outros *softwares* para ajudar a desenvolver e juntar diferentes componentes de um projeto.

A relação entre MDSD e engenharia de linha de produto de *software* pode ser comparada com a relação entre desenvolvimento baseado em componentes e tecnologia de objetos: um é construído sobre a outro. O MDSD pode ser visto como uma extensão da engenharia de linha de produto de *software*. O que o separa da clássica engenharia de linha de produto de *software* é a ênfase no processo de desenvolvimento de *software* ágil. Uma das prioridades mais altas no MDSD é produzir *software* funcional que pode ser validado pelos usuários e pelas partes interessadas o mais cedo possível. Isso é consistente com as metodologias de desenvolvimento de *software* ágeis.

O MDSD é interessante, pois representa um paradigma para o desenvolvimento de *software* industrializado que provê métodos efetivos para lidar com as causas primárias de altos custos de desenvolvimento à medida que ele escala e com a complexidade de ambientes de desenvolvimento distribuídos. O MDSD contém uma coleção de técnicas pragmáticas que podem ser aplicadas nas ferramentas disponíveis atualmente. Essas técnicas cobrem boas práticas para processos e organização, modelagem de domínio, ferramentas de arquitetura e desenvolvimento de plataforma de aplicação. Elas são descritas resumidamente por Bettin [4]. O MDSD previne a degradação de arquiteturas em grandes sistemas e contém técnicas que permitem automatizar vários aspectos repetitivos no desenvolvimento de *software*.

Os seguintes valores[4] são aplicáveis para o MDSD:

- É preferível validar *software* em construção a requisitos de *software*.
- Trabalha-se com propriedades específicas do domínio, o que pode ser qualquer coisa desde modelos, componentes, *frameworks* e geradores até linguagens e técnicas.
- Existe um esforço em automatizar a construção de *software* a partir de modelos de domínio, então conscientemente a construção de fábrica de *software* é considerada um aspecto distinto da construção de aplicações.
- Sustenta-se o surgimento de correntes de fornecimento para desenvolvimento de software, que implica a especialização de domínios específicos e permite personalização em massa.

O MDSD considera o MDA como um "sabor específico" do MDSD com a diferença que o MDA tem foco maior na padronização da metodologia. Por outro lado, o MDSD é uma abordagem com foco maior em técnicas e boas práticas que podem ser usadas em vários

[4] Estes valores foram definidos na conferência OOPSLA 2003 (*Object-Oriented Programming, Systems, Languages, and Applications 2003*).
Fonte: Traduzido de http://www.mdsd.info/mdsd_cm/page.php?page=core&id=6, último acesso em junho de 2006.

contextos. Em função disso, vários aspectos relevantes no MDSD continuam tendo a mesma importância no contexto MDA. Portanto, a experiência adquirida e documentada pelo MDSD é muito importante para conduzir ao sucesso projetos desenvolvidos usando o MDA. Por exemplo, na literatura do MDSD encontramos diretrizes como a explicação para o seguinte mito: *Código gerado a partir de um modelo é ruim e não é adequado para leitura humana.* Explicação: *Produza código limpo e claro. Sempre que possível, faça a extração de trechos reutilizáveis ou redundantes, não crie um gerador que produza códigos redundantes. Em outras palavras, se seus projetistas não se preocupam em eliminar redundância nos códigos, os geradores produzidos por eles irão gerar códigos redundantes. Portanto, o código gerado é tão bom ou ruim quanto os códigos produzidos pelos seus melhores projetistas*. Pelo esclarecimento sobre esse mito, podemos tirar uma lição de boa prática para criar geradores de código e, como dito anteriormente, isso é totalmente pertinente no MDA e pode ser extrapolado para os geradores de modelo.

2.1.2 *Model-Driven Architecture* - MDA

O conceito MDA foi desenvolvido pelo OMG e um de seus objetivos é padronizar a modelagem dirigida por modelos para aplicações *enterprise*[5], assim como padronizar a interoperabilidade entre essas aplicações.

Uma aplicação completa MDA consiste basicamente de dois tipos de modelos: o primeiro é denominado PIM (*Plataform Independent Model)* [18] e o segundo, PSM (*Plataform Specific Model*) [18]. O PIM é um modelo que descreve o negócio (domínio) em si, sem detalhes sobre como será implementado e sem detalhes da tecnologia/plataforma que será usada. Já o PSM é um modelo que descreve detalhes de implementação, tecnologia e plataforma de um PIM. Ele é gerado a partir de uma transformação, baseada em metamodelo[6], feita sobre o PIM. Esse processo é ilustrado na Figura 1. Para um mesmo PIM, pode existir mais de um PSM. Para cada plataforma que se deseja suportar existe pelo menos um PSM que descreve detalhadamente a implementação do PIM na plataforma. Um PSM pode ser transformado em outro PSM mais detalhado ou em código-fonte e assim por diante. Miller [18] e Santos [26] detalham o processo de transformação de metamodelo.

[5] Neste contexto, o termo *enterprise* pode ser entendido como aplicação para empresa. Geralmente tais aplicações têm requisitos não-funcionais similares e em alguns casos não-triviais de serem satisfeitos, como, por exemplo: escalabilidade, segurança, distribuição e permissões de acesso.

[6] Metamodelo é um modelo que descreve a estrutura de outro modelo. Na seção 2.1.4 (p. 14), o metamodelo é explicado detalhadamente no contexto do MDA.

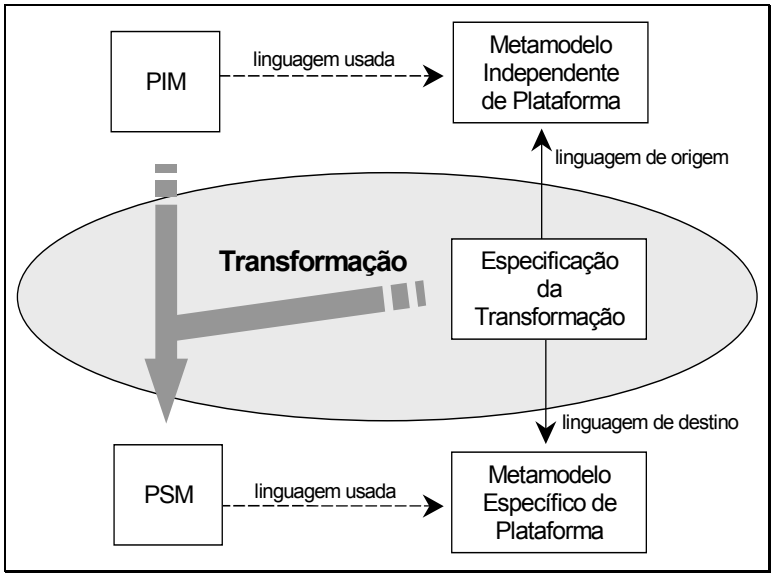


Figura 1 - Transformação de metamodelo

O MDA foi desenvolvido sobre padrões abertos, bem-estabelecidos, independentes de plataforma e também construídos pelo próprio OMG. Esses padrões são: UML, a notação de modelagem usada e suportada por todas as maiores companhias na indústria de *software*; *Meta Object Facility* (MOF) [42], a especificação para modelar linguagens; *XML Metadata Interchange* (XMI) [47], o padrão para armazenar e intercambiar modelos usando *eXtensible Markup Language* (XML) [5]; e *Common Warehouse Metamodel* (CWM) [39], a especificação que descreve o intercâmbio de metadados entre diferentes repositórios e armazéns de dados de uma corporação.

O desenvolvimento baseado em MDA abrange o ciclo de vida completo de desenho, implantação, integração e gerenciamento de aplicações e dados. Além disso, a MDA separa a lógica fundamental por trás de uma especificação das particularidades específicas do *middleware*[7] que a implementa. Isso possibilita o desenvolvimento rápido e a definição de especificações que usam novas tecnologias de implantação que são baseados em modelos de negócio testados e comprovados.

A arquitetura do MDA visa garantir:

[7] *Middleware* é um termo usado para se referir a servidores que oferecem infra-estrutura para alguns requisitos não-funcionais comuns a aplicações *enterprise*.

- **Portabilidade**, aumento de reutilização de aplicações e redução no custo e na complexidade do desenvolvimento e do gerenciamento de aplicações atuais e futuras.
- **Interoperabilidade entre plataformas**, usando rigorosos métodos para garantir que padrões baseados em múltiplas tecnologias implementem todas as funções de negócio de forma idêntica.
- **Independência de plataforma**, reduzindo tempo, custo e complexidade associados ao redirecionamento de aplicações para diferentes plataformas.
- **Especificidade no domínio**, por meio de modelos específicos de domínio que habilitam a implementação rápida de novas aplicações específicas para a indústria sobre diversas plataformas.
- **Produtividade**, permitindo que desenvolvedores, desenhistas e administradores de sistemas usem linguagens e conceitos em que eles se sintam confortáveis.

Os benefícios almejados com o uso de MDA são:

- *Custo reduzido ao longo do ciclo de vida da aplicação* - O MDA melhora a consistência do código e a manutenibilidade. A maior parte das organizações tem problemas em manter consistência entre projetos da arquitetura e os códigos das aplicações. Alguns desenvolvedores usam padrões de desenho enquanto outros, não. Com o MDA, o código gerado será consistente, o que permite que todos os desenvolvedores usem os mesmos padrões de desenho. Isso é vantagem significativa na perspectiva de manutenção, pois os desenvolvedores entendem mais facilmente os códigos uns dos outros, uma vez que é usado o mesmo paradigma de desenho e linguagem.
- *Tempo de desenvolvimento reduzido para novas aplicações* - Grande parte do modelo específico de tecnologia é gerada pelas transformações, assim como o código-fonte que esses modelos representam, incluindo a geração de algumas partes de lógica.
- *Melhoria na qualidade da aplicação* - Os artefatos gerados serão padronizados, simplesmente pelo fato de serem gerados automaticamente. Isso traz dois benefícios. Primeiro, com os modelos e código-fonte interno padronizados, eles se tornam mais fáceis de serem entendidos para quem já conhece a padronização. Segundo, a própria aplicação tende a ter comportamento de interação com o usuário padronizado, o que facilita também sua utilização. Esses dois tipos de padronização podem ser considerados atributos de qualidade que acabam sendo alcançados naturalmente com a utilização do MDA.
- *Aumento do retorno sobre investimentos feitos em tecnologia* - O conhecimento sobre a utilização de determinadas tecnologias pode ser implementado nas transformações. Conseqüentemente, essas tecnologias podem beneficiar pessoas que não as dominam completamente e quem as domina não despenderá tempo implementando trechos repetitivos de código.

- *Rápida inclusão de benefícios de tecnologias emergentes nos sistemas existentes* - A inserção de novas versões de tecnologia, ou até mesmo novas tecnologias, pode ser feita pela criação de transformações que as contemplam e que têm como origem o mesmo PIM usado em outras transformações que contemplam outras tecnologias. Isso facilita algumas tarefas relacionadas à alteração de tecnologias como: migração de um sistema para uma nova tecnologia ou introdução de uma nova tecnologia no desenvolvimento de um sistema. Em tese, uma vez que o PIM é o mesmo, basta executar as novas transformações para mudar para nova tecnologia.

2.1.2.1 Produtividade no desenvolvimento baseado em MDA

Bettin [3] apresenta uma comparação entre algumas métricas de três abordagens diferentes de implementação. A primeira usa a codificação totalmente manual. A segunda usa a modelagem UML para gerar a estrutura dos códigos e a terceira, o MDA para gerar a maioria do código. As métricas foram extraídas de uma aplicação pequena e simples.

A produtividade foi medida em termos de artefatos criados manualmente, ou seja, é apresentada a porcentagem de artefatos que não foram gerados automaticamente. Esses artefatos incluem código-fonte e modelos UML. O esforço gasto em porcentagem comparada com a primeira abordagem foi: 100% para a primeira abordagem, 105% para a segunda e 48% para a terceira. A comparação desses resultados sugere empiricamente que o MDA é a abordagem mais produtiva e requer menos esforço.

Um outro estudo de caso desenvolvido por Herst [13] foi feito com o propósito de verificar a afirmação de que o aumento da produtividade de desenvolvimento é impulsionado por ferramentas baseadas em MDA. Duas equipes desenvolveram a mesma aplicação. Uma usou uma ferramenta MDA, enquanto a outra equipe usou uma abordagem centrada em código com uma IDE[8] tradicional para produção de aplicação *enterprise*. Foi feita uma especificação detalhada da aplicação a ser desenvolvida e esta foi seguida pelas duas equipes. A mesma base de dados também foi usada a fim de evitar distorções no resultado final do estudo devido às diferenças na estrutura das bases de dados.

[8] IDE, do inglês *Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado, é um programa de computador que reúne características e ferramentas de apoio ao desenvolvimento de *software* com o objetivo de dar maior eficiência a esse processo. Geralmente, as IDEs unem funcionalidades para permitir que os desenvolvedores consigam um desempenho maior, desenvolvendo código com maior rapidez. Algumas características comuns em IDEs são: edição, compilação, depuração, distribuição e refatoração.

O resultado desse estudo foi que a equipe MDA desenvolveu a aplicação 35% mais rápido do que a outra equipe e em 330 horas, contra 508 horas da equipe com a IDE tradicional. Portanto, assim como apresentado por Bettin, o experimento mostrou que o MDA pode trazer ganhos significativos no desenvolvimento de *software*.

2.1.2.2 Comparação do MDA com o MDSD

A Tabela 1 mostra uma comparação entre o MDA e o MDSD apresentada por Bettin [4].

MDA	**MDSD**
Dirigido pelo desejo de desenvolver padrões que são úteis para implementadores alcançarem a interoperabilidade.	Definido por um grupo de praticantes de abordagens dirigidas por modelo que enxergam ferramentas de desenvolvimento de *software* como parte básica da infra-estrutura do software.
É baseado na fundamentação do MOF (explicado na seção 2.1.4) e na UML da OMG.	É baseado nos princípios de metamodelagem e modelagem e não está amarrado a um metamodelo ou linguagem de modelagem.
Conta com a noção de PIMs, PSMs e transformações automatizadas entre PIMs e PSMs. Assume que empresas da indústria facilmente serão capazes de entrar em acordo sobre o que constitui o termo "plataforma".	Também usa a noção de PIMs e PSMs, mas enfatiza a relatividade do termo "plataforma".
Propõe aumentar o nível de abstração das especificações de *software* para um nível independente de plataforma, para minimizar o impacto da tecnologia. Não prevê, como característica importante de competitividade, notações específicas de companhias individuais.	Propõe aumentar o nível de abstração, usando linguagens específicas de domínio, que podem ou não mapear uma notação gráfica baseada em UML. Reconhecimento de que notações específicas de domínio adicionam valor significativo se o desenvolvimento vai além do mais baixo denominador comum, o qual é inofensivo se compartilhado com competidores em uma indústria.
Enfatiza a distinção de abordagens de ferramentas CASE[9] baseando-se em padrões abertos. O *Query, Views and Transformation* (QVT[10]) [43] demonstra o comprometimento do OMG em desenvolver um padrão de indústria para transformações de modelos.	Encoraja e estimula o trabalho no padrão QVT. Entretanto, não afirma que o processo de padronização do QVT proverá uma resposta satisfatória para a interoperabilidade de ferramentas, isso porque os fornecedores representados no OMG irão querer reter a habilidade de destacar com a sua oferta específica.
Primordialmente provê a espinha dorsal conceitual e notacional. Não provê um arcabouço metodológico completo que possa ser usado para aplicar, na prática, uma abordagem dirigida por modelos.	Representa uma coleção de melhores práticas para abordagens dirigidas por modelos. Concentra nas especificidades de desenvolvimento de software distribuído em larga escala. Evita duplicação desnecessária através de consultas a melhores práticas para engenharia de linha de produto de software e desenvolvimento de software ágil.
É "agnóstico" relativo ao uso e valor da infra-estrutura de Fonte Aberta (*Open Source*).	Vê o desenvolvimento e uso da infra-estrutura de Código-fonte Aberto como um elemento essencial para a emergência futura de correntes de fornecedores de *software*.

Tabela 1 - Comparação entre o MDA e MDSD

[9] CASE significa *Computer-Aided Software Engineering*. Esta é uma classificação que abrange toda ferramenta baseada em computadores que auxiliam atividades de engenharia de *software*, desde análise de requisitos e modelagem até programação e testes.

[10] O QVT é uma especificação da OMG baseada nos padrões MOF e OCL; estes dois padrões serão apresentados logo a seguir. O QVT tem o objetivo de padronizar consultas em modelos, visões de modelos e transformações de modelos. Esta especificação ainda está em desenvolvimento.

Nesta comparação, Bettin favoreceu o MDSD, mas conforme discutido anteriormente, várias características do MDSD podem ser aplicadas ao MDA.

2.1.3 Perfil para UML

Segundo Kleppe [16], Perfil UML é um mecanismo que permite a formalização de uma linguagem representativa de uma personalização de um subconjunto da UML, acrescida de restrições para uso específico. Já existem alguns Perfis UML definidos, como por exemplo, o Perfil CORBA UML [45] e o Perfil EJB [41].

Um Perfil UML é definido por um conjunto de estereótipos, de restrições relacionadas e de valores etiquetados.

O *estereótipo* é definido por um nome e é ligado a elementos de um modelo UML. Por exemplo, no Perfil EJB, o estereótipo <<JavaClass>> é definido para uma classe na linguagem UML. Pelo estereótipo é possível definir novos elementos baseados nos que já existem na UML.

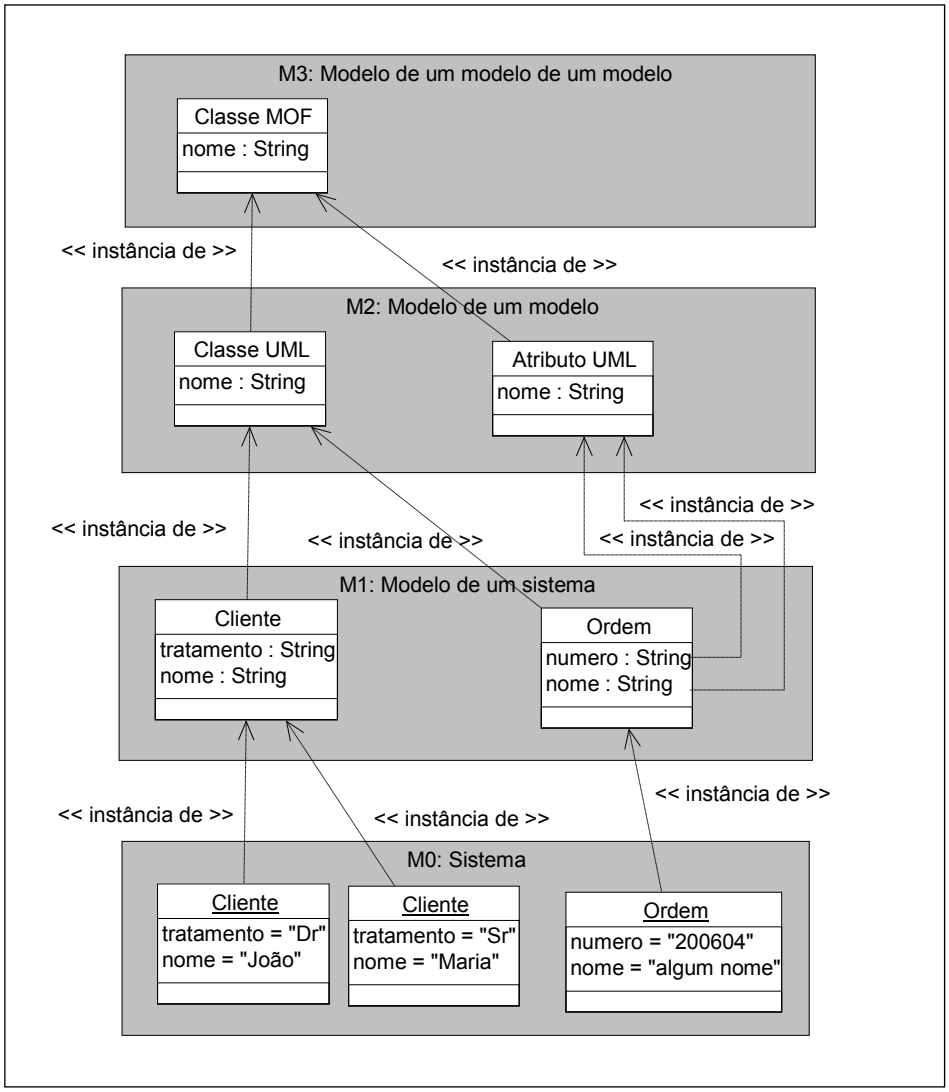
Uma *restrição* pode ser ligada a uma definição de estereótipo. Ela é expressa com *Object Constraint Language* (OCL) [33]. A OCL descreve as restrições sobre as instâncias de elementos do modelo que têm o estereótipo aplicado.

Um *valor etiquetado* é um meta-atributo adicional que é ligado a uma metaclasse do metamodelo UML. Este valor tem um nome, um tipo e é atado a um estereótipo específico. Em um modelo, pode-se atribuir o seu valor, mas somente a elementos que têm o estereótipo correspondente.

2.1.4 *Meta Object Facility* (MOF)

Conforme Kleppe [16], MOF é um padrão do OMG (*Object Management Group*) [40] que determina uma linguagem para definir linguagens de modelagem. Para entender o MOF é necessário entender o que é um metamodelo.

Para explicar o conceito de metamodelo, usaremos a Figura 2, que é a representação de quatro camadas de modelagem. Nesta figura, a camada M3 é representada pela linguagem MOF, a camada M2 é representada pela linguagem UML, a camada M1, por um modelo UML e a camada M0, pelo sistema descrito na camada anterior.



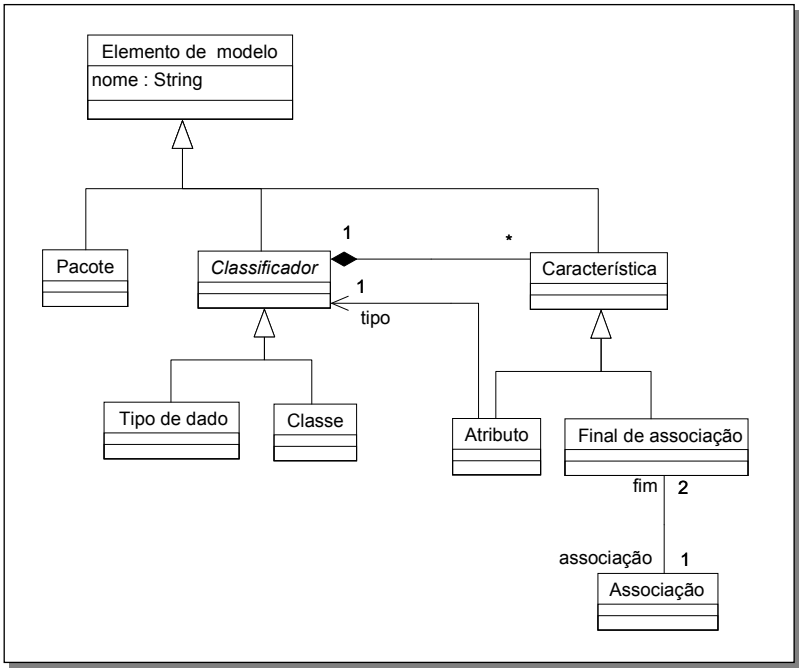
Fonte: Tradução de Kleppe [16] (p. 88)

Figura 2 - Quatro camadas de modelagem

A camada de mais alto nível, o metametamodelo MOF (M3), define uma linguagem abstrata e um arcabouço para especificar, construir e gerenciar metamodelos. Isso é fundamental para definir qualquer linguagem de modelagem como a UML ou até mesmo o próprio MOF. Todos os metamodelos, tanto os padronizados quanto os personalizados definidos pelo MOF, são posicionados na camada M2. Os modelos do mundo real, representados pelos conceitos definidos no metamodelo correspondente na camada M2, por exemplo o metamodelo UML, estão na camada M1. Finalmente, na camada M0 estão as representações de instâncias de conceitos do mundo real. O propósito das quatro camadas com um metametamodelo comum é suportar múltiplos metamodelos e modelos de tal forma que sejam escaláveis, possibilitando extensibilidade, integração e gerenciamento genérico de modelo e metamodelo. É pelo gerenciamento genérico que se faz possível a manipulação programática de um modelo (camada M1) descrito por um metamodelo (camada M2) determinado.

Apesar de o exemplo usado aqui ser a definição da própria linguagem UML, que de fato é definida pelo MOF, este pode ser usado para definir qualquer linguagem de modelagem.

A Figura 3 é uma descrição simplificada de parte do MOF. A caracterização completa pode ser encontrada na especificação do MOF [42]. Atualmente, existem algumas linguagens definidas pelo MOF. Além da UML, podemos citar o *Common Warehouse Metamodel* (CWM) [39].



Fonte: Tradução de Kleppe [16] (p. 132)

Figura 3 - Metamodelo MOF simplificado

2.2 Bibliografia relacionada

Esta seção, dividida em três partes, descreve vários trabalhos de modelagem da interface com o usuário. A pesquisa sobre esse assunto foi necessária, pois os trabalhos encontrados relacionados com o MDA não abordam a modelagem de interface com o usuário adequadamente.

Primeiramente, foram descritas algumas propostas de modelagem da interface com o usuário apresentadas antes do surgimento do MDA. Em cada proposta serão destacados aspectos relevantes que podem ser usados como base para elaboração de modelos, conforme especificado no MDA. A segunda parte apresenta algumas abordagens de modelagem em que não há preocupação em modelar a interface com o usuário, independentemente de plataforma. A terceira

parte apresenta algumas abordagens de modelagem que, de alguma forma, já consideram a modelagem da interface com o usuário um aspecto importante no modelo independente de plataforma do MDA.

2.2.1 Modelagem de interface com o usuário

Rosenberg [24] apresenta uma abordagem prática de como fazer análise orientada por objetos. Segundo o autor, essa abordagem funciona para grande variedade de projetos e o objetivo é guiar a codificação a partir de casos de uso[11]. Embora o foco seja na análise, ele descreve algumas atividades em que são identificadas e modeladas em alto nível as interfaces com o usuário. O autor sugere que os objetos identificados sejam separados em três tipos:

- **Fronteira:** objetos que fazem interface com o usuário.
- **Entidade:** usualmente são objetos do modelo de domínio.
- **Controle:** serve como uma junção entre os objetos de fronteira e de entidade.

O nível de detalhes modelado nessa abordagem não é suficiente para representar uma interface com o usuário de forma completa. Entretanto, a abordagem de Rosenberg tem foco em sempre demonstrar os relacionamentos entre a interface com o usuário e os objetos de domínio que são representados pelas entidades. A modelagem desses relacionamentos é essencial quando se pensa em modelar interface com o usuário com os objetivos do MDA.

Van Harmelen [31] reúne diversos autores para apresentar técnicas de desenho e modelagem de interface com o usuário. Entre elas pode-se destacar duas abordagens, a *Entity, Task, and Presenter* (ETP), proposta por Artim, e a Wisdom, proposta por Nunes [19]. Os dois propõem extensões na UML para modelar a interface com o usuário.

A modelagem seguindo o ETP é dividida em três grupos de elementos:

- **Entidade *(Entity):*** corresponde aos conceitos do domínio do usuário, ou seja, o que o usuário percebe em seu mundo real. É o mesmo conceito usado por Rosenberg já apresentado.

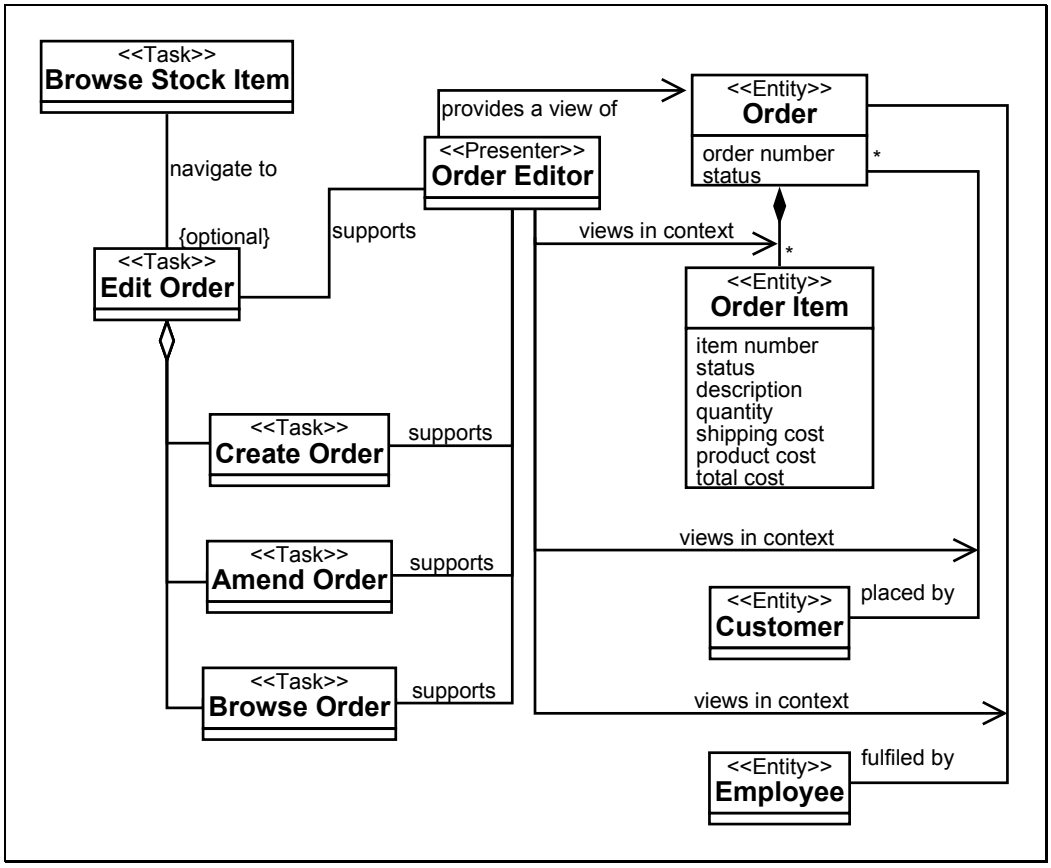
[11] Caso de uso é a especificação de uma sequência de ações que um sistema executa na interação com atores (normalmente uma pessoa, mas pode ser uma entidade externa, como um outro sistema) externos para prover um serviço de valor.

- **Tarefas *(Tasks):*** captura de alguma forma o conhecimento procedural que manipula os conceitos do domínio.
- **Apresentador *(Presenter):*** serve como uma representação no espaço do problema que exibe um conjunto de entidades de forma otimizada para ajudar alguém a completar uma ou mais tarefas.

A Figura 4 mostra um exemplo de diagrama de classes do ETP. Nele está ilustrada a maneira como os três grupos se relacionam. O diagrama mostra o apresentador, *Order Editor*, junto com as tarefas que o apresentador suporta (*Edit Order*, *Create Order*, *Amend Order* e *Browse Order*) e junto com as entidades exibidas pelo apresentador (*Order*, *Order Item*, *Customer* e *Employee*). Cada um dos três grupos é marcado apropriadamente com os estereótipos UML: *Entity*, *Task* e *Presenter*.

A associação *provides a view of* é unidirecional e aponta para a entidade que o apresentador irá mostrar. Várias associações rotuladas de *views in context* conectam o apresentador *Order Editor* com as associações entre *Order* e suas entidades associadas. O alvo dessas associações é a entidade a ser exibida no apresentador, mas como a entidade pode ser conectada por várias associações, simplesmente apontar a entidade não ajuda muito. Pela associação *views in context* que aponta para uma associação, não para uma entidade, o leitor do diagrama pode ver não só qual entidade está sendo exibida, mas também o papel dela na tarefa. É importante destacar que nesse exemplo seria possível representar, por meio de notas (elemento UML) nas associações *provides a view of* e *views in context*, os campos da entidade que devem ser disponibilizados pelo apresentador.

Vamos supor que o requisito determine que o usuário precise pesquisar freqüentemente pela informação de item de estoque para completar a ordem. No diagrama, essa pesquisa é representada pela tarefa *Browse Stock Item*. A navegação para esta tarefa é representada pela associação *navigate to*, que é opcional, ou seja, o usuário pode ou não acionar a tarefa *Browse Stock Item*. Portanto, na interface final do apresentador *Order Editor* deve aparecer um mecanismo para iniciar a tarefa *Browse Stock Item*.



Fonte: Artim [31] (p. 141)

Figura 4 - Um exemplo de diagrama de classes do ETP

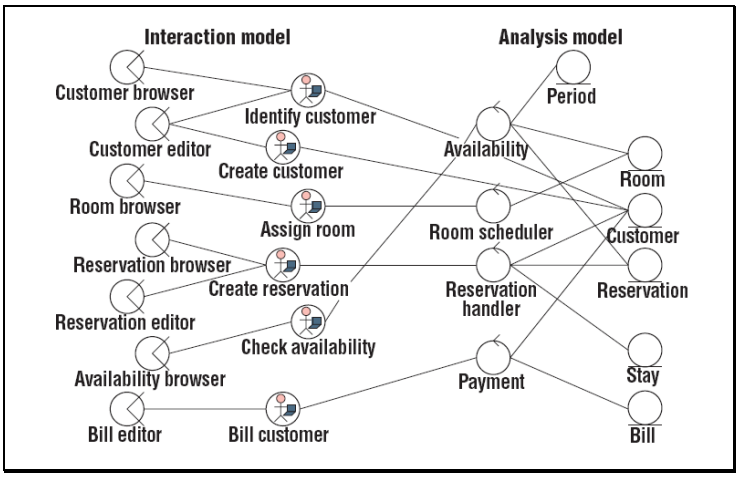
O importante desse modelo é o fato de ele deixar bem claro o relacionamento entre as entidades, tarefas e apresentadores. Ele representa informações importantes que são necessárias para a geração de código-fonte funcional. Por ele é possível identificar diretamente os campos e as associações das entidades necessárias para realizar uma tarefa e como eles são agrupados e apresentados ao usuário.

No Wisdom [19], existe uma classificação que divide o modelo em 4 grupos:

- **Entidade (*Entity*):** elemento do modelo de análise padronizado na UML. Este grupo de classes modela informações persistentes e comportamentos específicos da própria entidade.
- **Controle (*Control*):** elemento do modelo de análise padronizado na UML. Este grupo de classes representa coordenação, transações e controle de vários objetos. Nele devem ser encapsuladas lógicas complexas que não podem ser relacionadas a uma classe de entidade específica.
- **Tarefa (*Task*):** elemento do modelo de interação do Wisdom. Este grupo de classes estrutura a interação entre o usuário e o sistema, mapeia as entidades nas classes de espaço de interação e é responsável pela consistência entre múltiplos espaços de interação.

- **Espaço de interação (*Interaction space*):** elemento do modelo de interação do Wisdom. Este grupo de classes modela a interação entre o sistema e os usuários humanos.

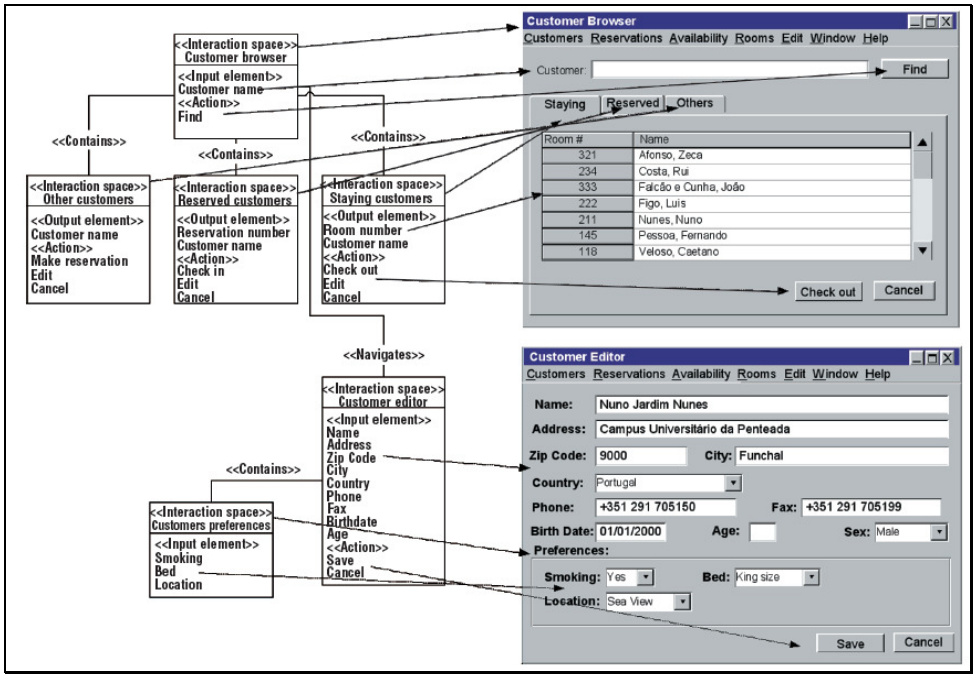
A Figura 5 representa um exemplo de artefato de análise do Wisdom de um sistema de reserva em hotel. As figuras sublinhadas representam as classes de entidade. As figuras com seta no topo representam classes de controle. As que têm um boneco e um computador são classes de tarefa e aquelas com seta no meio são classes de espaço de interação. Este exemplo mostra uma visão geral das associações entre as classes que o Wisdow permite modelar.



Fonte: Nunes [19] (p. 118)

Figura 5 - Exemplo de artefato de análise do Wisdom

A Figura 6 mostra como pode ser feita uma interface gráfica a partir de um dos artefatos do Wisdom. Esse exemplo ilustra as interfaces gráficas para os espaços de interação: Procura de cliente (*Customer browser*) e Edição de cliente (*Customer editor*).



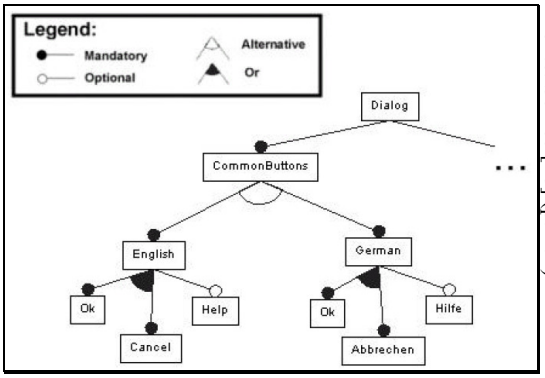
Fonte: Nunes [19] (p. 119)

Figura 6 - Exemplo do mapeamento de um modelo do Wisdow em interface gráfica

A Figura 5 e a Figura 6 são os diagramas mais relevantes para geração de código funcional. Além desses, existem outros diagramas de caso de uso, atividade e classe. Os apresentados são importantes já que neles estão contidas algumas informações que ajudam no processo de transformação em código-fonte executável e funcional. A partir desses diagramas é possível identificar quais são as entidades envolvidas em cada tela e quais os campos e comandos que deverão ser exibidos. Sobre esse aspecto, o Wisdom é bem parecido com o ETP, entretanto este permite especificar em nível mais detalhado quais são os campos e associações de cada entidade que farão parte da tela. Isso é possível por meio de notas nas associações entre o apresentador e a entidade.

Schlee [28] apresenta uma forma de descrever o comportamento e algumas restrições de interface de usuário independentemente de plataforma. Seu trabalho é baseado na Programação Gerativa [7] e compreende apenas a modelagem da interface, não fazendo referência ao modelo de dados. A Figura 7 mostra um exemplo do diagrama usado, o diagrama de características. Com ele é possível representar características obrigatórias, alternativas, opcionais e conjuntas (ou). Essa abordagem é interessante pelo fato de usar uma representação simples de características que são comuns nas interfaces com o usuário. Neste exemplo, há uma caixa de dialogo que tem três botões, que poderão estar em Inglês ou em Alemão. Isso é modelado pela associação, com característica de alternativa, entre *CommonButtos*, *English* e *German*. O usuário obrigatoriamente terá que acionar o comando *Ok* ou *Cancel*. Isso é modelado pela associação entre *English*, *Ok* e *Cancel, que tem característica obrigatória e conjunta (ou)*. O comando *Help* pode ou não ser acionado pelo usuário. Isso é modelado pela associação, com característica de opcional, entre

English e *Help*. A modelagem dos comandos em Alemão é idêntica à modelagem dos comandos em Inglês, a única diferença são os nomes do comandos.

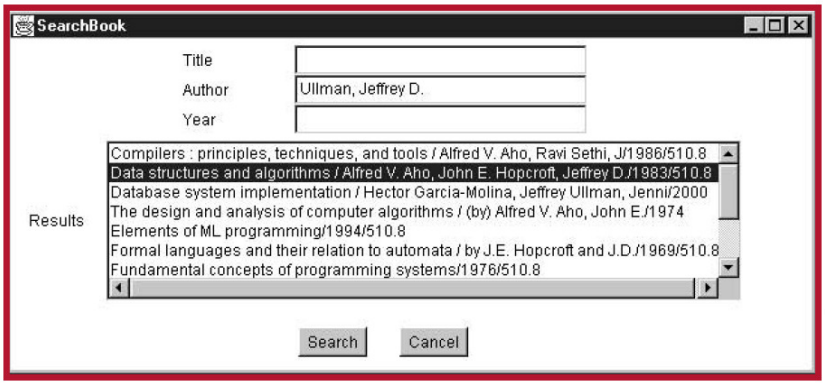


Fonte: Schlee [28] (p. 404)

Figura 7 - Exemplo de diagrama de características

Pinheiro da Silva [21] apresenta a UMLi, que é um metamodelo usado para modelar interface com o usuário que integra totalmente com a UML. Com a UMLi é possível modelar a interface abstrata com o usuário, que é uma modelagem independente de plataforma. Também é possível mapear as classes abstratas em representações concretas que consideram aspectos específicos da plataforma. Além disso, ela permite modelar as ações a serem executadas pelo usuário. O autor classifica a UMLi como a proposta mais madura tecnicamente para desenvolvimento de interface, o que realmente pode ser procedente, já que ele tentou reunir na UMLi os seguintes princípios:

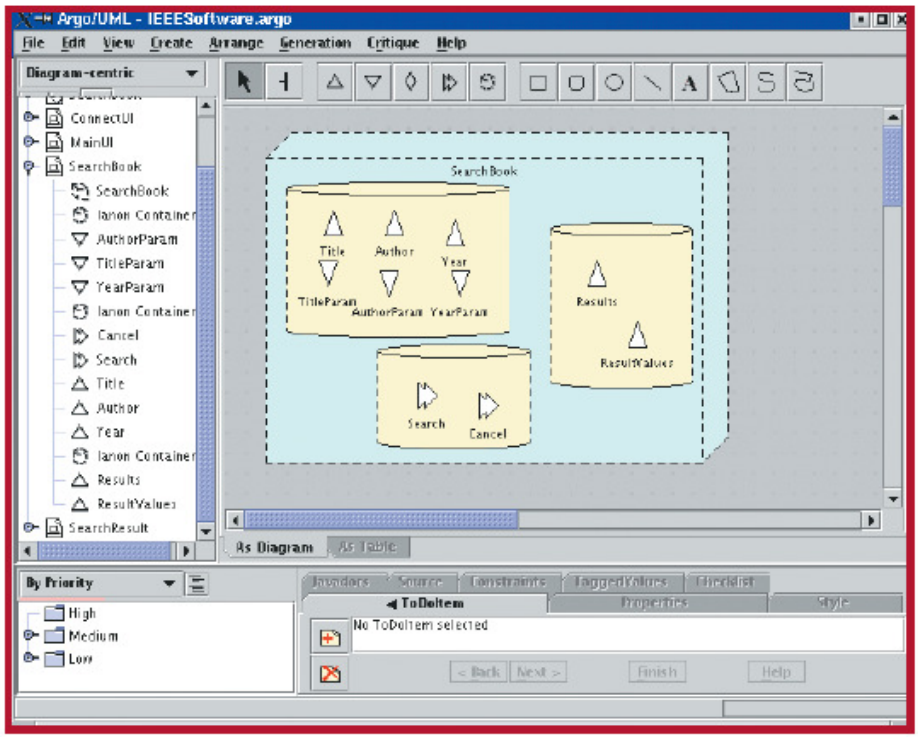
- Ser modesta, mantendo o padrão UML como um subconjunto no qual construções existentes continuem com suas regras e semântica.
- Sustentar expectativas de usuários correntes da UML, para os quais a experiência com a linguagem deveria ajudar a usar extensões da UML específicas para interface.
- Sustentar expectativas de usuários que fazem modelagem de interface, os quais têm experiência em técnicas de modelagem. Estes usuários não deveriam sentir que o desenho de interfaces seria mais limitado que as técnicas existentes.
- Suportar aplicações completas, de tal forma que as ligações entre os modelos de interface com o usuário e modelos UML sejam bem definidos e próximos.
- Introduzir o mínimo possível de modelos novos dentro da UML.



Fonte: Pinheiro da Silva [21] (p. 63)

Figura 8 - Exemplo de interface com o usuário de pesquisa por livros

A Figura 8 apresenta uma tela simples de pesquisa por livros. Essa tela é usada como exemplo para mostrar como uma interface com o usuário seria modelada em um diagrama UMLi. A Figura 9 apresenta a modelagem desta tela e, logo a seguir, os elementos UMLi usados nesse diagrama são descritos.



Fonte: Pinheiro da Silva [21] (p. 66)

Figura 9 - Diagrama de interface com o usuário da UMLi que modela a pesquisa por livros

O diagrama UMLi de interface com usuário é formado por seis elementos de modelagem:

- ***FreeContainers:*** desenhado em forma de cubos pontilhados. Um *FreeContainer* é uma classe de interação de alto nível que nenhuma outra classe de interação pode conter.

- ***Conteiners:*** desenhado em forma de cilindros pontilhados. Um *Container* é um mecanismo que agrupa classes de interação que não são *FreeContainers*.
- ***Inputters:*** desenhado em forma de triângulos de cabeça para baixo. Um *Inputter* recebe informação do usuário.
- ***Editors:*** não mostrado no diagrama da Figura 9. Um *Editor* facilita a troca de informação em duas vias.
- ***Displayers:*** desenhado em forma de triângulos que apontam para cima. Um *Displayer* envia informação para o usuário.
- ***ActionInvokers:*** desenhado em forma de setas que apontam para a direita. Um *ActionInvokers* recebe instruções diretas do usuário.

Além do diagrama de interface com o usuário, a UMLi usa o diagrama de caso de uso e o de atividade para representar a noção de tarefa como conceituada na comunidade MB-UIDE[12]. O diagrama de atividade da UMLi é uma extensão do diagrama de atividade da UML.

Embora este modelo pareça ser o mais maduro encontrado na literatura, ele não apresenta mecanismos para informar quais são os atributos de entidade utilizados na interface com o usuário. Para geração de código funcional esse tipo de informação é essencial e poderia ser resolvido da forma que foi proposta no ETP.

2.2.2 Geração da interface com o usuário a partir do modelo de domínio

Pleumann [23] propõe a noção de um ambiente MDR (*Model-Driven Runtime*) que é capaz de executar um modelo independente de plataforma para um propósito específico em vez de transformá-lo, como é feito no MDA. Ele apresenta também uma implementação da abordagem proposta que já foi usada em várias aplicações. A aplicação final é Web com interface gráfica, mas esta é inferida a partir do modelo de domínio. Como não existe um modelo de interface com

[12] A sigla MB-UIDE vem do termo em inglês *Model-Based User Interface Development Environments*, que se refere a abordagens que provêem a capacidade de projetar e implementar interfaces de usuários de forma declarativa e sistemática. Uma pesquisa de tecnologias MB-UIDE foi feita por Pinheiro da Silva [22]. Nesta pesquisa são apresentados 14 MB-UIDEs que são descritas, comparadas e analisadas a partir das informações encontradas na literatura.

o usuário, se houver requisitos de usabilidade específicos, estes não poderão ser atendidos. Se algum mecanismo for feito para atender esse requisito dentro dessa arquitetura, regras de negócio ficarão misturadas com regras de interação com o usuário. Alguns autores, entre eles Rosenberg [24], não aconselham misturar esses dois aspectos.

Kleppe [16] explica detalhadamente o funcionamento do MDA. Em seus exemplos, a interface com o usuário é representada somente no PSM e é gerada automaticamente a partir do PIM que representa o modelo de domínio sem modelagem de interface com o usuário. O autor defende que requisitos específicos de interface com o usuário podem ser adicionados ao PSM. Entretanto, essa abordagem de modelagem de interface torna os requisitos de interface dependentes da plataforma, o que não é alinhado a alguns dos objetivos do MDA, tais como: portabilidade e independência de plataforma.

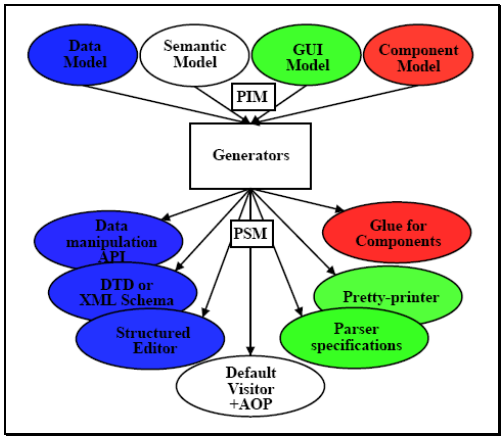
2.2.3 Modelagem de interface com o usuário no MDA

No trabalho de Courbis [6] é apresentada uma abordagem de desenvolvimento baseada em transformações de diferentes modelos de negócio. Essa abordagem é compatível com o MDA e depende da Programação Gerativa [7]. Para ilustrar essa abordagem, é apresentado um sistema chamado *SmartTools*.

Os principais resultados esperados por Courbis são: i) atingir melhor qualidade de *software* graças à separação dos modelos de negócio dos modelos de tecnologia; ii) ter um código mais direto; iii) permitir o desenvolvimento rápido; iv) facilitar a portabilidade de aplicações para novas tecnologias e plataformas.

Conforme ilustrado na Figura 10, nessa abordagem o PIM é dividido basicamente em quatro tipos de modelos: de Dados, Semântico, de Interface Gráfica com o Usuário (GUI[13]) e de Componentes. O Modelo de Dados é a base para todos os outros modelos. O Modelo Semântico representa as operações realizadas sobre os dados. O Modelo de GUI representa as visões que o usuário pode ter dos dados e o de Componentes representa como os dados com suas semânticas são encapsulados em serviços distribuídos.

[13] A sigla GUI é do termo em inglês *Graphical User Interface*.



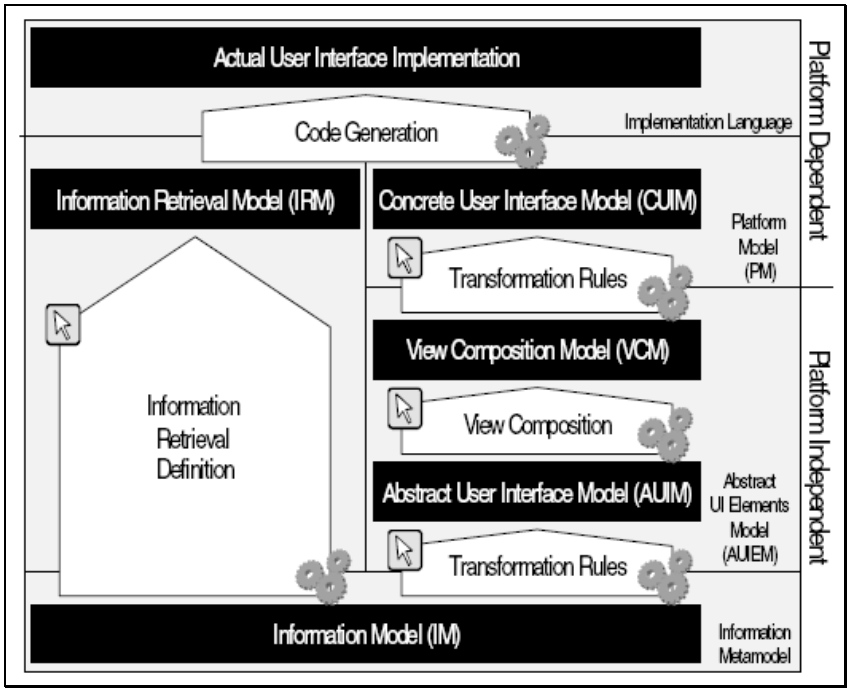
Fonte: Courbis [6] (p. 3)

Figura 10 - Abordagem MDA e de Programação Gerativa

O trabalho de Courbis se baseia no MDA pelo fato de ter modelos com características do PIM e do PSM e por aplicar transformações sobre os modelos conforme especificado pelo MDA. Entretanto, Courbis não deixa claro quais são os metamodelos usados no PIM e no PSM. Além disso, ele tende a representar os modelos diretamente com XML [5], em vez de usar extensões da UML ou metamodelos MOF.

Schattkowsky apresenta um artigo [27] no qual é proposta uma abordagem, semelhante ao MDA, para interfaces com o usuário baseada em transformações de modelos UML em diferentes níveis de abstração. Isso permite o desenho independente de plataforma de interfaces com o usuário e uma separação da lógica de aplicação dessa interface.

Ao comparar com o MDA, percebe-se que a abordagem do Schattkowsky trata a modelagem de interface com o usuário desde o PIM. Conforme ilustrado na Figura 11, o autor propõe um fluxo de desenho com três PIMs e dois PSMs. Os PIMs seriam o IM (*Information Model*) referente à modelagem dos dados do domínio da aplicação, o AUIM (*Abstract User Interface Model*) e o VCM (*View Composition Model*) referentes à modelagem da interface com o usuário. Basicamente, o AUIM combina os dados do IM com os elementos abstratos de interface com o usuário para acessar e manipular dados. O VCM separa o AUIM em varias visões sobrepostas e navegáveis. Os PSMs seriam o IRM (*Information Retrieval*) referente à implementação de acesso aos dados e o CUIM (*Concrete User Interface Model*) referente à implementação da interface com o usuário.



Fonte: Schattkowsky [27] (p. 203)

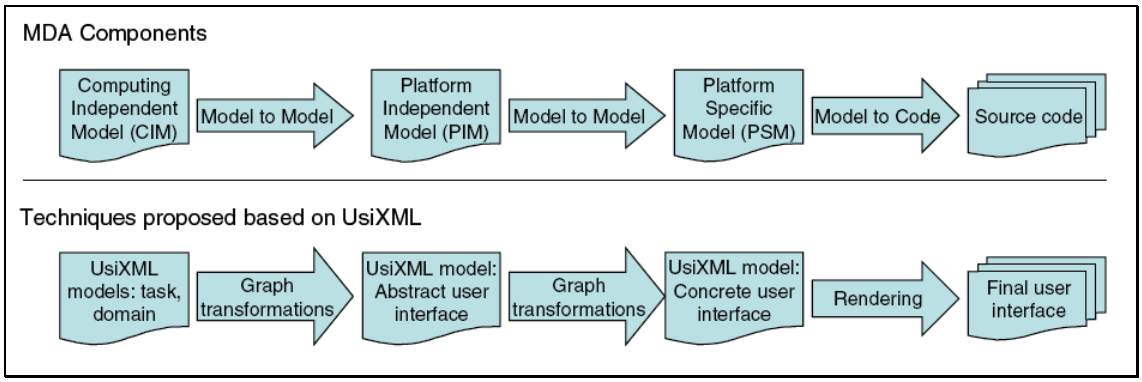
Figura 11 - Fluxo de desenho de interface com o usuário proposto por Schattkowsky

O trabalho do Schattkowsky reforça a necessidade de existir um PIM específico para modelar a interface com o usuário. Além disso, é proposto que o PIM de interface seja gerado automaticamente a partir do IM, que seria o modelo de dados que é a base para toda aplicação. Essa abordagem faz sentido, uma vez que as primeiras abordagens MDA geravam a modelagem de interface com o usuário automaticamente por meio de transformações. No entanto, o modelo gerado era um PSM. Isso significa que, se houver necessidade de mudar a plataforma, tudo o que foi alterado no PSM de interface com o usuário para atender requisitos específicos teria que ser refeito, voltando a um dos problemas que o MDA se propõe resolver, que é a independência de plataforma. Outro ponto forte deste trabalho foi a proposta de descrição dos modelos em extensões da UML.

Ao tentar entender cada modelo proposto, conseguimos identificar alguns pontos que poderiam ser melhorados. No trabalho do Schattkowsky, a descrição dos modelos de interface com o usuário não foram detalhados suficientemente a ponto de permitir a melhor compreensão de como funcionaria a modelagem proposta. A princípio, pelo que foi apresentado, a modelagem ficaria muito detalhada, o que pode levar a um modelo muito grande e de difícil geração, manutenção e leitura humana. O PIM, como definido no MDA, exige uma formalidade que permita que ele seja interpretado pelo computador com o objetivo de gerar outro modelo mais detalhado. Geralmente, o PIM tem que ser gerado integralmente por humanos ou, como proposto pelo autor, pode ser parcialmente gerado, mas, de qualquer forma, a parte gerada tem que sofrer alterações realizadas por humanos. Apesar disso, se o metamodelo do PIM exigir um modelo muito detalhado, gerar ou dar manutenção no PIM manualmente pode ser uma tarefa inviável.

Schattkowsky não demonstrou preocupação com usabilidade. Como o trabalho é referente à interface com o usuário, trabalhar a usabilidade é um aspecto importante que não pode ser desconsiderado. Outra possível melhoria seria considerar esses aspectos.

Vanderdonckt [32] apresenta um trabalho preocupado em considerar aspectos de usabilidade. Seus modelos são todos baseados no formato *USer Interface eXtensible Markup Language* (UsiXML – http://www.usixml.org), que é uma Linguagem de Descrição de Interface com o Usuário bem-formada, baseada em esquema XML. Os modelos e transformações propostos podem ser mapeados na classificação do MDA, conforme mostrado na Figura 12. Esses modelos são de: domínio, tarefa, interface abstrata com o usuário, interface concreta com o usuário e interface final com o usuário.



Fonte: Vanderdonckt [32] (p. 28)

Figura 12 - Distribuição dos modelos UsiXML de acordo com a classificação MDA

Conforme apresentado pelo autor, já existem várias ferramentas, que em conjunto, suportam as transformações para os modelos UsiXML da Figura 12. Inclusive, existem ferramentas para gerar aplicações em plataformas diferentes.

Tal como proposto pelo padrão MOF, a semântica dos modelos UsiXML é baseada em metamodelos expressos em termos de diagramas de classes UML. Entretanto, as ferramentas atuais que suportam o UsiXML não têm funcionalidades que mapeiam formalmente o metamodelo ao modelo, que é uma característica essencial existente no MOF. Por exemplo, as ferramentas baseadas no MOF permitem modelar formalmente o metamodelo e, a partir dessa definição formal, as ferramentas conseguem criar mecanismos de manipulação do modelo definido pelo metamodelo. Como exemplo desses mecanismos, podemos citar a geração de uma API para manipular programaticamente o modelo.

O MDA é baseado nos padrões XMI e MOF para evitar padrões específicos que não interoperam entre ferramentas CASE. O UsiXML inviabiliza essa interoperação. Além disso, já existem ferramentas que implementam os conceitos de XMI e MOF, permitindo automações básicas relacionadas aos modelos, como, por exemplo, persistência, leitura e manipulação

programática dos modelos, além da interoperabilidade. Portanto, a abordagem apresentada por Vanderdonckt [32] não segue os padrões MOF e XMI do MDA, ficando, assim, a desejar no que se refere ao aspecto de interoperabilidade entre ferramentas CASE.

Capítulo 3

Metodologia

3.1 Tipo de pesquisa

Observando o método científico e conforme Jung [15], é importante salientar que a presente pesquisa é tecnológica, com objetivos de caráter descritivo, que utiliza procedimento experimental e foi desenvolvida em laboratório com fins educativos.

Trata-se de pesquisa tecnológica, uma vez que tem como objeto a aplicação de conhecimentos fundamentais, como o uso do paradigma MDA, para geração de novos conhecimentos sobre o assunto. A elucidação sobre o tema, em conjunto com a experimentação desses conceitos, é essencial para a aplicação em contextos reais.

A pesquisa tem objetivos de caráter descritivo por registrar e analisar o processo de aplicação do MDA do início ao fim, considerando a definição de modelos, implementação de transformações e aplicação desses modelos e transformações no desenvolvimento de *software*.

O trabalho é considerado experimental, já que simula condições reais de desenvolvimento de *software*. Neste sentido, se permitiu inserir conceitos de interface com o usuário e usabilidade dentro do paradigma MDA, sem desrespeitar suas regras e padrões. Conforme apresentado na seção 2.2.1, tais conceitos já existem na literatura, mas não haviam sido abordados dentro do MDA.

A pesquisa foi realizada em ambiente laboratorial pelo fato de sua aplicação ter sido guiada por uma especificação de requisitos de um produto desenvolvido com fins educacionais, em que as variáveis inerentes ao desenvolvimento de *software* são controladas.

3.2 Procedimentos metodológicos

O primeiro passo foi a definição de quais tecnologias seriam usadas no trabalho. Assim, foi definido que seria uma aplicação Web usando o *Hibernate* [2] como mecanismo de persistência e o *Struts* [50] para a parte de interface com o usuário. O primeiro é uma biblioteca de mapeamento objeto/relacional amplamente utilizado e aceito no mercado. Ele ganhou tanto destaque que o padrão de persistência do EJB 3.0[14] da especificação J2EE 5.0[15] foi completamente alterado em relação à versão anterior e foi totalmente baseado no *Hibernate*. A persistência do EJB 3.0 não foi usada neste trabalho porque ainda não havia sido lançada formalmente. O *Struts* é uma biblioteca de desenvolvimento Web de interfaces gráficas que também é amplamente utilizado e largamente aceito no mercado. Atualmente, existem várias alternativas a ele, mas nenhuma se destaca isoladamente. O *Struts* foi escolhido por ser usado no laboratório Synergia e pelo conhecimento prévio dessa tecnologia pelo pesquisador.

Em segundo lugar, foi definida a ferramenta de apoio ao desenvolvimento das transformações e a definição dos modelos necessários para que o desenvolvimento seguisse o paradigma MDA. A ferramenta escolhida foi o *Rational Software Architect* (RSA).

De acordo com o MDA, foi necessário definir o PIM e o PSM. Em seguida, foi necessário definir as regras de transformação de PIM para PSM e de PSM para código-fonte. A definição do PIM foi baseada nos trabalhos discutidos na seção 2.2, conforme descrito a seguir.

Nos primeiros PIMs definidos, conforme apresentado por Kleppe [16], a utilização do MDA se dava de tal forma que, no final do processo, um sistema é gerado a partir de um PIM que representa o modelo de domínio. Essa abordagem deixa de modelar o aspecto de interação com o usuário em um nível elevado de abstração. Este aspecto foi deixado para ser tratado no PSM que é gerado automaticamente. Mesmo que esse PSM seja alterado para refletir requisitos de usabilidade, a clareza deste modelo é prejudicada, pois ele mistura aspectos de interação com o usuário com detalhes de implementação dessa interação, usando uma tecnologia específica.

[14] EJB significa *Enterprise JavaBeans*. A arquitetura do EJB é para desenvolvimento e implantação de aplicações baseadas em componentes de negócio. Aplicações escritas usando a arquitetura EJB são escaláveis, transacionais e seguras para acesso de vários usuários. Sua especificação está disponível em: http://jcp.org/aboutJava/communityprocess/pr/jsr220/index.html.

[15] J2EE significa *Java 2 Platform, Enterprise Edition*, ele é um padrão da indústria para desenvolver aplicações Java (do lado do servidor) portáveis, robustas, escaláveis e seguras. Sua especificação está disponível em: http://jcp.org/aboutJava/communityprocess/pfd/jsr244/.

Quando as interfaces não têm muitos requisitos específicos de usabilidade, essa abordagem poupa bastante trabalho ao modelar a interface com o usuário, pois esta é gerada automaticamente.

Nos trabalhos de Schlee [28], Schattkowsky [27], Vanderdonckt [32] e Courbis [6], apresentados na seção 2.2.3, já existe no PIM uma representação da interface com o usuário. Apesar disso, como discutido anteriormente, estes trabalhos não descrevem o modelo PIM usado e falham em seguir todos os aspectos da abordagem MDA. Por isso, optou-se por estudar melhor o assunto e propor uma abordagem de transformação para a interface com o usuário.

Conforme apresentado na seção 2.2.1, ao estudar aspectos de usabilidade percebeu-se que antes do surgimento do paradigma MDA já existiam tentativas de modelar a interação com o usuário em um nível mais alto da abstração, em que aspectos de tecnologia e plataforma são desconsiderados e o foco é em como a tarefa deve ser realizada para atender às necessidades de negócio que o sistema se propõe atender. Este tipo de modelagem se adequa totalmente ao conceito do PIM. Portanto, além das abordagens levantadas no parágrafo anterior, que usam o MDA, foram levantadas mais quatro abordagens de representações de interface com o usuário que não utilizam o MDA. Entre elas, três são extensões da UML com características parecidas que permitem modelar a interface com o usuário considerando o modelo de domínio e questões de usabilidade.

Por causa das questões apresentadas acima, decidimos acrescentar ao PIM a modelagem de interface com o usuário. Dessa forma, o PIM ficou separado em dois modelos, que, neste trabalho, denominaremos **PIM *Domain*** e **PIM UI**, em que UI se refere ao termo em inglês *User Interface*. Esses dois modelos serão definidos e descritos mais detalhadamente na seção 4.1.1.

O PIM UI foi baseado nos trabalhos de Van Harmelen [31], Rosenberg [24], Schlee [28], Pinheiro da Silva [21], Courbis [6], Schattkowsky [27] e Vanderdonckt [32].

Com o intuito de manter o mesmo grau de automatização, em relação à interface de usuário, alcançado pelos primeiros PIMs, foi decido que o PIM UI seria gerado a partir de uma transformação sobre o PIM *Domain*. No entanto, o PIM UI poderia ser alterado para atender requisitos específicos de usabilidade. No formato realizado tradicionalmente (PIM transformado diretamente em PSM), a modelagem desses requisitos seria feita diretamente sobre o PSM. Modelando tais requisitos no PIM UI, podemos alcançar dois benefícios inerentes ao conceito do PIM, como se pode conferir a seguir:

1. Ao se alterar a plataforma da aplicação, os requisitos modelados no PIM *Domain* e no PIM UI não seriam perdidos, pois, de acordo com a arquitetura do MDA, os PIMs seriam mantidos e os PSMs seriam gerados novamente para nova plataforma. Na forma tradicional, o PSM teria que ser novamente alterado para refletir os requisitos de interface com o usuário, pois tradicionalmente esses requisitos teriam que ser expressos diretamente no PSM que modela a interface com o usuário.
2. A modelagem desses requisitos fica limpa, clara e com nível de abstração adequado, bem próximo ao que foi proposto por alguns autores de usabilidade, conforme apresentado na

seção 2.2.1, antes da existência do MDA. Detalhes de tecnologias e plataforma não seriam considerados neste modelo. Dessa forma, consegue-se manter alto nível de abstração em relação à modelagem da interação. Os detalhes de tecnologias e plataforma seriam tratados em um nível de abstração mais baixo, na transformação de PIM para PSM e no próprio PSM.

O PSM também foi dividido em dois, da mesma forma usada como o PIM. Um é utilizado para representar o modelo de domínio implementado com o uso da tecnologia *Hibernate* e outro representa o modelo de interface com o usuário implementado usando a tecnologia *Struts*. Esses modelos foram denominados **PSM Hibernate** e **PSM Struts**. Esta divisão foi necessária para separar de forma clara a implementação do modelo de domínio da implementação do modelo de interface com usuário e pelo fato de que as tecnologias são totalmente independentes uma da outra. Qualquer das duas poderia ser substituída por outra tecnologia. Neste caso, seria necessário pelo menos um PSM para a tecnologia substituta.

Com a subdivisão do PIM e do PSM, também foi necessário desdobrar as transformações previstas no MDA. As transformações necessárias ficaram da seguinte forma:

- Transformação de PIM para PIM:
  - Transformação de PIM *Domain* para PIM UI.
- Transformação de PIM para PSM:
  - Transformação de PIM *Domain* para PSM *Hibernate*.
  - Transformação de PIM UI para PSM *Struts*.
- Transformação de PSM para código-fonte:
  - Transformação de PSM *Hibernate* para código-fonte.
  - Transformação de PSM *Struts* para código-fonte.

Portanto, conforme ilustrado na Figura 13, o processo para criação da aplicação fica da seguinte forma:

- **Passo 1:** A partir do modelo de análise e da especificação de requisitos do *software* é criado, manualmente, o PIM *Domain*.
- **Passo 2:** O PIM UI é gerado automaticamente por transformação a partir do PIM *Domain*.
- **Passo 3:** Se necessário, o PIM UI é alterado manualmente para contemplar requisitos do software e de usabilidade.

- **Passo 4 e 5:** O PSM *Hibernate* e o PSM *Struts* são gerados automaticamente pelas transformações a partir do PIM *Domain* e do PIM UI, respectivamente.
- **Passo 6 e 7:** Se necessário, o PSM *Hibernate* e o PSM *Struts* são alterados manualmente para satisfazer questões técnicas.
- **Passo 8 e 9:** O código-fonte é gerado automaticamente por transformações a partir do PSM *Hibernate* e do PSM *Struts*.
- **Passo 10 e 11:** As regras de negócios documentadas de forma textual no PIM *Domain* e no PIM UI são codificadas diretamente no código-fonte.

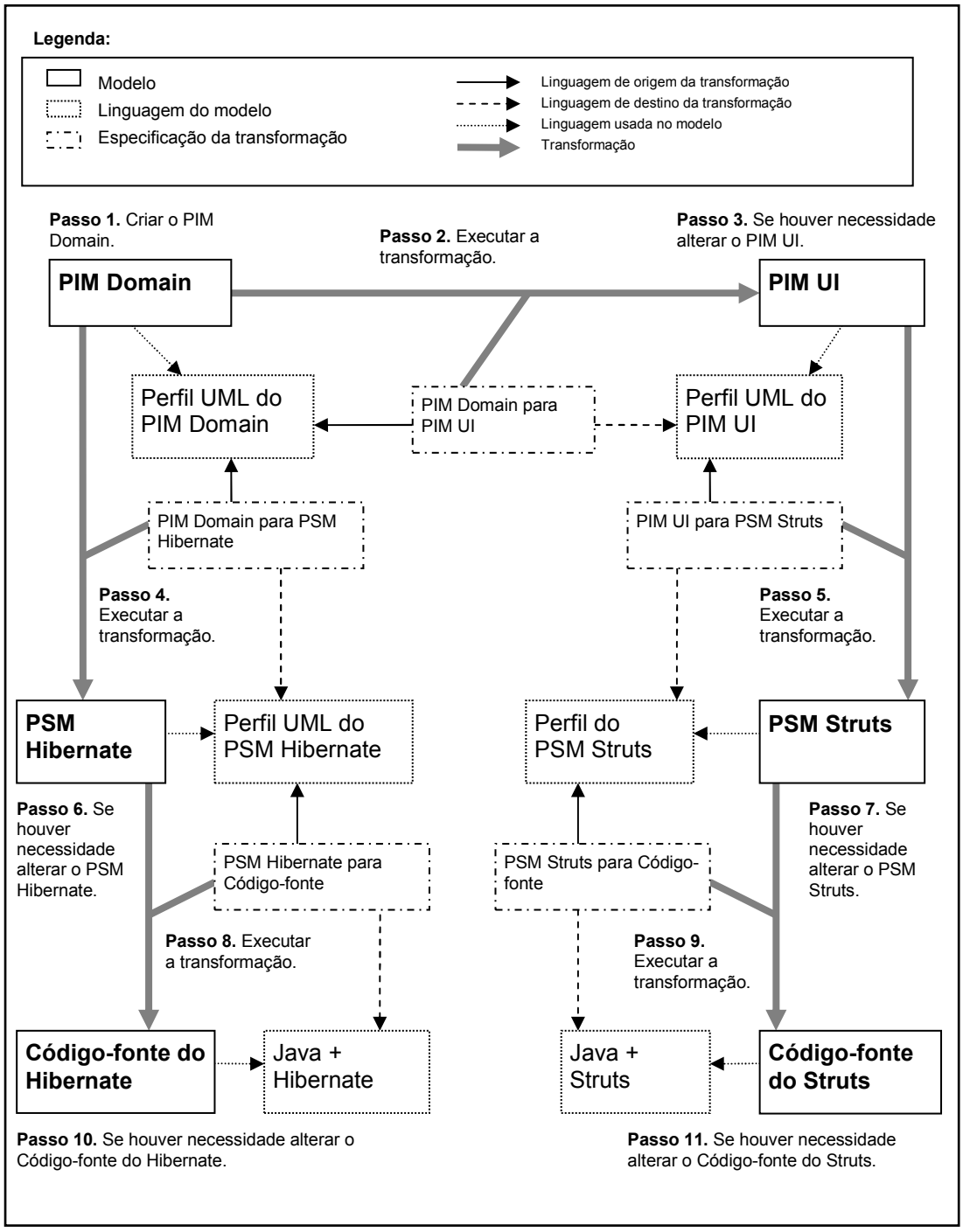


Figura 13 - Transformações, modelos e passos para criação da aplicação.

Usando este processo, foi desenvolvida uma aplicação. Esta aplicação foi contrastada com a mesma aplicação desenvolvida usando o processo Praxis.

Capítulo 4

Resultados e discussão

4.1 Modelos

4.1.1 Modelos independentes de plataforma

No MDA, a modelagem do PIM é onde começa todo o processo de desenvolvimento. Por isso, o primeiro passo foi definir a linguagem de modelagem que usaríamos para o PIM. De acordo com a abordagem MDA, poderíamos usar uma extensão da UML, por meio de perfis, ou definir uma nova linguagem, por meio do MOF. Foi escolhido usar o mecanismo de perfil para adicionar algumas restrições à UML, criando assim a linguagem PIM para este trabalho. Essa escolha se deve ao fato de que, atualmente, a UML é muito difundida e aceita. Além disso, ela suporta muito bem nossa necessidade de expressão.

O PIM foi dividido em dois modelos de tal forma que tenham características bem diferentes e objetivos distintos. O primeiro, denominado **PIM *Domain***, tem o objetivo de modelar o domínio da aplicação ou sistema, ou seja, os conceitos e regras de negócio, sem se preocupar com aspectos de interação com o usuário. O segundo, denominado **PIM UI**, tem o objetivo de modelar o aspecto que não é tratado no primeiro, a interação com o usuário. Este modelo não deverá replicar o que já foi definido no PIM *Domain*, ele simplesmente deverá fazer referência quando necessário. Portanto, ele é dependente do PIM *Domain*.

4.1.1.1 PIM *Domain*

Para representar o PIM *Domain* foi usado um subconjunto da UML 2.0 [46] sem nenhuma extensão. Os elementos de modelagem que formam esse subconjunto estão descritos resumidamente na Tabela 2. Maiores detalhes estão descritos na seção seguinte. Esses aspectos foram escolhidos porque são notações que representam conceitos comumente usados e necessários ao se modelar o domínio de uma aplicação.

Elemento de modelagem	Descrição
Entidade	É definida por uma classe estereotipada e representa os dados persistentes da aplicação e as regras de negócio relacionadas à entidade.
Enumeração	É definida por uma classe estereotipada. Veja a Figura 16. A enumeração é uma classe que representa um conjunto de elementos fixos, em que estes elementos podem ter comportamentos comuns ou específicos.
Associações	Todas as associações devem ter a navegabilidade definida, como mostra a Figura 15. Foi necessário forçar essa obrigatoriedade, pois ao gerar o PSM, pode ser necessário identificar qual é o lado mais forte da associação, mesmo se não estiver modelada como agregação.
Composição e agregação	Utilização da composição e agregação deve ser usada de acordo com suas definições na UML. Veja a Figura 18.
Tipos dos atributos	Todos os atributos deverão ter seus tipos definidos. Os tipos predefinidos pela UML 2.0 são: booleano, inteiro, natural e String. Outros tipos podem ser criados, mas devem ser tipos independentes de plataforma, por exemplo: data e valor monetário.
Obrigatoriedade dos atributos	Todo o atributo é considerado obrigatório, a não ser que sua cardinalidade seja definida como [0..1]. Veja a Figura 19.
Propriedades dos atributos	Os atributos poderão ser assinalados com as seguintes características: - É único ou exclusivo - Somente leitura - Estático - Constante - Valor inicial Veja a Figura 19.

Tabela 2 - Principais elementos de modelagem da UML 2.0 considerados no PIM *Domain*

Conforme especificado pelo MDA, sobre o PIM são executadas uma ou mais transformações que geram um ou mais PSMs. Considerando essa premissa, os elementos descritos na tabela acima foram escolhidos de acordo com os seguintes critérios:

- Primeiro, procurou-se usar os elementos mais conhecidos e utilizados pela comunidade que utiliza a UML para modelar sistemas. Isso permitirá maior aceitação e entendimento do modelo proposto.
- Esse subconjunto deve conter o menor número possível de elementos de tal forma que permita gerar os detalhes necessários no PSM. Isso é importante para que as regras de transformação não tenham complexidade desnecessária, pois para cada elemento e para cada PSM existirá pelo menos uma regra de transformação que o converta em um elemento do PSM.
- Antes de criar qualquer elemento de modelagem, verificamos se na UML 2.0 já existia alguma forma de modelar o que era necessário. Por exemplo: na UML 2.0 existe uma propriedade do atributo de classe para indicar se ele é somente para leitura. Em vez de criar um estereótipo para representar isso, usamos essa propriedade.

A seguir, apresentaremos as justificativas da escolha de cada elemento apresentado na Tabela 2.

A escolha do elemento entidade se deu pelos seguintes fatores. A entidade é usada na atividade de análise de sistemas [24]. A utilização dela é tão difundida que foi incorporada ao

documento de superestrutura da UML [46], que define as construções em nível de usuário necessárias para a UML.

Em várias aplicações, a ocorrência de elementos fixos de uma mesma classe é muito comum. Além isso, para cada elemento fixo pode haver uma regra de negócio associada. Por esse motivo, é importante modelar esses conjuntos de elementos com uma classe especial, que é a enumeração que foi acrescentada à UML na versão 2.0.

A associação, que pode ser classificada como agregação ou composição, é a forma de expressar relacionamentos em classes da UML. Esse é um elemento básico usado na modelagem de classes, por isso, ele não poderia deixar de ser escolhido.

Como o PIM *Domain* tem o objetivo de permitir, a partir dele, gerar o PSM e, posteriormente, o código-fonte da aplicação, é importante definir o tipo de cada atributo das classes. Por definição, os tipos não podem ser específicos de plataforma, mas devem ter um nível de abstração suficiente para permitir mapeá-lo em qualquer tipo específico de tecnologia. A UML 2.0 já tem definidos os tipos primitivos: *string*, inteiro, natural e booleano. Entretanto, se for necessário, pode-se criar novos tipos que fazem sentido para a aplicação, por exemplo: dinheiro, data, CPF e outros. A especificação da UML 2.0 permite a criação de tipos personalizados, portanto, esse elemento da UML 2.0 já atende à nossa necessidade.

A necessidade de especificar a obrigatoriedade de atributos de uma classe é uma questão que está presente na maioria das aplicações. Pelo mecanismo de cardinalidade da UML, é possível especificar se um atributo é ou não obrigatório, mesmo se ele não estiver representando um lado de uma associação. Por isso, esse mecanismo também foi escolhido para fazer parte do PIM *Domain*.

A UML 2.0 define algumas propriedades para os atributos de classe, entre elas foram escolhidas: é único ou exclusivo, somente leitura, estático, constante, valor inicial. Essa escolha também foi feita porque pelo uso dessas propriedades é possível representar alguns comportamentos comuns em aplicações de forma geral e que têm a necessidade de serem especificados. Por exemplo, dizer que o atributo é único, indica que não pode existir mais de um objeto com o mesmo valor dele. Outro exemplo: informar o valor inicial de um atributo significa que, ao criar um objeto, o atributo já será preenchido com esse valor inicial. Na implementação final, gerada pelas transformações, a tela de criação do objeto poderá trazer o campo que representa a atributo preenchido com esse valor.

A fim de mostrar como o PIM *Domain* pode ser utilizado, na Figura 14 apresentamos um exemplo de como ficaria o PIM *Domain* do sistema Merci, que foi especificado para fins educativos do Praxis [20]. Este sistema foi escolhido por apresentar uma especificação definida em todas as etapas de desenvolvimento. Sua especificação completa pode ser encontrada no endereço eletrônico “http://www.wppf.uaivip.com.br/praxis/2.1/Praxis 2.1.htm”. O Merci tem como missão o apoio informatizado ao controle de vendas, de compras, de fornecedores e de

estoque de uma mercearia específica. As principais funções do sistema relacionadas ao diagrama da Figura 14 são: controle de usuários que terão acesso ao Merci, representado na figura pelas classes *Usuário* e *Grupo de usuários*; controle manual de entrada e saída de mercadorias (estoque), representado pela classe *Mercadoria*; inclusão, exclusão e alteração de mercadorias, também representado pela classe *Mercadoria*; inclusão, exclusão e alteração de fornecedores, representado pela classe *Fornecedor*; inclusão, exclusão e alteração de pedidos de compra, representado pelas classes *Pedido de Compra*, *Status*, *Item de compra*, *Item de mercadoria*.

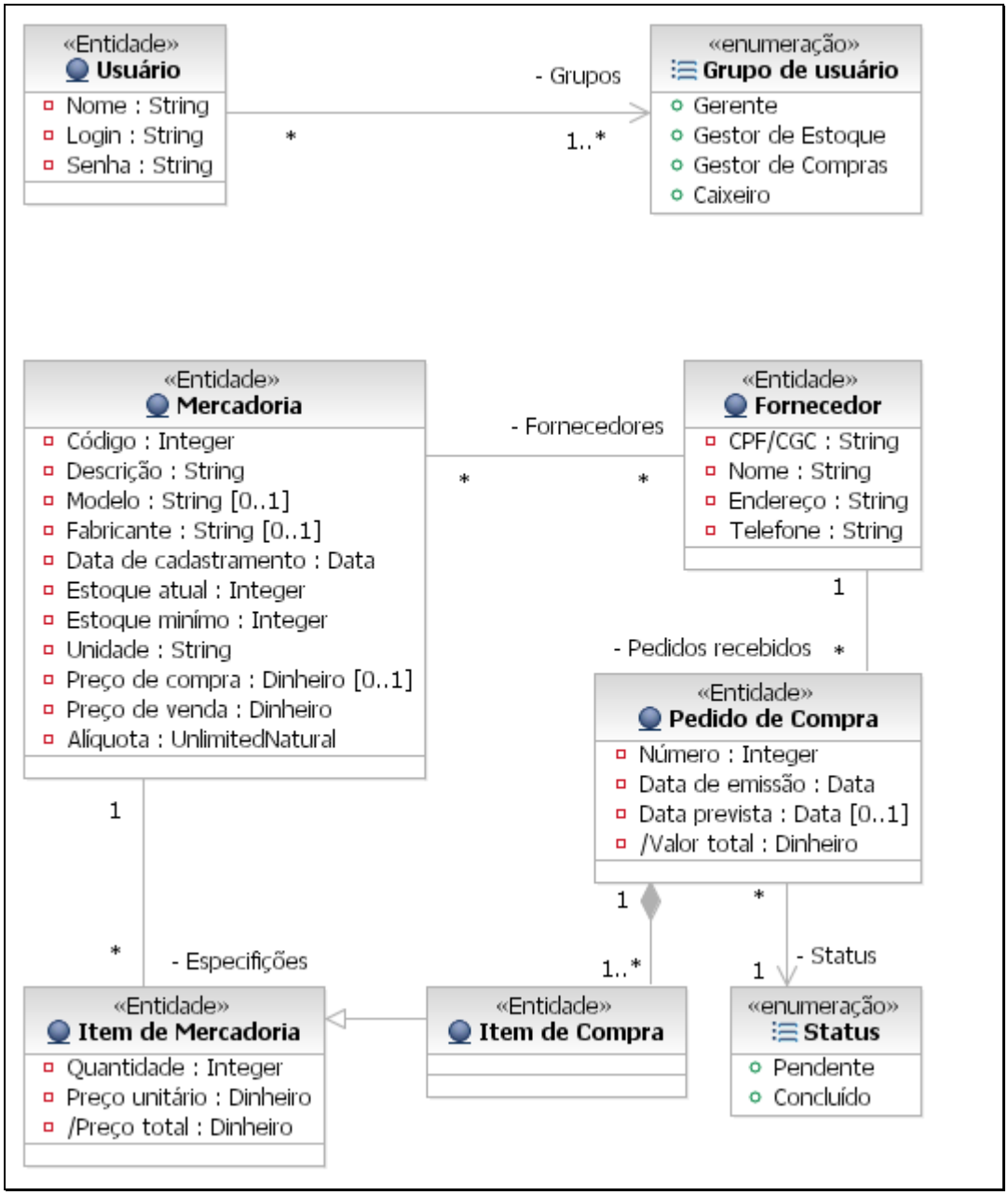


Figura 14 - Diagrama de exemplo do PIM *Domain* referente ao sistema Merci.

Como pode ser observado, o diagrama da Figura 14 não tem nenhum detalhe específico de alguma tecnologia ou plataforma. Entretanto, como será explicado detalhadamente nas seções seguintes, é possível, a partir desse modelo, gerar modelos com detalhes específicos de tecnologia e, posteriormente, gerar código-fonte funcional.

4.1.1.1.1 Utilização dos elementos de modelagem

Esta seção tem o objetivo de mostrar como os elementos de modelagem do PIM *Domain* podem ser utilizados para permitir a realização das transformações. Serão apresentadas as seguintes situações de uso desses elementos:

- Navegabilidade de associações;
- Enumerações;
- Herança;
- Tipos de associações;
- Configuração de propriedades de atributo de classe.

Estas são as situações em que as transformações estão preparadas para entender e, portanto, seguindo as regras descritas nesta seção, o modelo será considerado bem-formado e as transformações funcionarão corretamente.

No diagrama da Figura 15, são mostradas as três possíveis direções de navegabilidade das associações, conforme especificado na UML. As transformações suportam tanto navegabilidade bidirecional quanto unidirecional. A associação entre as classes *Classe associação bidirecional* e *Classe raiz* representa a navegabilidade bidirecional. A associação entre as classes *Classe associação não navegável* e *Classe raiz* e entre *Classe associação navegável* e *Classe raiz* representa a navegabilidade unidirecional. Essa situação de utilização é apresentada para mostrar que a navegabilidade pode ser usada na modelagem.

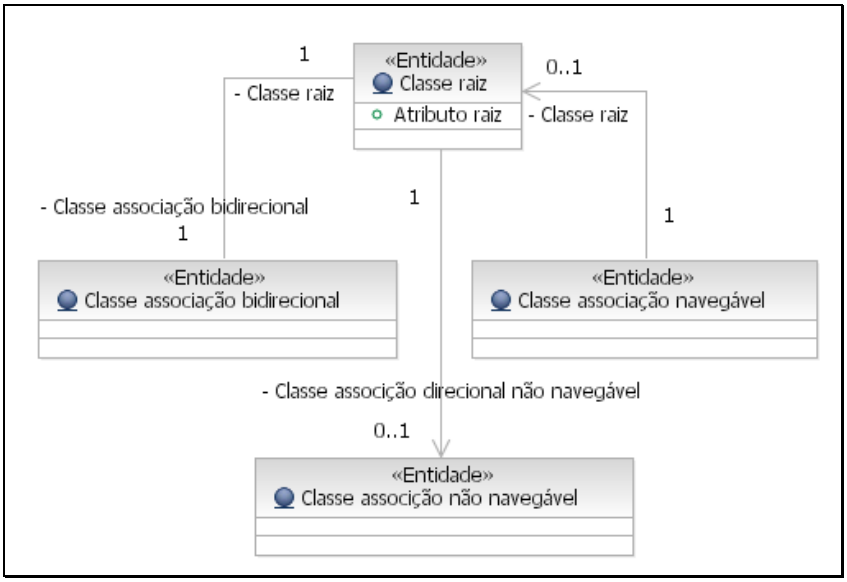


Figura 15 - Navegabilidade de associações no PIM *Domain*

No diagrama da Figura 16, mostramos como as enumerações poderiam ser modeladas. Consideramos que as enumerações sempre terão associação com a navegabilidade unidirecional. Essa restrição foi imposta já que atende aos casos comumente utilizados. No momento, não estamos preocupados em criar transformações para casos menos utilizados. O que pode variar é a cardinalidade da associação. Do lado da enumeração, consideramos três possibilidades: zero ou um (0..1), um (1) ou muitos (*). Do lado da entidade (Classe raiz) consideramos sempre a cardinalidade de *muitos* (*), essa prescrição foi feita pelo fato de a enumeração representar valores fixos e, geralmente, estes valores podem estar associados a várias instâncias. Estas situações de utilização de enumerações foram apresentadas para destacar o fato de que a enumeração só poderá estar associada à entidade e só poderá ser utilizada a navegabilidade e as cardinalidades apresentadas a seguir.

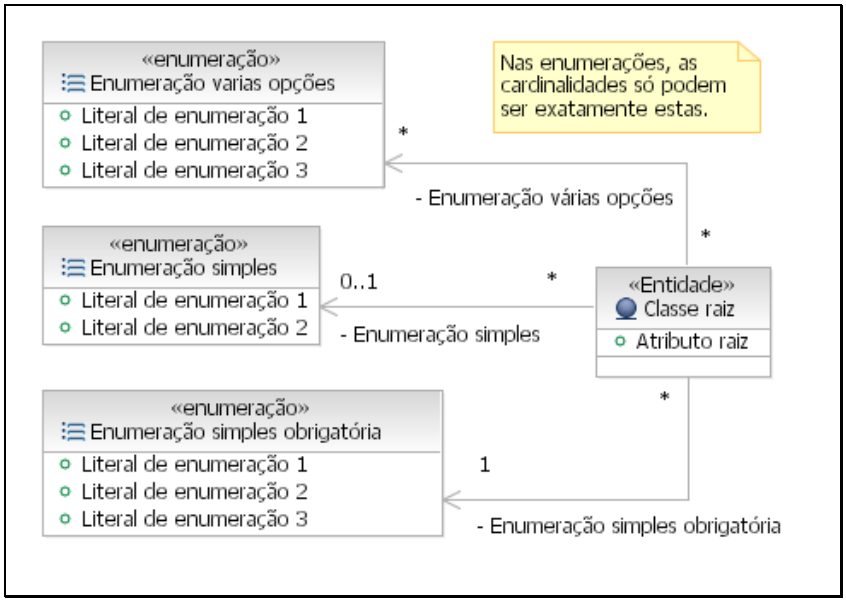


Figura 16 - Formas de utilização da enumeração no PIM *Domain*

Com relação à herança, foi considerada apenas a herança simples, sendo possível vários níveis. O diagrama da Figura 17 mostra um exemplo simples de herança. Essa situação foi definida para restringir o uso da herança múltipla e para mostrar que classes abstratas podem ser usadas, mas deve existir pelo menos uma classe concreta herdando da classe abstrata.

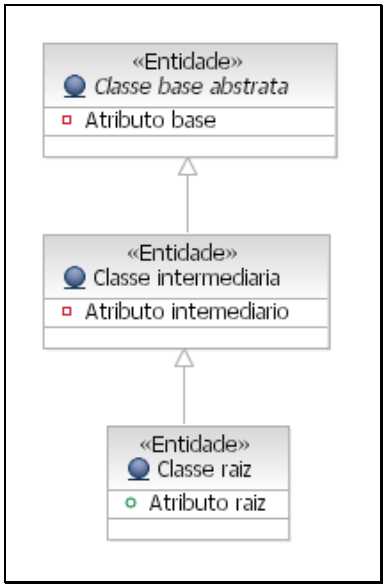


Figura 17 - Herança no PIM *Domain*

O diagrama da Figura 18 mostra os quatro tipos de associações que devem ser suportados pelas transformações: composição, agregação, simples e composta (ternária). Na transformação de PIM *Domain* para PSM *Hibernate*, cada um desses tipos é criado no PSM

Hibernate com algumas propriedades e comportamentos específicos que atendem às suas semânticas. Por exemplo: conforme definido na UML, a composição faz com que o ciclo de vida do objeto representando o lado parte da associação tenha seu ciclo de vida associado ao objeto representando o lado todo. Portando, se o objeto que representa o todo é excluído, o objeto composto, representando o lado parte, também o será. A entidade composta também é considerada parte do objeto todo, sua existência não faz sentido se não estiver associada ao objeto todo. Por isso, se o objeto composto for desassociado, automaticamente o objeto composto será excluído.

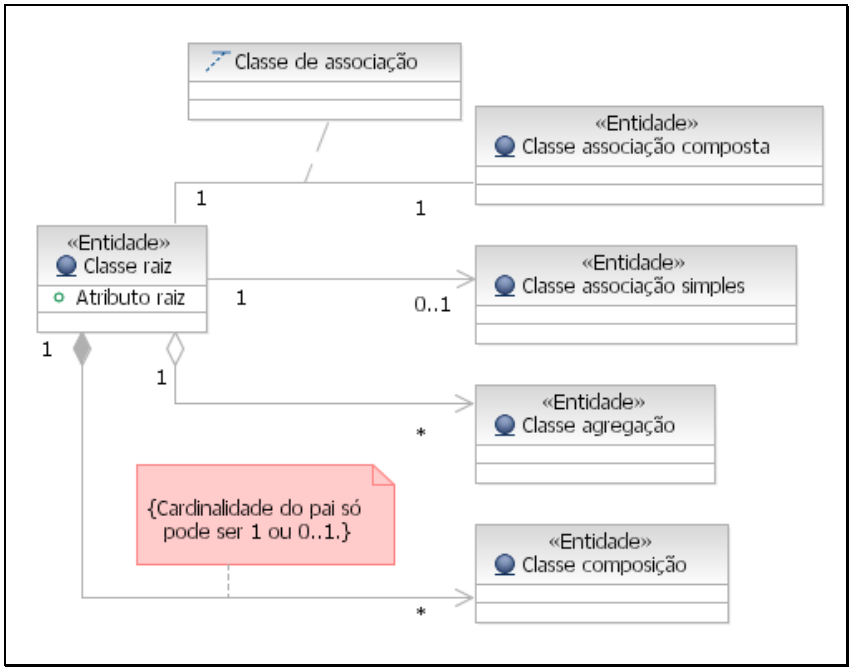


Figura 18 - Tipos de associações no PIM *Domain*

Para cada propriedade de uma classe é possível configurar se ela á estática, constante, única ou não-obrigatória ([0..1]). Além disso, é possível também definir o tipo da propriedade e o valor inicial, se houver. A Figura 19 ilustra como algumas dessas configurações são exibidas graficamente no diagrama.

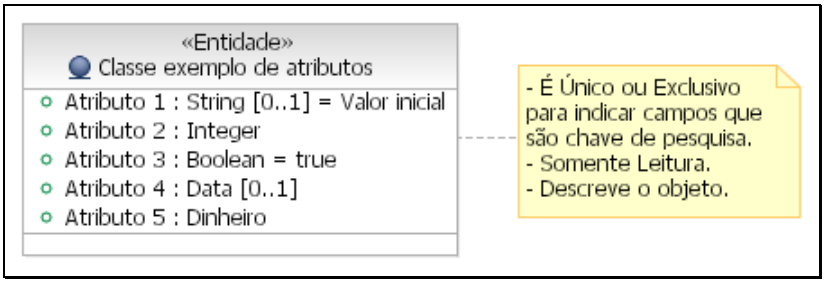


Figura 19 - Configurações das propriedades no PIM *Domain*

A partir de um PIM *Domain* que contém as informações como as ilustradas é possível gerar, por meio de transformações específicas de plataforma, PSMs bem detalhados que já contêm várias regras de domínio simples e comuns. Essas transformações poupam grande quantidade de trabalho manual, que é propenso a erros simplesmente pelo fato de ser feito manualmente.

4.1.1.2 PIM UI

A definição do PIM UI foi mais complexa, pois a utilização pura da UML 2.0 não é suficiente para modelar a interface com o usuário. A seguir é descrito o Perfil do PIM UI e as origens e justificativas de cada elemento desse modelo. Conforme explicado anteriormente, o modelo PIM UI é gerado a partir do PIM *Domain* e pode ser alterado de acordo com as necessidades dos requisitos e de usabilidade.

4.1.1.2.1 Perfil PIM UI

A Tabela 3 apresenta os estereótipos que foram necessários para compor o Perfil PIM UI.

Estereótipo	Aplicável a	Descrição	Restrições	Origem: Justificativa
Apresentador	Classe	Uma abstração de alto nível de tela ou partes dela.	-	ETP: equivalente ao Espaço de interação do Wisdom, ao *FreeContainer* e ao *Contêiner* da UMLi.
Comando	Classe	Uma abstração de alto nível de ação iniciada pelo usuário.	Deve ser suportado em pelo menos uma tela e pode gerar navegação para outras telas.	UMLi: na UMLi esse estereótipo é chamado de *ActionInvokers*. O Wisdom tem um estereótipo chamado Ação que representa o mesmo conceito, só que como uma operação de uma classe. O ETP tem o estereótipo Tarefa que é representado de forma semelhante ao *ActionInvokers* da UMLi, a diferença é que no ETP ele é uma abstração de mais alto nível que pode representar uma seqüência de ações do usuário.
Visão	Associação	Indica que o apresentador é uma visão de certa entidade, ou seja, indica qual entidade está sendo manipulada pelo apresentador.	A associação deve ser entre Apresentador e Entidade, navegável somente do apresentador para a entidade e a associação deve ser simples.	ETP: no ETP existe o conceito de associação de visão, mas ela não é identificada explicitamente por um estereótipo. Para explicitar esse tipo de associação e facilitar a automação nas transformações foi necessário criar este estereótipo.
Contém	Associação de composição	Dado um apresentador, informa quais são os outros apresentadores que o compõem.	A associação deve ser entre Apresentador e Apresentador, navegabilidade unidirecional e a associação deverá ser de composição.	Wisdom: este conceito também está presente na UMLi, em que é representado simplesmente pela composição da UML. Para explicitar esse tipo de associação e facilitar a automação nas transformações foi necessário manter este estereótipo.
Suporta	Associação de composição	Indica quais comandos determinado apresentador suporta.	A associação deverá ser entre Apresentador e Comando, navegável somente de Apresentador para Comando e o Apresentador deve ser composto de Comandos.	ETP: no ETP existe este conceito, mas ele não é identificado explicitamente por um estereótipo. Para explicitar esse tipo de associação e facilitar a automação nas transformações, foi necessário criar este estereótipo.

Estereótipo	Aplicável a	Descrição	Restrições	Origem: Justificativa
Navega	Associação de agregação	Indica uma possível navegação, ou seja, possível mudança de Apresentador, após a execução de um comando.	A associação deverá ser entre Comando e Apresentador ou entre Comando e Comando, navegável somente do Comando para o Apresentador e o Comando deve ser agregado de Apresentador.	ETP e Wisdom: no ETP existe este conceito, mas ele não é identificado explicitamente por um estereótipo. No Wisdom existe este estereótipo, mas ele é usado para mostrar as possíveis navegações entre telas que é uma modelagem de mais alto nível do que no ETP que explicita qual tarefa leva para qual tarefa. Para explicitar esse tipo de associação e viabilizar a automação nas transformações, foi necessário manter este estereótipo, mostrando que um comando pode gerar uma navegação para um apresentador ou para um outro comando que irá navegar para um apresentador. A navegação entre dois comandos permite a reutilização de comandos que tem como funcionalidade a recuperação ou processamento de dados que serão exibidos em um apresentador.
Campo editável Campo não editável	Texto de nota	Indica que um atributo será exposto a um usuário pelo Apresentador a que a nota está ligada. O usuário poderá alterar o valor do atributo se editável. Se não editável, o valor não poderá ser alterado.	Deverá ser aplicado no início de qualquer linha do texto de uma nota que está ligada a um apresentador. A linha em que o estereótipo é aplicado deverá ter um texto que representa uma referência a um atributo de uma entidade. Esta entidade deverá ser a entidade ligada ao apresentador da nota. Poderão ser feitas referências indiretamente a outras entidades, por exemplo, “Item de Mercadoria.Mercadoria. Código”.	ETP, Wisdom e UMLi: no ETP é sugerido colocar em uma nota os atributos da entidade que devem ser exibidos, mas não diz nada em relação ao campo poder ser alterado ou não. No Wisdom e na UMLi já existem estereótipos para dizer se o campo pode ser ou não alterado. Só que no Wisdom o estereótipo é aplicável a atributos de classe e na UMLi, as classes representam os campos. Nesse trabalho, optamos por unir as três notações usando estereótipos dentro das notas. Optamos por usar a nota, pois ela é mais flexível, permitindo referenciar o caminho até o atributo e, mesmo nesse caso, é possível visualizar essa informação diretamente no diagrama, pois a nota pode ser exibida.

Tabela 3 - Estereótipos do perfil PIM UI

Além dos estereótipos da Tabela 3, o PIM UI contém os estereótipos descritos na Tabela 4. Estes são específicos de comandos e representam ações comuns em vários sistemas ou aplicações. Todos estes estereótipos são aplicáveis somente a classes que já são do estereótipo comando. Eles permitem que a implementação dos comandos que representam seja automatizada. Além disso, esses comandos serão portáveis entre plataformas. Isso é possível porque para cada um desses estereótipos podem ser criadas regras de transformação para cada plataforma que se deseja atender. Dessa forma, poderá ser gerado o PSM para cada uma. A partir do PSM será possível gerar o código-fonte com o comportamento que o estereótipo representa. As classes com os estereótipos representados na Tabela 4 sempre estarão associadas a um apresentador. Este, por sua vez, poderá estar associado a uma entidade. Cada uma dessas classes com o estereótipo comando poderá representar operações sobre a entidade associada.

Estereótipo	Descrição
Excluir	Exclui permanentemente a entidade e o apresentador que representa a entidade.
Inserir	Cria uma entidade em memória e vai para o apresentador, que permite informar os dados dessa nova entidade.
Alterar	Vai para o apresentador, que permite editar os dados da entidade associada à ação de alteração.
Visualizar	Vai para o apresentador, que permite visualizar os dados da entidade associada à ação de visualização.
Voltar	Volta para o apresentador, que estava ativo antes de ativar o apresentador corrente.
Salvar	Salva a entidade em questão. Se houver alguma inconsistência de regra de negócio, é exibida uma mensagem explicativa para que o usuário passa corrigi-la e tentar salvar novamente.
Cancelar	Cancela as alterações realizadas no apresentador corrente.
Inserir detalhe	Cria uma entidade em memória e adiciona ao apresentador pai um apresentador que representa a nova entidade para que o usuário possa preencher os campos da entidade.
Excluir detalhe	Exclui em memória a entidade e o apresentador que representa a entidade.
Pesquisar	Em um apresentador que apresenta várias instâncias de uma entidade, essa ação permite filtrar as instâncias a serem exibidas. O filtro é feito pelos campos do apresentador aos quais esta ação está associada.
Exibir todos	Se o Pesquisador foi acionado, a ação Exibir todos permite cancelar o filtro e exibir novamente todas as instâncias da entidade.

Tabela 4 - Estereótipos de comandos do PIM UI

As ações da tabela são básicas de um sistema de cadastro. O intuito é que sejam acrescentadas mais comandos à medida em que ações comuns forem detectadas para determinados tipos de aplicações.

A seguir, são apresentados alguns diagramas que exemplificam a utilização do PIM UI. Como exemplo, continuaremos a usar o Merci, apresentado na seção 4.1.1.1. O próximo passo é gerar o PIM UI pela transformação de PIM *Domain* para PIM UI. Para simplificar o exemplo, explicaremos apenas as classes do PIM UI geradas pela transformação aplicada sobre as classes Usuário e Grupo de usuário PIM *Domain* da Figura 14. Estas classes são apresentadas separadamente na Figura 20. O resultado da execução desta transformação será a modelagem independente de plataforma das interfaces com o usuário para a funcionalidade de controle de usuários que terão acesso ao Merci. Esta funcionalidade consiste em incluir, alterar e excluir usuários do sistema. A seguir, são apresentados os diagramas da modelagem gerada pela transformação.

As telas apresentadas na Figura 21, na Figura 23 e na Figura 25 são o resultado final, depois da execução de todas as transformações. Elas são apresentadas a partir dessa seção para facilitar o entendimento dos modelos que permitirão sua geração ao executar a última transformação.

A transformação de PIM *Domain* em PIM UI consiste em criar interfaces com o usuário que permitam a realização de algumas operações básicas sobre as entidades que são alvo da transformação. Estas operações são: 1) criar instâncias da entidade; 2) alterar os atributos de instâncias já existentes; 3) excluir instâncias já existentes; e 4) executar pesquisas que permitam encontrar instâncias já existentes. Essa pesquisa funciona da seguinte forma: o usuário informa os

campos de pesquisa que são referentes a atributos da entidade e o sistema encontra todas as entidades cujos valores dos atributos contenham o valores informados pelo usuário.

A Figura 20 contém apenas uma entidade, o *Usuário*, portanto as operações básicas que o PIM UI irá representar serão sobre ela. A associação entre *Usuário* e *Grupo de usuário* será considerada como um atributo da entidade *Usuário*, em que seu valor poderá ser qualquer combinação dos elementos de enumeração do *Grupo de usuário*: *Gerente*, *Gestor de Estoque*, *Gestor de Compras* e *Caixeiro*.

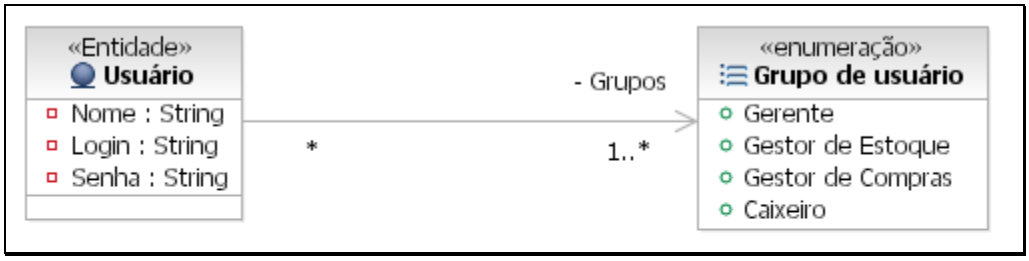


Figura 20 - PIM *Domain* da classe de entidade Usuário do sistema Merci

Para facilitar o entendimento do PIM UI, primeiro será apresentada a imagem da tela implementada e, logo em seguida, a modelagem que representa a tela.

A primeira é a tela de Gestão de usuário apresentada na Figura 21. Ela possui comandos que permitem exibir todos os usuários, excluir um específico e pesquisar pelo nome ou login do usuário. Essa tela também possui comandos que permitem criar, visualizar ou alterar usuários, mas estes comandos apenas abrem outra tela onde estas tarefas são efetivadas.

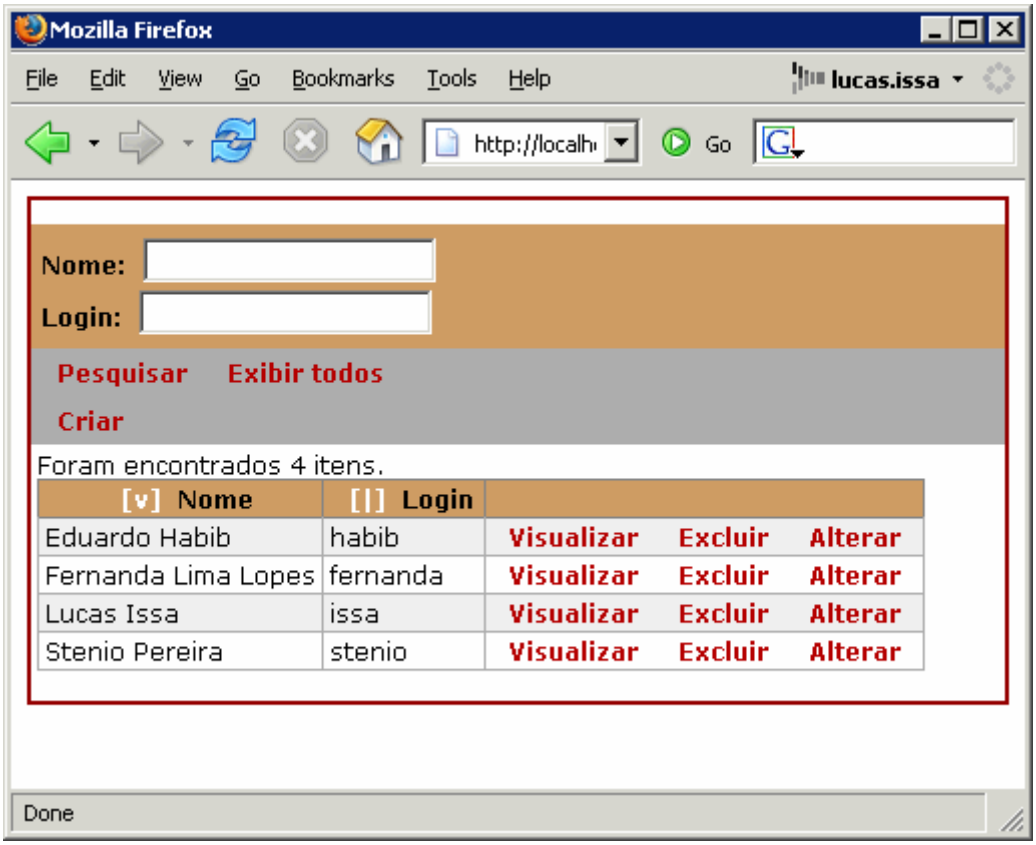


Figura 21 - Tela de gestão de usuário

O diagrama da Figura 22 apresenta a modelagem da tela apresentada na Figura 21. Essa tela é representada pelas classes *Pesquisa de usuário*, *Gestão de usuário* e *Descritor de usuário*. É possível identificá-la, pois essas classes estão com o estereótipo <<Apresentador>> e estão associadas por uma composição com o estereótipo <<Contém>>. A composição entre as classes *Gestão de usuário* e *Descritor de usuário* tem a cardinalidade zero ou muitos (*) do lado da última classe. Isso caracteriza a listagem de informações sobre vários usuários. As informações listadas são *Nome* e *Login* do *Usuário*, que estão representadas pela nota ligada ao *Descritor de usuário*. A entidade *Usuário* é diretamente identificada pela associação que tem o estereótipo <<Visão>>. Para cada usuário listado é possível alterar ou visualizar suas informações ou excluí-lo. Este comportamento é representado respectivamente pelas classes com os estereótipos <<Comando, Alterar>>, <<Visualizar, Comando >> e <<Excluir, Comando>>. Destes comandos, o primeiro vai para tela de *Edição de usuário*, o segundo vai para tela de *Visualização de usuário* e o terceiro permanece na mesma tela. A classe com os estereótipos <<Inserir, Comando>> representa o comportamento de criação de uma instância da entidade *Usuário* que também vai para a tela de *Edição de usuário*. A pesquisa é representada pela classe *Pesquisa de usuário* onde é possível entrar com o *Nome* ou *Login* do usuário. Essa entrada de dados é modelada pela nota ligada à classe e pela associação com o estereótipo <<Visão>>, indicando a qual entidade as entradas se referem. A execução da pesquisa é representada pela classe com os estereótipos <<Pesquisar, Comando>> que, ao executar a pesquisa, permanece na mesma tela, mas exibe

apenas as entidades dos usuários que contiverem os valores informados. Por fim, a classe com os estereótipos <<Exibir todos, Comando>> representa o cancelamento da pesquisa, voltando a exibir todas as entidades dos usuários.

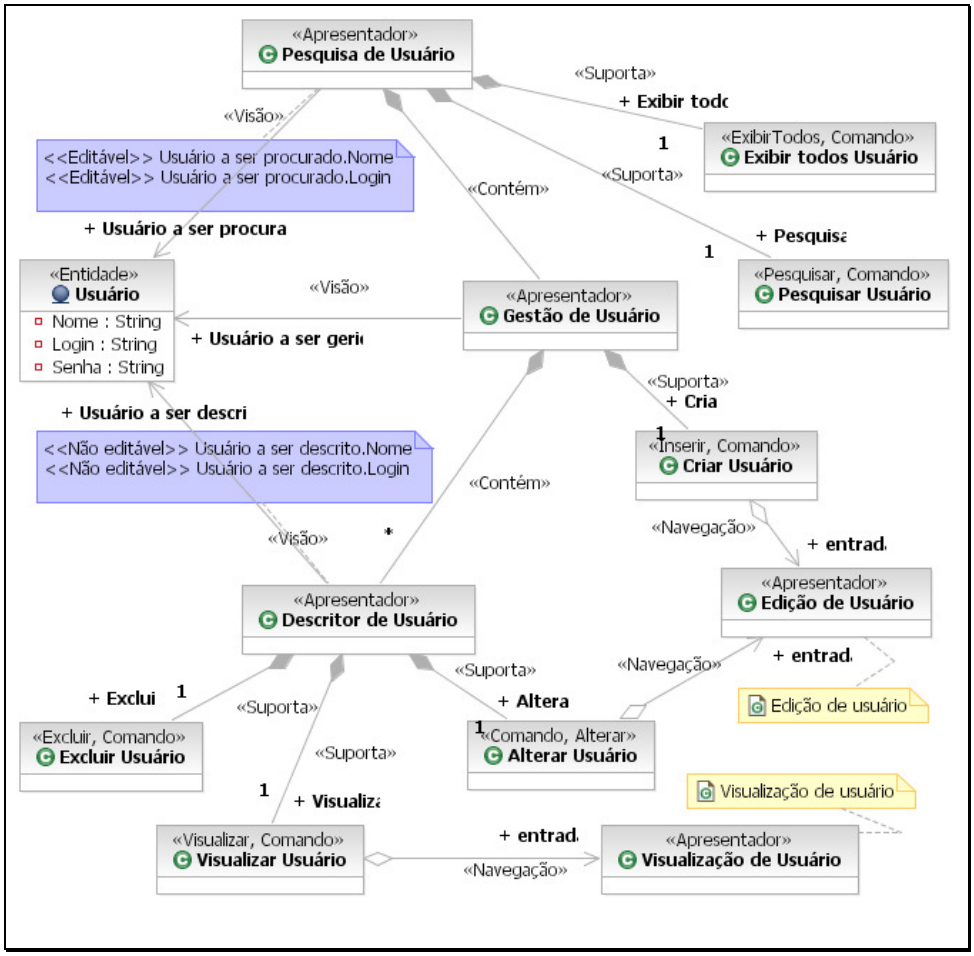


Figura 22 - Diagrama do PIM UI de Gestão de usuário

A próxima interface é a *Tela de visualização de usuário* apresentada na Figura 23. Ela permite que as informações da entidade *Usuário* sejam visualizadas. Essa tela é exibida pelo acionamento do comando visualizar da tela de gestão de usuário e tem dois comandos: o alterar, que exibe outra tela para editar os dados, e o voltar, que retorna a para tela de gestão.

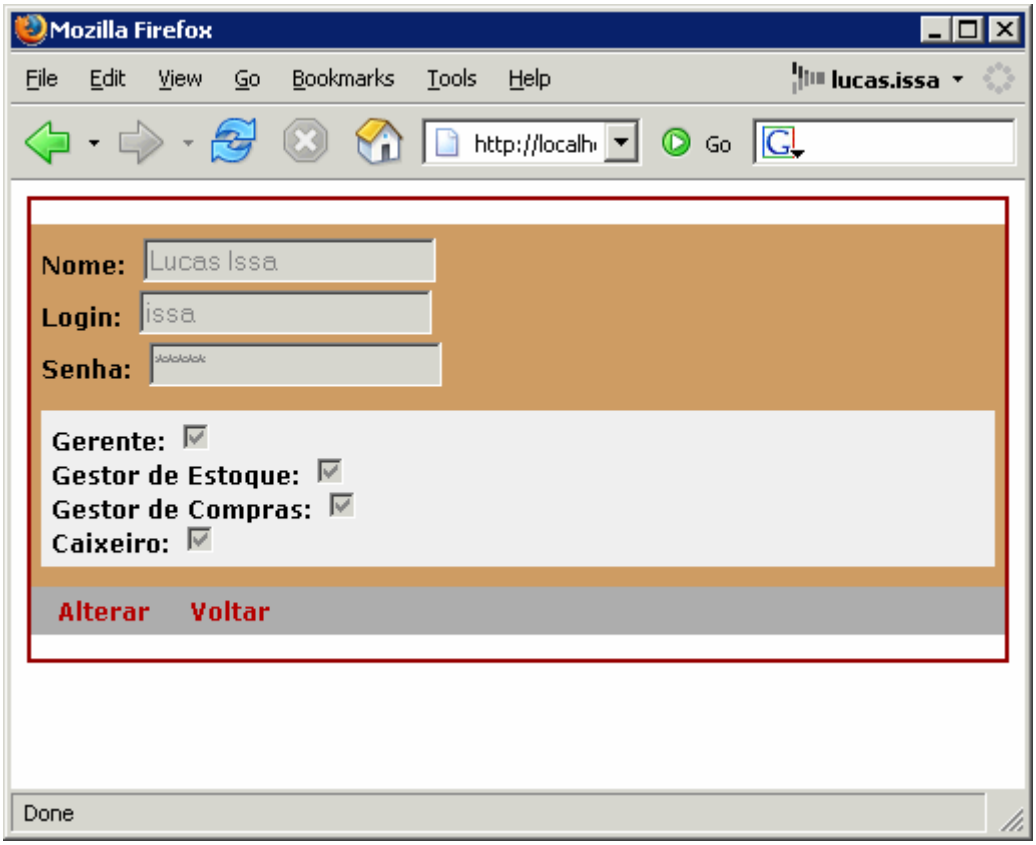


Figura 23 - Tela de visualização de usuário

O diagrama da Figura 24 apresenta a modelagem da tela apresentada na Figura 23. Essa tela é representada pela classe *Visualização de usuário*. É possível identificar que essa classe representa uma tela, pois está com o estereótipo <<Apresentador>>. As informações que serão visualizadas são representadas pela nota ligada à classe *Visualização de usuário*. Nesta nota, estão os atributos da entidade *Usuário* que deverão ser exibidos. A partir do apresentador *Visualização de usuário* é possível mudar para tela de alteração desses dados ou voltar para tela de gestão exibindo todos os usuários. Este comportamento é representado pelas classes com os estereótipos <<Alterar, Comando>> e <<Voltar, Comando>>.

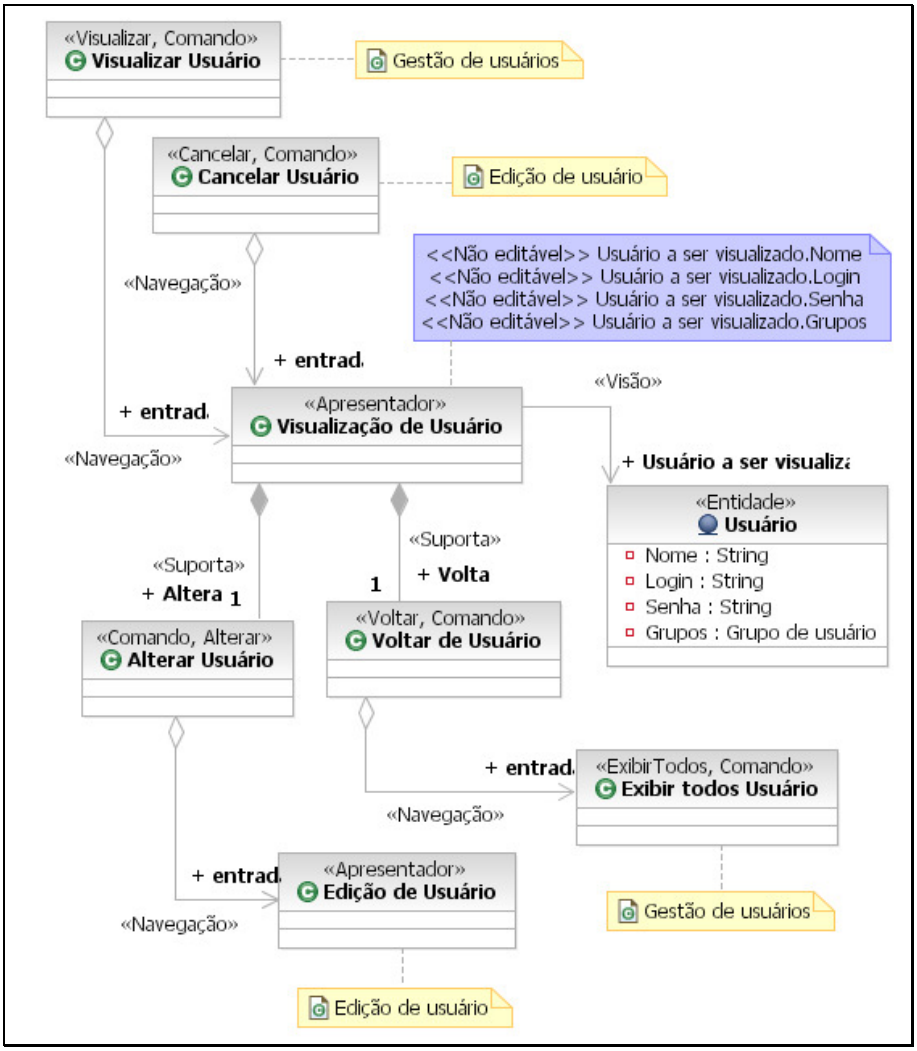


Figura 24 - Diagrama do PIM UI de Visualização de usuário

A última interface é a *Tela de edição de usuário* apresentada na Figura 25. Ela permite inserir ou alterar as informações da entidade *Usuário*. Note que nela os campos estão habilitados, enquanto na tela de visualização os campos estão desabilitados. Essa tela é exibida pelo acionamento do comando criar ou alterar da tela de gestão de usuário. Ela tem dois comandos: o salvar, que mantém os dados inseridos ou alterados, e o comando cancelar, que descarta os dados informados pelo usuário. Os dois comandos vão para a tela de visualização se executados com sucesso.

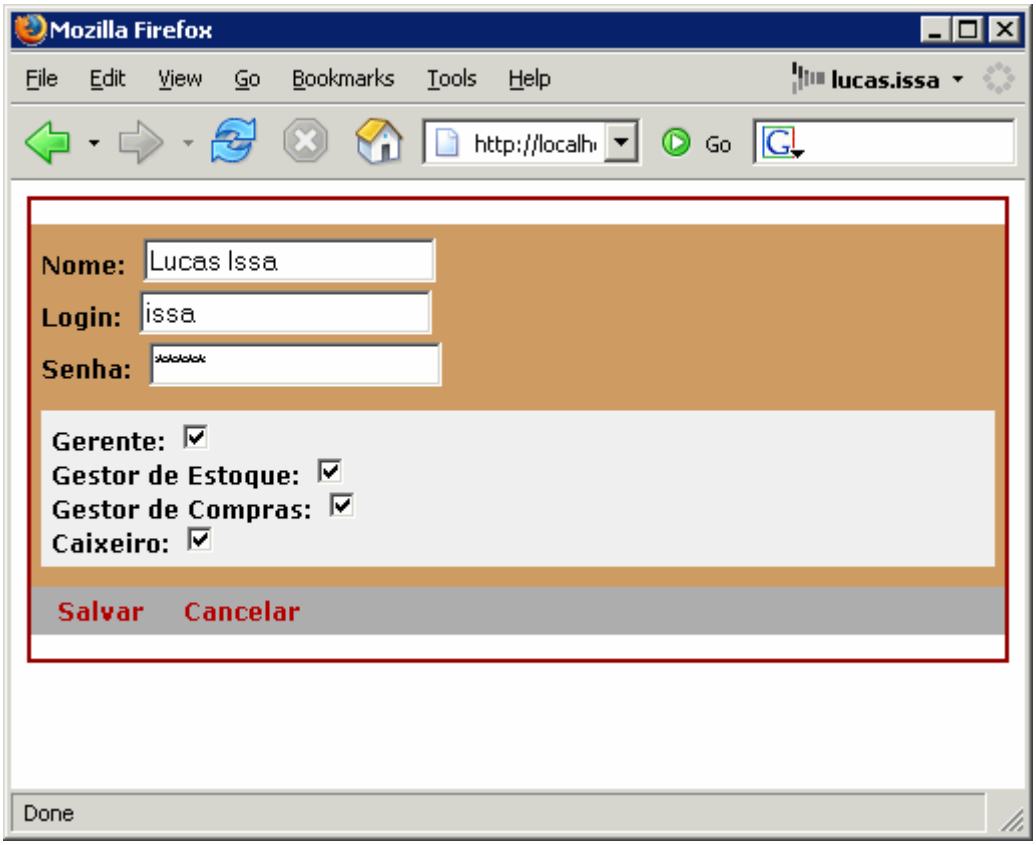


Figura 25 - Tela de edição de usuário

O diagrama da Figura 25 apresenta a modelagem da tela apresentada na Figura 26. Ela é semelhante à representada no diagrama da Figura 24. As diferenças são que os campos representados pela nota são editáveis e as tarefas da tela são Salvar e Cancelar os dados alterados da entidade *Usuário*. Estas tarefas são representadas respectivamente pelas classes com os estereótipos <<Salvar, Comando>> e <<Cancelar, Comando>>. Nesse diagrama, o comando *Cancelar* tem uma peculiaridade: seu acionamento pode gerar uma navegação para lugares diferentes. Se a tela está representando um usuário que ainda não foi criado, o *Cancelar* gera uma navegação para a tela de gestão exibindo todos os usuários. Se a tela está representando um usuário que já foi criado, o *Cancelar* gera uma navegação para tela de visualização do usuário. O primeiro caso ocorre quando a tela é acionada pelo comando *Criar usuário* da tela de gestão e o segundo caso, quando a tela é acionada pelo comando *Alterar* da tela de gestão ou da tela de visualização.

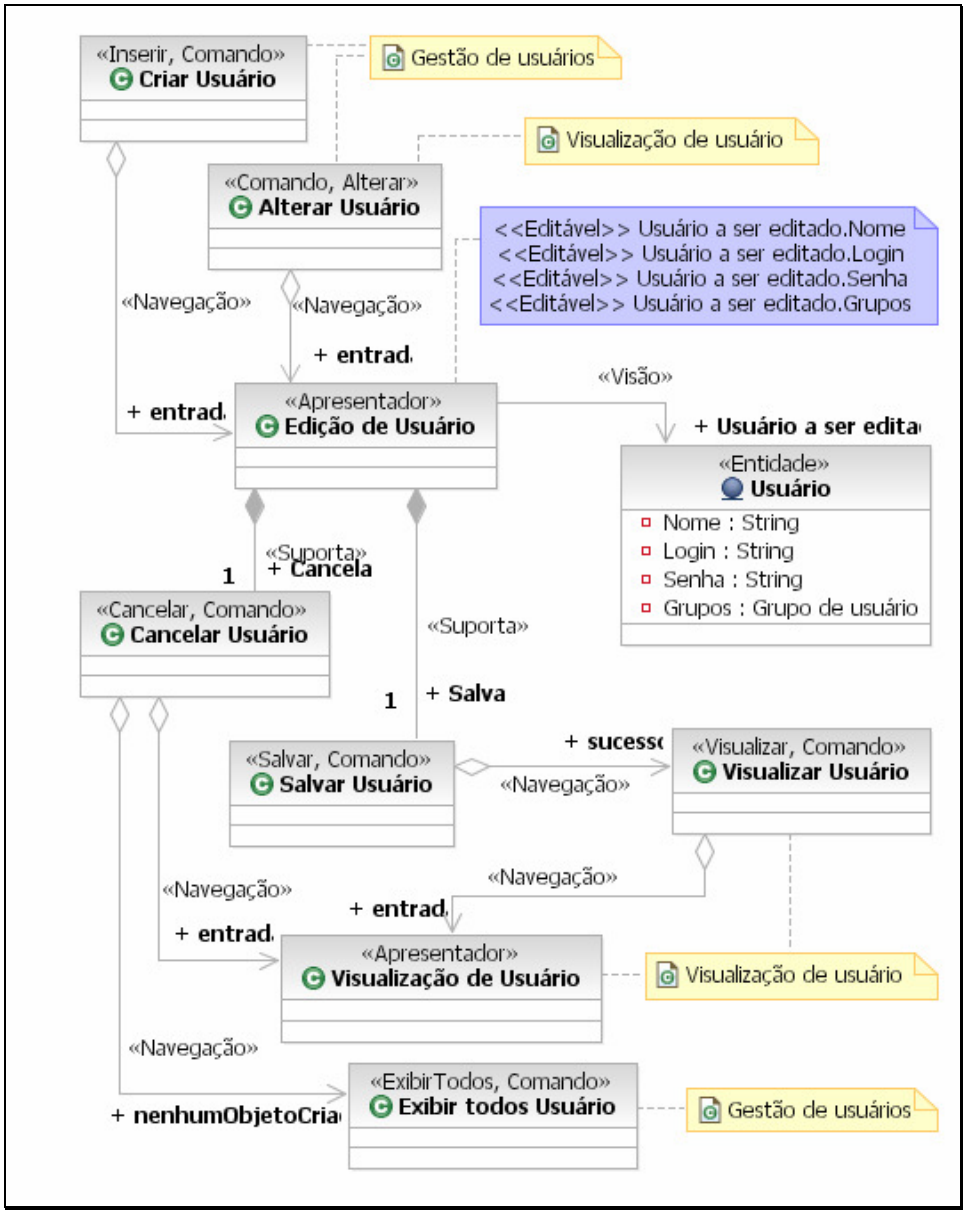


Figura 26 - Diagrama do PIM UI de Edição de usuário

Com esse último diagrama, a modelagem gerada pela transformação de PIM *Domain* para PIM UI sobre as classes *Usuário* e *Grupo de usuário* está completa. Essa modelagem poderia ser modificada para atender requisitos específicos necessários para o sistema. No caso da funcionalidade de controle de usuário do Merci, essa modelagem atende completamente. Na verdade, ela vai até um pouco além do que é necessário. De acordo com a especificação do Merci, não seria necessária a tela de visualização e nem o comando de pesquisa na tela de gestão. Neste caso, bastaria remover os elementos da modelagem que representam esses comportamentos. No

caso desse exemplo, não seria necessário criar novos elementos de modelagem, mas isso poderia ser feito se a especificação do sistema levasse a essa necessidade.

4.1.2 Modelos dependentes de plataforma

Para atingir o objetivo do trabalho, também foi necessário definir a linguagem de modelagem para o PSM que usamos. Da mesma forma que o PIM, poderíamos usar uma extensão da UML por meio de perfis ou definir uma nova linguagem pelo MOF para fazer essa definição. Para o PSM, também foi escolhido usar o mecanismo de perfil para adicionar algumas restrições e extensões à UML, criando assim a linguagem PSM para este trabalho.

A linguagem PSM também foi separada em duas, seguindo o mesmo critério usado para separar o PIM. A primeira linguagem, o **PSM Hibernate**, tem o objetivo de modelar o domínio da aplicação ou sistema. A segunda linguagem, o **PSM Struts**, tem o objetivo de modelar a interface com o usuário.

A divisão desses modelos também segue o padrão MVC (*Model View Controler*) [29]. Esse é um padrão de arquitetura de aplicações que visa a separar a lógica da aplicação (Modelo) da interface do usuário (Visão) e do fluxo da aplicação (Controlador), permitindo que a mesma lógica de negócios possa ser acessada e visualizada por várias interfaces. Esse padrão é compatível com a divisão domínio e interface com o usuário. Para implementar a camada Modelo do MVC, será usada a tecnologia *Hibernate*. Para implementar as camadas Controlador e Visão, será usada a tecnologia *Struts*.

Além desses dois PSM, poderia ser gerado mais um para o banco de dados, mas optamos por não gerar. Essa decisão foi tomada porque o PSM *Hibernate* já conterá informações de mapeamento para o banco de dados. A partir dessas informações, é possível gerar o código-fonte para o *Hibernate*, já contendo anotações sobre o mapeamento para o banco de dados que serão feitas usando a tecnologia XDoclet [49]. Trata-se de uma tecnologia que permite programação orientada por atributo em Java, ou seja, pela adição de metadados (atributos ou anotações) ao código-fonte é possível tornar o código mais significativo. Isso é feito por marcadores *JavaDoc*[16] especiais. O XDoclet faz uma varredura (*parsing*) do código-fonte que identifica os marcadores e gera vários artefatos, como, por exemplo: XML, código-fonte e outros. A união das anotações XDoclets com o *Hibernate* permite que o mapeamento e o banco de dados sejam gerados automaticamente a partir das classes Java geradas. Essa abordagem é interessante, pois permite elevar o nível de abstração da modelagem de dados para o nível usado na Orientação por Objetos.

[16] *JavaDoc* é o padrão de documentação por meio de comentários no código-fonte Java.

4.1.2.1 PSM *Hibernate*

O primeiro PSM é o PSM *Hibernate* gerado pela transformação de PIM *Domain* para PSM *Hibernate* e representa a camada de modelo na tecnologia *Hibernate* [2]. Este modelo é descrito por uma extensão do Perfil Java para UML [41].

Esse modelo contém, nos comentários das classes e das propriedades, anotações XDoclet para o *Hibernate* [49] [2]. Estas anotações são as informações que permitem mapear as classes e suas propriedades para o banco de dados relacional. Usando esse recurso é possível usar o mesmo modelo em vários bancos de dados relacionais, desde que o *Hibernate* tenha suporte ao banco de dados. O interessante desta abordagem é o fato dela ser prática e direta, pois as informações de mapeamento ficam anotadas nas próprias classes, ou seja, em lugar bastante intuitivo.

A única extensão que foi feita ao Perfil UML para Java foi adicionar o estereótipo *Código-Fonte*, que é aplicável a comentário de método. Dessa forma, é possível definir, já no PSM, alguns pequenos trechos de código Java sem atrapalhar a documentação do método.

A Figura 27 representa o exemplo de uma parte de um PSM *Hibernate*. O diagrama dessa figura mostra as principais classes geradas pela transformação de PIM *Domain* para PSM *Hibernate* executada sobre as classes *Usuário* e *Grupo de usuário* do Merci apresentadas da Figura 20. Como pode ser observado, no PSM *Hibernate*, todos os nomes já foram convertidos para nomes válidos Java, seguindo o padrão de nomenclatura Java. Além disso, na classe *Usuario* foram gerados os métodos *get* e *set* para cada atributo. Foi criado também um método de validação para cada um. Esses métodos já contêm o código-fonte de validação necessário para cada atributo. No caso da classe *Usuário*, as validações são: obrigatoriedade de preenchimento dos atributos *Nome*, *Login* e *Senha*, e é verificado também se o login informado já existe. O método *validarSalvamento( )* simplesmente chama os métodos de validação de cada atributo. A classe *GrupoDeUsuario* contém uma constante estática para cada valor fixo da enumeração. Esses valores são: *Gerente*, *Gestor de Estoque*, *Gestor de Compras* e *Caixeiro*. Essa classe contém também o método estático *getTodos( )* que retorna todos esses valores fixos. Por fim, é gerada a classe *UsuarioControle* que permite criar, salvar, excluir e recuperar instâncias da classe *Usuario*. Os métodos salvar, excluir e recuperar não estão na classe *UsuarioControle* porque são genéricos da classe herdada pelo *UsuarioControle*.

Figura 27 - Diagrama do PSM *Hibernate*

4.1.2.2 PSM *Struts*

O segundo PSM gerado é o PSM *Struts*. Este modelo também foi definido por meio de um Perfil UML personalizado, que foi baseado em conceitos e artefatos da tecnologia *Struts* [50] e nas extensões definidas por Ahmed [1] e Koch [17].

Ahmed [1] descreve uma forma de modelar componentes e conceitos Java para Web, denominados Servlet e JSP [38]. Entre esses conceitos podemos citar o próprio JSP, formulário, submissão de formulário, link e redirecionamento.

Koch [17] propõe uma extensão da UML para modelar navegação e apresentação de aplicações Web. O autor introduz conceitos como classe navegacional, navegabilidade direta, índice, consulta, menu, entre outros conceitos existentes em páginas Web e que são muito comuns em aplicações Web.

4.1.2.2.1 Perfil PSM *Struts*

O Perfil do PSM *Struts* é composto pelos estereótipos descritos nas tabelas a seguir, em que a Tabela 5 descreve os estereótipos de classes, a Tabela 6 descreve o estereótipo de atributos, a Tabela 7 descreve os estereótipos de dependências e a Tabela 8 descreve os estereótipos de associações. Logo depois são apresentados exemplos de utilização desses estereótipos.

Estereótipo	Descrição
<<Tiles>>	A classe estereotipada de *Tiles* representa uma definição *Tiles* [8]. Por esta definição é possível expressar modelos de páginas JSP em tempo de execução e herança entre páginas JSPs. Os atributos ou associação desta classe irão descrever as propriedades da definição *Tiles*.
<<JSPF>>	A classe estereotipada de JSPF representa fragmentos de páginas HTML dinâmicas [38] que podem ser usadas por definições *Tiles*.
<<Action>>	A classe estereotipada de *Action* representa classes de ação do *Struts* [50]. Estas classes contêm métodos que são invocados ao acionar comandos ou *links* nas páginas JSP. Cada método retorna uma referência a um JSP ou a uma definição *Tiles,* definindo, assim, qual é o conteúdo a ser exibido após o acionamento do comando ou *link*.
<<FormBean>>	A classe estereotipada de FormBean representa classes de formulário do *Struts* [50]. Estas classes são responsáveis por transferir os dados do formulário da ação para o JSP e vice-versa.

Tabela 5 - Estereótipos de classes do PSM *Struts*

Estereótipo	Descrição
<<Field>>	A classe estereotipada de *Field* representa os campos de um JSP. Esses campos podem permitir que o usuário altere os dados ou podem ser somente para leitura.
<<Column>>	A classe estereotipada de *Column* representa as colunas de uma tabela.

Tabela 6 - Estereótipos de atributos do PSM *Struts*

Estereótipo	Descrição
<<Input>>	A dependência estereotipada de *Input* representa a configuração de página de entrada de dados de uma ação do *Struts*. Esta página pode ser um JSP, uma definição *Tiles* ou uma ação do *Struts*.
<<Forward>>	A dependência estereotipada de *Forward* representa a configuração das possíveis páginas que podem ser exibidas após a execução de alguma operação de uma ação. Assim como o *Input*, esta página pode ser um JSP, uma definição *Tiles* ou uma ação do *Struts*.

Tabela 7 - Estereótipos de dependências do PSM *Struts*

Estereótipo	Descrição
<<Table>>	A associação estereotipada de *Table* representa uma tabela HTML, em que as colunas são representadas pelas propriedades ou operações do JSPF do lado composto da associação.
<<Include>>	A associação estereotipada de *Include* representa a inclusão de um JSP em outro.
<<Command Link>>	A associação estereotipada de *Command Link* representa um *link* HTML. Esta associação deve ser unidirecional de tal forma que a direção da associação indique a ação a ser executada ou JSP a ser exibido ao acionar o *link*.

Tabela 8 - Estereótipos de associações do PSM *Struts*

A seguir são apresentados alguns diagramas que exemplificam a aplicação do PSM *Struts*. Este exemplo mostra a modelagem gerada pela execução da transformação de PIM UI para PSM *Struts* sobre o modelo representado pela Figura 22, Figura 24 e Figura 26.

A Figura 28 mostra a forma como a tela de gestão de usuários (Figura 21) foi implementada. A classe *LeiautePrincipal* é uma classe base para todas as telas. Ela representa a parte comum a todas as telas, por exemplo, cabeçalho, rodapé e menus. A parte comum de todas elas não é gerada automaticamente, pois normalmente essa parte é diferente em cada sistema e portanto o beneficio de gerá-la automaticamente não é grande. O importante é criar essa classe como ponto de extensão em que é possível simplesmente acoplar a parte comum. A classe *PesquisaDeUsuarioTiles* representa a tela de gestão de usuários. Ela herda do *LeiautePrincipal* e, portanto, herda a parte comum da tela no sistema. Por meio da composição *corpo*, é especificado

o JSP que é responsável por montar a tela de gestão de usuários. Esse JSP é o *PesquisaDeUsuarioJSPF* que contém os campos *Nome* e *Login*, os comandos *Pesquisar* e *Exibir todos* e inclui outro JSP, o *GestaoDeUsuarioJSPF* que monta o restante da tela. O JSP *GestaoDeUsuarioJSPF* contém o comando *Criar* e a tabela formada pelo JSPF *DescritorDeUsuarioJSPF*. Essa tabela têm as colunas *Nome* e *Login* e em cada linha da tabela tem os comandos *Excluir*, *Alterar* e *Visualizar*. Cada comando da tela é executado por uma ação do *Struts*, estas ações são representadas pelas classes com o estereótipo <<Action>>. Pela associação *name*, é especificado o formulário que é usado tanto pela ação quanto pelo JSP. A ação é responsável por algumas outras coisas, o que é explicado mais adiante na Figura 31. O formulário é representado pelo <<FormBean>>, em que a classe que o implementa é especificada pela composição *type*. Nesse caso, o formulário *PesquisaDeUsuarioFormBean* é implementado pela classe *PesquisaDeUsuarioForm*, que tem como parte de sua implementação a classe *GestaoDeUsuarioForm*. Note que o formulário tem atributos equivalentes aos campos do JSP.

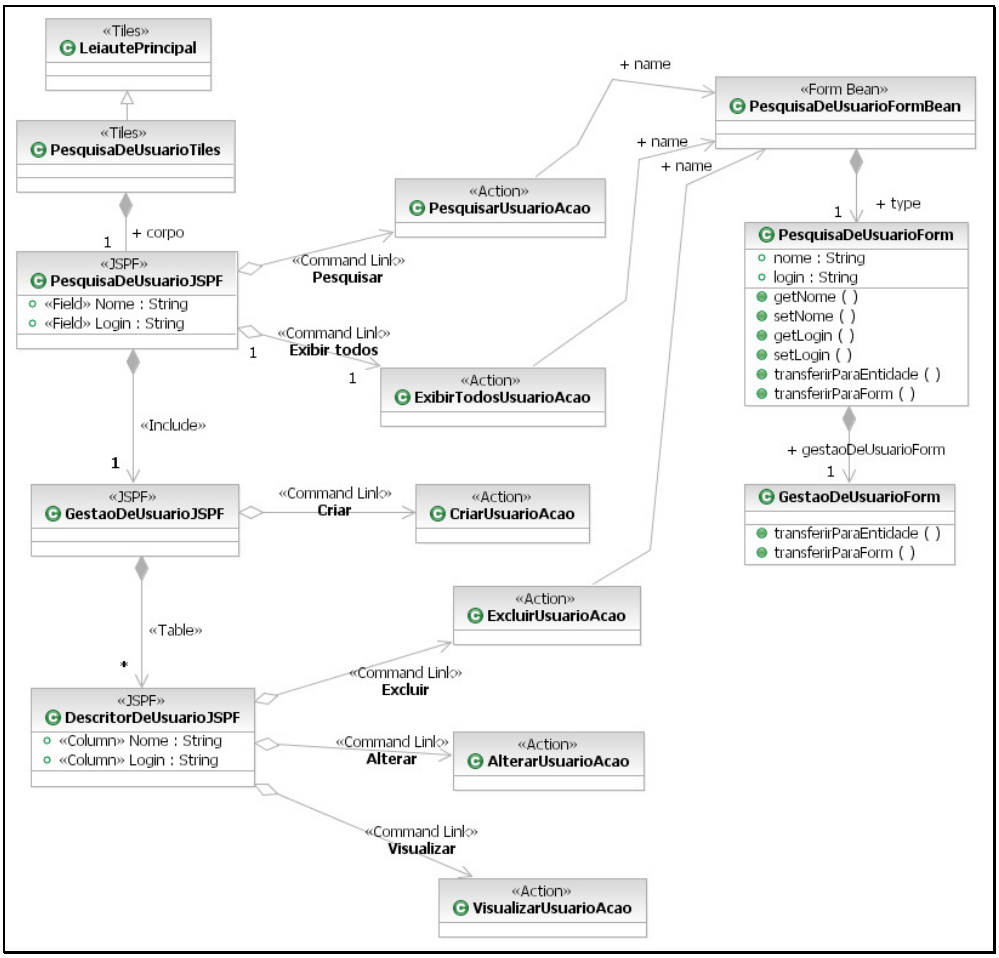


Figura 28 - PSM *Struts* da tela de gestão de usuários

A Figura 29 mostra a forma como a tela de visualização de usuário (Figura 23) foi implementada. Este diagrama é muito parecido com o anterior, portanto destacaremos apenas as

partes específicas. A tela é montada pelo JSP *VisualizacaoDeUsuarioJSPF*, que contém os campos *Nome*, *Login*, *Senha* e *Grupos* e os comandos *Voltar* e *Alterar*. A ação *VoltarDeUsuarioAcao* usa o formulário *VisualizacaoDeUsuarioFormBean* enquanto a ação *AlterarUsuarioAcao* usa o formulário *EdicaoDeUsuarioFormBean*, que é detalhado no diagrama seguinte.

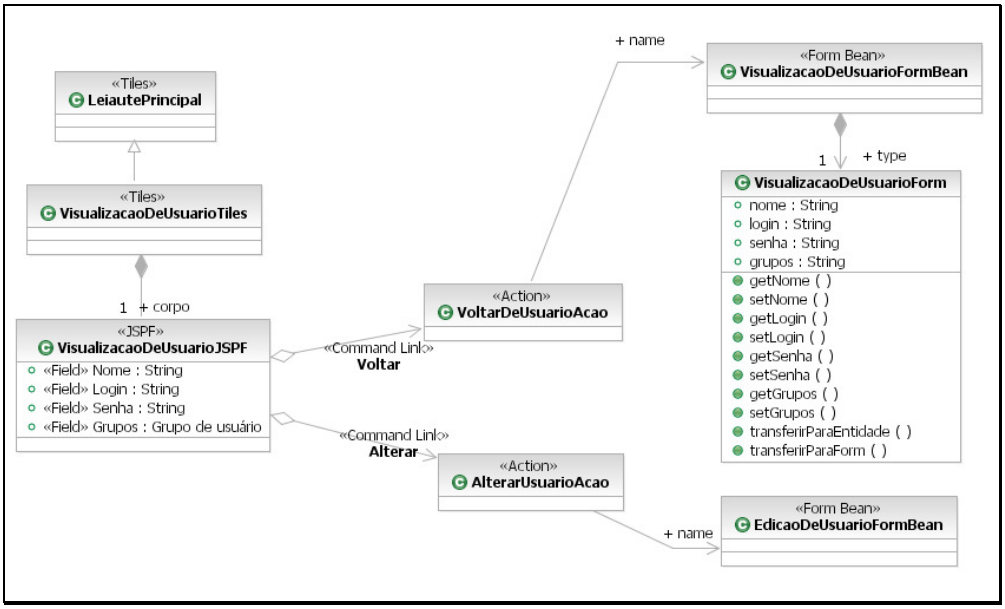


Figura 29 - PSM *Struts* da tela de visualização de usuário

A Figura 30 mostra a forma como a tela de edição de usuário (Figura 25) foi implementada. A diferença deste diagrama para o anterior é que: as classes são nomeadas de *Edicao* em vez de *Visualizacao*, os comandos do JSP *EdicaoDeUsuarioJSP* são *Salvar* e *Cancelar* e, por último, os campos do JSP são campos editáveis, enquanto no diagrama anterior eles não o são. Essa última diferença está presente no modelo, mas não é expressa no diagrama, pois é marcada pela propriedade *Somente leitura* dos atributos que representam os campos.

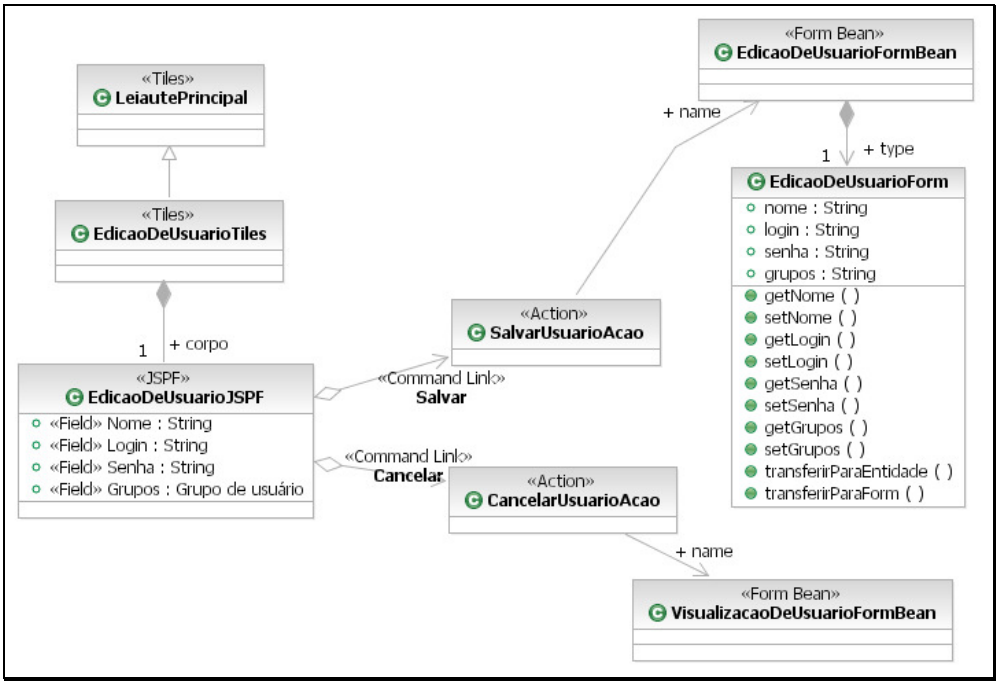


Figura 30 - PSM *Struts* da tela de edição de usuário

A Figura 31 detalha as ações dos diagramas acima. Note que cada ação contém os atributos *tipo*, *className*, *classeDeControle* e *name*. Estes atributos especificam respectivamente a classe Java que implementa o comportamento da ação, a classe Java de mapeamento, a classe de controle que será usada pela ação e o formulário que será usado pela ação. A classe de controle é definida pelo valor inicial do atributo, isso não foi mostrado nesse diagrama para deixá-lo mais limpo e claro. No caso destas ações, a classe de controle será sempre *UsuarioControle*, apresentada no diagrama da Figura 27. Além dessas informações, a ação tem que especificar obrigatoriamente qual é a tela de entrada, representada pela dependência <<Input>>. Essa tela pode ser um <<Tiles>> ou outra ação <<Action>>. No segundo caso, a ação considerada como tela de entrada tem que ter como entrada um <<Tiles>>, caso contrário, ao acionar a ação, poderá ocorrer um *loop* infinito ao tentar exibir uma tela de entrada que no final tem que ser um <<Tiles>>. Além da tela de entrada é possível especificar telas alternativas que seriam exibidas de acordo com a lógica implementada pela classe Java da ação. Essa alternativa é expressa pela dependência <<Forward>>, que obrigatoriamente tem que ser nomeada. Por exemplo, a ação implementada pela classe *AcaoSalvar* tem que especificar o <<Forward>> *sucesso* e a ação implementada pela classe *AcaoCancelar* tem que especificar o <<Forward>> *nenhumObjetoCriado*. No primeiro exemplo, se o salvamento não ocorrer com sucesso, é exibida a tela *EdicaoDeUsuarioTiles*. Se ocorrer com sucesso, é executada a ação *VisualizarUsuarioAcao*, que exibe a tela *VisualizacaoDeUsuarioTiles*. No segundo exemplo, se o objeto (*Usuário*) está sendo criado, ao executar o cancelamento, é executada a ação *ExibirTodosUsuarioAcao*, que exibe a tela *PesquisaDeUsuarioTiles*. Se o objeto (Usuário) está sendo alterado, ao cancelar é exibida a tela *VisualizacaoDeUsuarioTiles*.

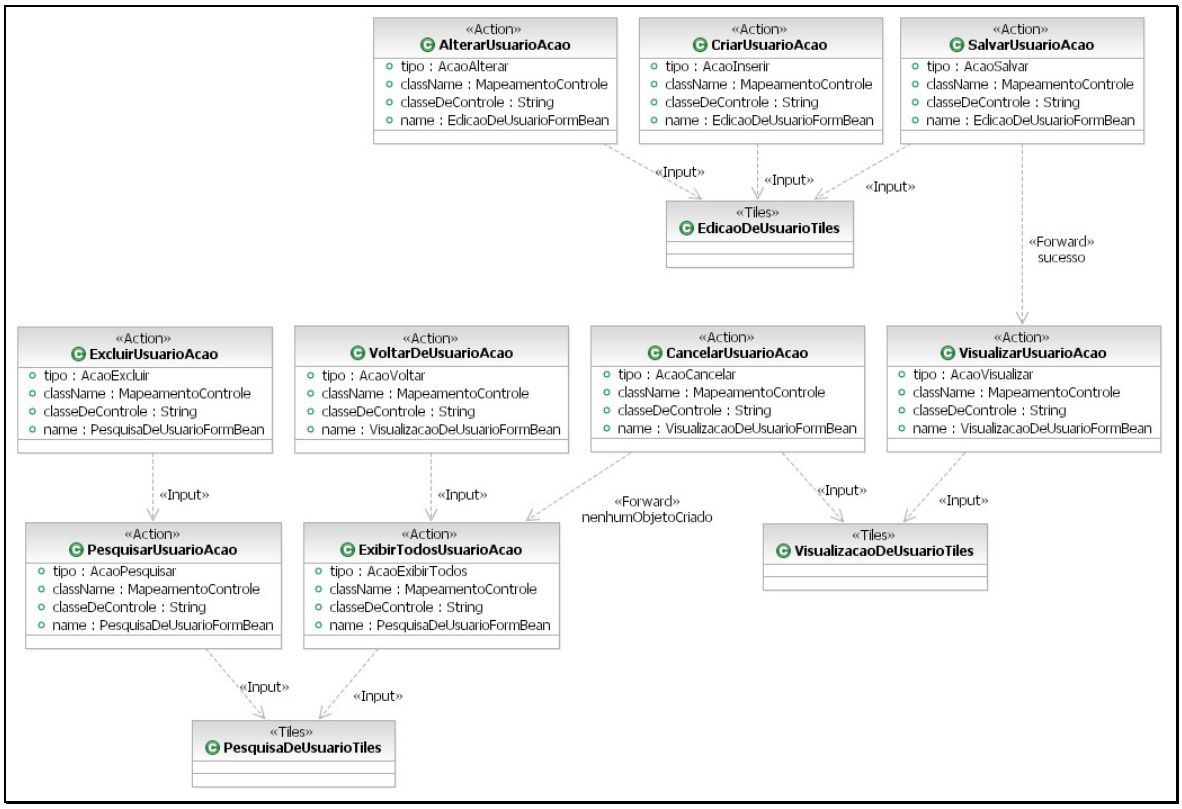


Figura 31 - PSM *Struts* que detalha as ações das telas de gestão, visualização e edição de usuário

4.2 Considerações sobre as transformações

Na seção 4.1, junto com os modelos foram descritas sucintamente as principais regras de transformações de modelo para modelo. Depois de gerados os modelos PSM *Hibernate* e PSM *Struts*, é necessário executar mais duas transformações para gerar o código-fonte executável da aplicação. Essas transformações são bem diretas, pois estes modelos já estão em nível de detalhamento bastante elevado. No caso do exemplo de controle de usuários do Merci, que está sendo usado ao longo deste trabalho, a transformação de PSM *Hibernate* para código-fonte gera as classes Java do diagrama da Figura 27. A transformação de PSM *Struts* para código-fonte gera as classes Java dos formulários, os JSPs e os arquivos XML, que representam as ações do *Struts*, os *FormBeans* do *Struts* e as definições do *Tiles* que estão representados na Figura 28, Figura 29, Figura 30 e Figura 31.

Com esse código-fonte gerado é possível executar a aplicação. A Figura 21, Figura 23 e Figura 25 representam as telas de controle de usuários do Merci. Além de terem sido usadas como

exemplo no decorrer deste trabalho, essas telas foram, de fato, geradas automaticamente pelas transformações a partir do PIM *Domain* representado na Figura 20, seguindo os passos descritos neste capítulo.

4.2.1 Ferramentas e tecnologias

Para implementar as transformações e os modelos proposto por este trabalho, foi feito um levantamento das ferramentas que suportam transformações no estilo do MDA. A escolha da ferramenta para implementar as transformações foi difícil, pois são ferramentas novas e, até hoje, evoluem rapidamente. Entre as possibilidades pesquisadas, escolhemos a *Rational Software Architect* (RSA), da IBM. Resumidamente, os principais motivos desta escolha foram: implementação de transformação na linguagem Java que permite maior flexibilidade e uso das APIs (*Application Programming Interface*) EMF (*Eclipse Modeling Framework*) [10] e JDT (*Java Development Tools*) [51] do Eclipse que são APIs não-proprietárias de código-fonte aberto. Essas APIs são usadas para manipular modelos UML ou MOF e código-fonte Java, respectivamente. Esses são os principais artefatos manipulados em uma transformação.

4.3 Método de Desenvolvimento e Geração de Interfaces - MDGI

A criação do Método de Desenvolvimento e Geração de Interfaces (MDGI) é um dos resultados deste trabalho. Até o momento descrevemos a aplicação de um exemplo específico, que foi descrito em detalhes para mostrar como o MDGI pode ser aplicado. Nesta seção, descreveremos o MDGI de forma genérica.

Normalmente, no método de desenvolvimento tradicional, sem usar técnicas do MDSD e MDA, quando é feito o modelo de desenho (projeto) do sistema, este serve apenas como referência para a implementação. Tanto o próprio modelo quanto os mecanismos de automatização existentes são limitados e praticamente todo o sistema deve ser implementado manualmente. Com isso, à medida que a implementação do sistema avança, esse modelo vai perdendo seu valor, pois normalmente as alterações realizadas durante a implementação não são nele refletidas. Embora o modelo de referência tenha um papel importante no desenvolvimento, o esforço nele gasto não é reaproveitado na implementação do sistema, pois os mecanismos de automatização para gerar código-fonte são limitados.

Com a introdução do MDA, o modelo passa a ter um valor maior durante o desenvolvimento do sistema, pois alterações realizadas no modelo implicam alterações

automatizadas no código-fonte. Entretanto, o MDA não especifica como esses modelos devem ser usados para tratar a interação com o usuário. O MDGI propõe uma forma de tratar esse aspecto.

O MDGI prescreve a separação do PIM em dois aspectos, um que descreve o domínio do sistema e outro que descreve a interação com o usuário, e a existência de pelo menos uma transformação do primeiro para o segundo aspecto. Conforme descrito ao longo deste trabalho, o PIM *Domain* e o PIM UI são as instanciações destes dois aspectos, assim como a transformação de PIM *Domain* para PIM UI é a instanciação da transformação definida como necessária pelo MDGI.

A separação do PIM nesses dois aspectos permite trabalhar em alto nível e ainda ter flexibilidade suficiente para modelar a interação do sistema com o usuário de maneira a conseguir atender os requisitos do sistema de forma satisfatória. A modelagem da interação com o usuário em alto nível é importante, pois podemos abstrair de detalhes de tecnologia. Conforme já discutido na seção 2.2.1, esse nível de abstração já foi proposto por alguns autores, mas nossa abordagem vai além, permitindo que o modelo seja usado para gerar o sistema com código-fonte de qualidade, separando os conceitos de domínio da interface com o usuário. E, ainda, a modelagem da solução que atende aos requisitos do sistema pode ser reaproveitada para gerar a aplicação para diferentes tecnologias ou plataformas. Isso é possível graças ao mecanismo de transformação de PIM para PSM previsto no MDA. Além dessa independência de plataforma, o mecanismo de transformação permite que o conhecimento de utilização de determinada tecnologia seja embutido na própria transformação. Com isso, a aplicação desse conhecimento pode ser automatizada pela execução da transformação sobre os modelos PIMs.

Não basta apenas dividir o PIM em dois aspectos. A transformação do PIM *Domain* para o PIM UI também tem um papel importante no MDGI. Essa transformação deve implementar a aplicação de padrões para gerar o modelo de interação com o usuário que poderá ser alterado para adaptar-se aos requisitos do sistema. Depois de aplicado o MDGI, os passos seguintes seguem o MDA. Os dois modelos PIM, o de domínio e o de interação, são transformados em modelos PSM, que por sua vez são transformados em código-fonte. Portanto, o MDGI pode ser considerado uma forma para se usar ou criar o PIM.

A separação da modelagem de domínio da modelagem da interação com o usuário é um ponto importante neste trabalho. A separação desses dois aspectos está presente em várias técnicas no processo de desenvolvimento de *software*. Entre elas destacamos, por exemplo, as de análise, como a apresentada por Rosenberg [24], que comprova a importância da separação destes dois aspectos. No desenvolvimento de sistemas também existem padrões que recomendam a implementação de objetos de negócio sem envolver aspectos de interface com o usuário [30]. Esses objetos, em sua maioria, são responsáveis por implementar os dados e regras de negócio do domínio da aplicação. Na modelagem independente de plataforma proposta pelo MDA, isso não poderia ser diferente. Por esse motivo e por outros, apresentados ao longo deste trabalho, propõe-se a separação do PIM nesses dois aspectos. A seguir destacamos alguns benefícios dessa separação:

- Conceitos e regras do domínio podem ser reutilizados em várias interfaces com o usuário sem replicação de modelagem e, conseqüentemente, de código. Por exemplo, na *Tela de edição de usuário*, mostrada na Figura 25 (p. 53), pode ser marcado se o usuário é *Gerente*, *Gestor de Estoque*, *Gestor de Compras* e/ou *Caixeiro*. No caso do Merci, essas quatro opções são os grupos de que o usuário pode participar e em qualquer parte do sistema que eles aparecerem as opções serão sempre essas, ou seja, é uma característica do domínio da aplicação, e não da tela específica. Portanto, é importante que essa característica seja modelada separadamente da tela. No nosso exemplo, a modelagem é feita no diagrama da Figura 20 (p. 48). Dessa forma, a modelagem da tela não precisa replicar as informações já modeladas, basta referenciar, como é feito no diagrama da Figura 26 (p. 54). Nesse diagrama, essa referência é feita pelo trecho "<<Editável>> Usuário a ser editado.Grupos" na única nota desse diagrama. No nosso exemplo, podemos mostrar a reutilização do conceito de *Grupo de usuário* em outra tela por meio da *Tela de visualização de usuário* mostrada na Figura 23 (p. 51). No diagrama dessa tela, representado na Figura 24 (p. 52) esse conceito é reutilizado simplesmente fazendo-se referência, da mesma forma que foi feito no diagrama da outra tela.
- Os conceitos do domínio são modelados de forma clara e o relacionamento das interfaces com o usuário com esses conceitos é explícito, facilitando o entendimento das regras de negócio do sistema. Isso evita a duplicação de trabalho, pois não é necessário descrever os conceitos e suas regras em toda interface em que ele é usado, basta referenciá-lo. Isso também favorece a manutenção, pois como o conceito está descrito apenas em um lugar, alterações nele são centralizadas.
- Permite a geração automática da interface com o usuário a partir de padrões aplicados no modelo de domínio (PIM *Domain*).

A transformação de PIM *Domain* para PIM UI implementada e descrita neste trabalho contempla a aplicação de apenas um padrão que pode ser chamado Caso de uso CRUD[17]. Conforme já explicado detalhadamente, o caso de uso CRUD consiste em criar interfaces com o usuário para criar, visualizar, alterar e excluir entidades que representam os conceitos do domínio do sistema. Depois de aplicada a transformação e, conseqüentemente, o padrão CRUD, pode-se alterar a modelagem da interface para atender os requisitos, excluindo dela alguma dessas operações e/ou incluindo outras. Essa característica de alterar a aplicação do padrão é análoga à

[17] O termo CRUD vem do inglês *Create* (criar), *Retrive* (recuperar), *Update* (atualizar) e Delete (apagar). Este termo é muito usado no contexto de banco de dados representando as operações básicas. Entretanto, é muito comum que casos de uso especifiquem estas operações.

idéia de padrões de projeto [11] que declara que o padrão é um guia para solução de um problema. A aplicação do padrão deve ser adaptada ao contexto onde ele está sendo aplicado.

O CRUD foi escolhido porque é um padrão básico presente em grande parte das aplicações, principalmente nas aplicações *enterprise*. Entretanto, nossa transformação pode ser estendida para acrescentar novos aspectos ou variações a esse padrão. Podem ser implementadas também outras transformações com o intuito de automatizar a aplicação de outros padrões, inclusive o intuito é que, em projetos reais, isso seja feito.

Conforme já explicado, o MDGI obriga a existência de um modelo que represente o domínio da aplicação e um que represente as interfaces com o usuário. Os dois não precisam ser o PIM *Domain* e o PIM UI que foram definidos neste trabalho, mas recomendamos a utilização deles. No entanto, se os modelos definidos aqui não satisfizerem as necessidades específicas de desenvolvimento, podem ser alterados, estendidos ou redefinidos.

4.4 Discussão dos resultados

Nesta seção, comparamos duas abordagens de desenvolvimento do sistema Merci. Na primeira abordagem será usado o processo Praxis padrão. A aplicação desenvolvida dessa forma encontra-se no endereço eletrônico "http://www.wppf.uaivip.com.br/praxis/2.1/Praxis 2.1.htm". Neste endereço são disponibilizados o código-fonte e toda a documentação de implementação da aplicação conforme o Praxis padrão. Na segunda abordagem, serão usados os modelos e transformações definidos neste trabalho.

A comparação será feita nos seguintes aspectos:

- Diferenças dos modelos UML usados.
- Quantidade de código-fonte gerado automaticamente.
- Quantidade de elementos UML gerados.
- Padronização e reaproveitamento de convenções e soluções de implementação (reuso).
- Quantidade de código-fonte das transformações.
- Qualidade.
- Produtividade.

4.4.1 Diferenças dos modelos UML

No Praxis, a modelagem do desenho (*design*) é separada em duas partes: o Modelo do Desenho Externo (MDE) e o Modelo do Desenho Interno (MDI). A primeira tem o objetivo de descrever a funcionalidade do sistema sob o ponto de vista do usuário, sem detalhes técnicos. A segunda tem o objetivo de descrever internamente como o sistema foi implementado.

O MDE é constituído por classes que representam as interfaces com o usuário, dispostas em dois diagramas de classes, um de estrutura dinâmica, que modela detalhadamente a navegação entre as telas, e um diagrama de estrutura estática, que modela as associações entre as telas: generalizações, composições e agregações. O relacionamento de generalização modela a herança, a composição modela partes da tela e a agregação modela, em alto nível, a possível navegação entre telas. Além desses diagramas de classes, o MDE contém também os diagramas de seqüência que apresentam os principais fluxos das interações do usuário com o sistema. Esses diagramas de seqüência são agrupados por Casos de Uso de Desenho[18]. O MDE contém também os protótipos das interfaces com o usuário.

O MDI é constituído pelas classes que implementam o sistema. Elas são dispostas em diagramas de classes de fronteira, de controle e de entidade.

Os conceitos de MDE e MDI do Praxis têm algumas características parecidas com os conceitos de PIM e PSM do MDA. O MDE e o PIM se propõem modelar o sistema com foco na funcionalidade e não em detalhes de tecnologia ou plataforma. A diferença entre os dois é que no MDE existe a preocupação em modelar o sistema sob o ponto de vista do usuário, ou seja, só as interfaces com o usuário são modeladas. No PIM, não existe essa restrição e, normalmente, tem-se uma preocupação grande em modelar os conceitos que o sistema representa e implementa. O MDI e PSM se propõem modelar detalhes de implementação do sistema como um todo, considerando principalmente os aspectos de tecnologia ou plataforma.

É importante destacar que, no MDE, a falta de elementos que modelam os conceitos do sistema proíbe a execução de transformações sobre o mesmo, com o objetivo de automatizar o processo de desenvolvimento do *software*, como é proposto pelo MDA. O PIM *Domain* proposto neste trabalho supre justamente essa deficiência. Já o PIM UI o complementa com informações de interação com o usuário. A junção do PIM *Domain* e do PIM UI permite a automatização do desenvolvimento do *software* e ainda continua a permitir a modelagem de interface com o usuário, em alto nível, sem detalhes de implementação. Além disso, o PIM UI é gerado automaticamente a partir do PIM *Domain*. Para requisitos específicos, que vão além das operações básicas de

[18] O Caso de Uso de Desenho descreve a solução final de interação do usuário com o sistema de um Caso de Uso de Análise inteiro ou parte dele.

consulta, criação, exclusão e alteração de entidades, faz-se necessário alterar ou complementar o PIM UI. Entretanto, no contexto de elaboração do PIM, foi eliminado o trabalho manual que envolve essas operações básicas, que são bastante comuns em sistemas.

Existem ainda outras diferenças entre o PIM e o MDE, como, por exemplo, a existência de protótipos e de Casos de Uso de Desenho no MDE. Mas a diferença mais importante, considerando o contexto deste trabalho, foi a já descrita. Esses elementos não existentes no PIM poderiam continuar a ser criados manualmente ou talvez não sejam mais necessários, pois as transformações permitem gerar um sistema funcional bem próximo do que se pretende como versão final do sistema. Com isso, é possível fazer validações com essa versão do sistema, gerada automaticamente.

4.4.2 Quantidade de código-fonte gerado

Como estamos comparando a implementação da mesma funcionalidade e desejamos que o código-fonte gerado tenha pelo menos a mesma qualidade, uma de nossas expectativas é que a quantidade de linhas do código-fonte seja bem parecida entre as implementações das duas abordagens.

Conforme a apresentado na Tabela 9, a diferença na quantidade de linhas é muito pequena entre as duas abordagens, confirmando nossa expectativa. Detalhes da contagem de linhas são apresentados na seção A.1.1.

Abordagem de desenvolvimento	Quantidade de linhas do sistema
Não-automatizado (Praxis)	689
Automatizado (MDGI)	697

Tabela 9 - Comparação da quantidade de linhas do código-fonte

4.4.3 Quantidade de elementos UML gerados

Sobre os elementos UML usados na modelagem das duas abordagens existem grandes diferenças, tanto em relação à quantidade quanto ao tipo dos elementos. A Tabela 10 mostra essas diferenças, que são justificadas pelo fato de os modelos serem bem diferentes entre as duas abordagens, conforme discutido na seção 4.4.1.

Modelo	Diagramas de classe	Classes	Propriedades	Associações	Notas de referência a atributos	Referências a atributos	Operações	**Total de elementos dos diagramas de classes**	Diagramas de seqüência	Objetos	Mesagens	**Total de elementos dos diagramas de seqüência**	**Total**
Quantidade de elementos no Praxis													
MDE Merci	1	3	6	2				**12**	2	3	3	**8**	**20**
MDE Plataforma Praxis	3			7			6	**16**	7	22	29	**58**	**74**
Total MDE	**4**	**3**	**6**	**9**			**6**	**28**	**9**	**25**	**32**	**66**	**94**
MDI Entidade	1	2	5	1			27	**36**					**36**
MDI Controle	1	1					14	**16**					**16**
MDI Fronteira	1	1					8	**10**					**10**
MDI Plataforma Praxis (Fronteira)	3	3					43	**49**					**49**
Total MDI	**6**	**7**	**5**	**1**			**92**	**111**					**111**
Quantidade de elementos na abordagem MDGI													
PIM Domínio	1	2	3	1				**7**					**7**
PIM UI	2	9		13	2	6		**32**					**32**
Total PIM	**3**	**11**	**3**	**14**	**2**	**6**		**39**					**39**
PSM Hibernate	1	3	4	1			22	**31**					**31**
PSM Struts	3	15	19	24			12	**73**					**73**
Total PSM	**4**	**18**	**23**	**25**			**34**	**104**					**104**

Tabela 10 - Comparação da quantidade de elementos de modelagem

A Plataforma Praxis foi incluída na contagem dos elementos na abordagem Praxis porque é nela que grande parte da funcionalidade está descrita.

Ao comparar o total de elementos do MDE (94) com o total do PIM (39), percebemos que o MDE tem mais que o dobro de elementos. Isso se deve aos diagramas de seqüência que não estão presentes na abordagem MDGI. Desconsiderando os diagramas de seqüência, o total de elementos de diagramas de classe do MDE (28) já fica mais próximo do total do PIM (39). Agora essa diferença se deve ao fato de o MDE do Praxis não representar os conceitos do sistema, que na abordagem MDGI é representado pelo PIM *Domain*.

Ao comparar o total de elementos do MDI (111) com o total do PSM (104), percebemos que os valores são bem próximos. Esse fato é justificado pelo mesmo motivo da seção anterior, pois esses modelos representam diretamente o código-fonte do sistema.

Essas comparações todas são feitas para reforçar dois pontos. Primeiro, os diagramas de seqüência representam uma quantidade grande em termos de elementos UML e,

conseqüentemente, construí-lo ou consultá-lo demandará esforço considerável. A abordagem MDGI dispensa os diagramas de seqüência, conforme explicado na seção 4.4.1. Segundo, a abordagem MDGI não necessita muitos elementos a mais do que uma abordagem não-automatizada. Os elementos a mais necessários são representados pelo PIM *Domain*. Apesar de serem mais elementos no modelo, são fáceis de serem criados por dois motivos. Primeiro, porque comparado ao PIM UI é necessário uma quantidade bem menor de elementos. Segundo, porque esses elementos podem ser extraídos quase que diretamente do modelo de entidade da análise, o qual é usado como insumo essencial pelo Praxis para gerar o MDE, mas não é incluído no MDE.

4.4.4 Padronização e reutilização

Tanto na implementação do Merci pelo Praxis, quanto na implementação deste trabalho, houve a preocupação de criar elementos reutilizáveis. Entretanto, a nossa abordagem (MDGI) permite uma dimensão de reutilização que não é possível no Praxis. Essa dimensão é possível graças às regras de transformação, que são reutilizadas na transformação de cada elemento de modelagem transformado. Esse não é simplesmente um reaproveitamento de componente de *software*, mas pode ser considerado como o reuso do conhecimento de utilização de alguma tecnologia. Neste trabalho foram criadas transformações que permitem a reutilização de conhecimentos relacionados à implementação de vários aspectos nas tecnologias *Hibernate* e *Struts*.

Por exemplo, como uma composição entre duas classes deve ser implementada usando o *Hibernate*? A resposta a essa pergunta está implementada em uma regra de transformação específica. O mesmo ocorre com a classe de enumeração, a navegação de uma página para outra, o salvamento dos dados de uma entidade e vários outros aspectos.

Além dessa reutilização, é importante destacar que a execução dessas transformações gera outro resultado importante. Os modelos e o código-fonte criados pelas transformações serão altamente padronizados, pois os elementos de modelagem com as mesmas características, serão gerados pela mesma regra de transformação. Principalmente em sistemas desenvolvidos por equipe, esta padronização é difícil de se alcançar, em função da arbitrariedade de soluções possíveis, pois, a solução de implementação dada por cada desenvolvedor dependerá de sua experiência técnica. Por mais que se tente difundir os padrões de soluções e desenho na equipe, é difícil manter a disciplina, dada a natureza humana, nesse tipo atividade. Portanto, a padronização é difícil de se alcançar e não é totalmente garantida. A longo prazo, essa padronização ajuda na manutenção do sistema, pois as soluções de implementação serão mais claras e centralizadas.

4.4.5 Quantidade de código-fonte das transformações

O volume de código-fonte das transformações é um dos pontos mais importantes da comparação entre as duas abordagens. Conforme apresentado na Tabela 11, a contagem de linhas

das transformações foram divididas em três módulos. Mais detalhes dessa contagem de linha, veja na seção A.1.2.

Módulo da transformação	Quantidade de linhas
1. Regras de transformações	3.915
2. Elementos reutilizáveis entre as regras	1.638
3. Manipulação de XML (gerado automaticamente)	21.230

Tabela 11 - Quantidade de linhas de código-fonte das transformações

O módulo 3 é usado para manipulação dos XMLs gerados na transformação de PIM *Struts* para código-fonte. Este módulo foi todo gerado automaticamente a partir dos DTDs[19] do XML do *Struts* e do *Tiles*. Como o esforço para gerar essa grande quantidade de linhas de código é muito pequeno, uma vez que se conhece o DTD do XML, não faz sentido incluir este módulo na comparação. Além disso, poderiam ter sido usadas outras abordagens de manipulação de XML que dispensariam esse módulo. O número de linhas de código foi exposto aqui por completeza e para que fiquem claros os motivos pelos quais o módulo não foi incluído na comparação.

O módulo 2 contém funcionalidades que facilitam a manipulação dos elementos UML e são usadas em várias regras de transformações. Ele foi separado em um módulo à parte porque seu conteúdo não sofrerá muitas modificações caso seja necessário alterar ou acrescentar regras de transformação.

O módulo 1 contém as lógicas das regras de transformação. A quantidade de linhas desse módulo (3.915 linhas, ver Tabela 11) é praticamente seis vezes maior do que a quantidade de linhas de código gerado na aplicação (697 linhas, ver Tabela 9). Em relação a essa diferença, gostaríamos de destacar alguns aspectos:

1. Essa grande diferença é um indicativo forte de que o esforço necessário para implementar a transformação é bem maior do que para implementar a aplicação manualmente, o que seria normalmente esperado.
2. Apesar dessa diferença de tamanho ao comparar com o código-fonte do sistema gerado, deve-se lembrar que, além do código-fonte, essas transformações geram também os modelos PIM UI, PSM *Hibernate* e PSM *Struts*. Em outras palavras, parte do código do módulo é responsável pela criação desses modelos e não pela geração direta do código da aplicação. Em nossa abordagem, não faz sentido desconsiderar a parte do módulo responsável por gerar os modelos, pois sem eles não é possível gerar o código-fonte.

[19] Document Type Definition (DTD) é uma das formas de descrever a estrutura de documentos XML.

Embora a quantidade de linhas comparada seja em relação apenas ao código gerado por outra abordagem, é importante destacar que além do código são gerados também os modelos, o que é uma característica positiva.

3. A implementação das transformações tende a ser mais complexa do que a do sistema, pois ela tem que considerar a complexidade dos modelos e código-fonte que ela está gerando mais a complexidade de implementação da própria transformação.

Apresentados esses aspectos, sugere-se que, ao optar por implementar transformações, primeiro é preciso certificar-se de que cada regra de transformação tenha um grau mínimo de reutilização (por exemplo, superior a seis vezes). Cabe ressaltar que a reutilização pode também se dar no desenvolvimento de um mesmo sistema. Quanto maior essa reutilização mais diluído será o custo de implementação das transformações. Caso contrário, o custo/benefício da abordagem MDGI não justifica.

É importante frisar que o parâmetro linhas de código usado para chegar a esse número de 6 vezes é controverso. Entretanto, ele já é um indicativo forte e já nos permite ter uma noção mais clara do custo da criação de regras de transformação.

A fim de tentarmos confirmar esse valor de seis vezes, usamos um outro parâmetro: o esforço para implementar as transformações foi de 196,2 horas e o esforço total para implementar o sistema sem automatização foi de 36,1 horas, conforme apresentado na Tabela 12. Ao comparar esses dois valores chegamos ao valor 5,4, que é bem próximo de seis. Isso novamente é indicador de que para implementar as transformações é necessário esforço seis vezes maior do que para implementar a funcionalidade desejada no sistema.

4.4.6 Qualidade

De forma geral, a qualidade não é diretamente ligada ao fato de o desenvolvimento ser ou não automatizado. A qualidade está mais ligada aos processos de verificação, revisão, inspeção, testes e experiência das pessoas envolvidas no desenvolvimento da aplicação ou das transformações. O Praxis prevê todas essas tarefas para garantir a qualidade do *software*. Entretanto, estas tarefas podem e devem continuar a ser realizadas na abordagem de desenvolvimento automatizado, para garantir a qualidade do produto final.

Por outro lado, quanto maior a utilização de transformações, maior será a tendência de soluções padronizadas, tanto do ponto de vista do desenho interno quanto do externo. Levando em conta a qualidade do sistema, isso pode ser considerado fator positivo, pois essa padronização é naturalmente alcançada. No entanto, no processo sem automatizações, essa padronização pode ser alcançada com treinamentos, revisões e outras atividades afins.

4.4.7 Produtividade

Observe na Tabela 12 que o esforço para gerar o sistema (8,7 horas) é 4 vezes menor que o esforço para implementar o sistema (36,1 horas). Portanto, se desconsiderarmos a implementação das transformações, existe um ganho de produtividade considerável.

Modelo	Esforço em horas
Abordagem não automatizada	
MDE	5,7
MDI	12,7
Código Fonte	17,7
Total	**36,1**
Abordagem automatizada	
PIM Domain	0,8
PIM UI	4,0
PSM Hibernate	0,4
PSM Struts	1,2
Código Fonte	2,3
Total	**8,7**

Tabela 12 - Esforço para construção de parte do sistema

A fim de confirmar se a quantidade de esforço gasta no Merci (abordagem não-automatizada) é próxima do esforço de desenvolvimento fora de ambientes controlados, comparamos os dados do Merci com os dados do *Relatório de métricas* de um projeto do Synergia. Os dados usados desse relatório são de casos de uso com complexidade considerada simples que seria a complexidade equivalente ao Caso de uso gestão de usuário do Merci. No processo personalizado do Synergia, o MDI não é considerado um modelo formal. O processo personalizado considera que o Desenho Interno é realizado durante a tarefa de codificação e é documentada pelo próprio código-fonte. Na comparação, mostrada na Tabela 13, pode-se verificar que a diferença do esforço total entre o caso de uso do Merci e a média dos casos de uso do Synergia é pequena (3,5%). Isso dá evidências que os dados apresentados pelo Merci são dados bem próximos da realidade.

Modelo	Esforço em horas
Caso de uso gestão de usuário do Merci	
MDE	5,7
Código Fonte + MDI	30,4
Total	**36,1**
Média de casos do uso simples de um projeto do Synergia	
MDE	5,8
Código Fonte + Desenho Interno	31,6
Total	**37,4**

Tabela 13 - Comparação de esforço de construção do Merci com um projeto do Synergia

Como foi discutido na seção 4.4.5, o esforço de implementação das transformações é grande e a tendência é que o custo desse esforço se dilua a medida que as transformações sejam reutilizadas em funcionalidades diferentes do sistema ou dos sistemas.

É importante destacar que na implementação das transformações deve-se ter cuidado especial, pois transformações com soluções ruins poderão espalhar soluções ruins pelo sistema inteiro. Por isso, as transformações devem implementar soluções maduras e consolidas. Caso contrário, grandes retrabalhos podem ocorrer cujas conseqüências não foram exploradas nesta pesquisa.

Capítulo 5

Conclusões

A partir de experiências de produção de *software* usando modelos, este trabalho identificou possíveis aspectos nos quais o desenvolvimento pode ser melhorado. Sobre esse assunto foram estudadas e apresentadas metodologias como o MDSD e MDA. Com esses estudos, decidiu-se fazer uma aplicação prática da utilização do MDA com o intuito de analisar o resultado final dessa aplicação. Nenhum trabalho, dentre os pesquisado sobre MDA, especifica como devem ser tratados o desenvolvimento e a modelagem da interação com o usuário, o que é aspecto importante na produção de *software*. Foram estudados modelos de interação com o usuário antes e depois do surgimento do MDA e nenhum deles se adequava completamente à especificação de MDA. Por isso, baseando-se nesse estudo, decidiu-se criar o PIM UI, que é um modelo independente de plataforma de interação com o usuário. As principais contribuições desta pesquisa foram a criação do MDGI e a definição do PIM UI.

Além do PIM UI, foram definidos também três modelos: o PIM *Domain* e PSM *Hibernate* e o PSM *Struts*. Sob o ponto de vista de contribuição deste trabalho, estes modelos têm importância secundária. O PIM *Domain* porque seus elementos e a forma com que eles são usados já são bem conhecidos em várias experiências usando o MDA. O PSM *Hibernate* e o PSM *Struts* por representarem a implementação em tecnologias e plataformas específicas. Esses modelos podem ser substituídos por outros representando a implementação em outras tecnologias. No entanto, eles são fundamentais para aplicação do MDA e portanto foram fundamentais para possibilitar a aplicação prática do PIM UI, validando-o informalmente e possibilitando a análise dessa aplicação.

A aplicação MDGI proposta neste trabalho foi comparada com o desenvolvimento usando o Praxis, no qual não é previsto nenhum tipo de automatização de geração de modelos e de código-fonte funcional. Ao analisar esta comparação, foram destacados três aspectos em que houve grandes diferenças.

Primeiro, sob o ponto de vista de padronização e reutilização, foi possível encapsular, com o uso das regras de transformações, a implementação de um dado modelo, independente de tecnologia, em determinada tecnologia. O Praxis, assim como a maioria dos processos, não se preocupa em padronizar, documentar e repetir esse mapeamento, que fica a cargo do conhecimento específico de cada projetista e de cada implementador e, portanto, esse mapeamento acaba as vezes não sendo considerado no processo de desenvolvimento de *software*. Com a

abordagem MDGI, atingiu-se um nível de abstração em que se conseguiu fazer modelagem independente de tecnologia e reutilizar conhecimentos implementados sob forma de regras de transformações para gerar implementações específicas de tecnologia. Esse nível de reaproveitamento não é possível sem a adoção de abordagens como o MDA e o MDSD, que naturalmente acabam proporcionando uma padronização na forma com que esse mapeamento é feito. Com o PIM UI foi demonstrado que esse tipo de reutilização é possível também no nível de interface com o usuário.

Segundo, conforme mostrado na comparação de produtividade, na apresentação de quantidade de elementos UML e código-fonte gerados e conforme comprovado pelos artigos apresentados na seção 2.1.2.1, a abordagem MDGI proporciona ganho de produtividade considerável, desconsiderando o esforço de implementação das transformações.

Terceiro, apesar deste ganho de produtividade, é importante ressaltar que ele não vem de graça. Inclusive, o custo é elevado e corresponde ao de implementação das transformações e da definição dos modelos necessários para elas. Neste trabalho, nossas medições indicam que a implementação das transformações chega a gastar um esforço seis vezes maior do que o necessário para implementar o que elas se propõem a gerar. Portanto, antes de decidir implementar transformações é altamente recomendado analisar se elas serão suficientemente reutilizadas a ponto de pagarem esse custo e ainda obterem ganhos.

Além dessa questão, é importante ressaltar que devem ser implementadas transformações que geram modelos e código-fonte com soluções maduras e consolidadas. Caso contrário, o risco desse custo alto pode ser ainda mais elevado, acarretando grandes retrabalhos para correção das soluções imaturas.

Outro ponto que é importante destacar em relação ao código-fonte gerado pelas transformações é que ele tenha qualidade, em especial que não contenha redundâncias. Trechos de código que seriam idênticos entre as transformações devem ser isolados em componentes fora delas e apenas ser invocados pelo código gerado pela transformação. Estes podem ser componentes de interface com o usuário, de entidade, de controle ou até mesmo utilitários. O importante é que tenham o código-fonte aberto para que a aplicação seja mais facilmente estendida, se necessário. Conforme elucidado nas boas práticas do MDSD, geração de código-fonte que não segue este principio tende a ser complexa, de difícil manutenção e extensibilidade.

Neste trabalho, também foi mostrado que o MDE e o MDI do Praxis têm conceitos parecidos ao PIM e ao PSM do MDA, respectivamente. Por isso, desde que no PIM sejam contemplados conceitos de interface com o usuário como o PIM UI faz, a inclusão da abordagem MDGI dentro do processo Praxis é praticamente direta, embora os modelos tenham que ser alterados para expressar tudo que é necessário para a geração de modelos e código. Dessa forma, os modelos e transformações propostos neste trabalho podem ser usados diretamente no processo Praxis.

5.1 Proposta de trabalhos futuros

Conforme exposto na seção 1.3 (Limites do trabalho), o MDA é um assunto complexo e, portanto, tem muitos aspectos que geram grandes discussões. Os próprios limites impostos a este trabalho podem gerar trabalhos futuros. Além destes, proporemos, a seguir, outros possíveis desdobramentos.

Os resultados desta pesquisa são baseados em dados de um experimento isolado e com as variáveis controladas. Embora esse experimento tenha permitido criar os modelos necessários, validá-los informalmente e colher alguns dados, é importante ver como esses modelos funcionariam na prática. Portanto, a fim de verificar se as conclusões obtidas são confirmadas, sugerimos fazer uma avaliação utilizando as recomendações canônicas da Engenharia de *Software* experimental [34] ou aplicar a abordagem proposta em projetos reais usando a metodologia pesquisa-ação [30]. Lembramos que o projeto deve satisfazer as questões levantadas na conclusão em relação à reutilização das transformações, caso contrário, com certeza, os resultados serão negativos.

Outra proposta de trabalho seria definir mecanismos que permitam que o desenvolvedor adicione detalhes aos modelos e lógica ao código-fonte gerados sem que estas alterações sejam perdidas ao regerá-los. Estes mecanismos são importantes para permitir um desenvolvimento mais centrado no modelo.

A transformação de PIM *Domain* para PIM UI descrita neste trabalho aplica apenas o padrão Caso de uso CRUD. Apesar do CRUD ser um padrão bastante utilizado, uma pesquisa futura poderia ser a identificação de outros padrões a serem aplicados ao PIM *Domain* para gerar o PIM UI, implementar estes padrões, fazer um experimento de utilização deles pela execução das transformações e analisar os resultados.

Um aspecto importante não detalhado aqui foi o de como mapear a associação entre entidades para a interface com o usuário. Existem várias possibilidades e formas de fazer esse mapeamento. Cada tipo de associação pode ter um diferente mapeamento, por exemplo, como seria o de uma associação simples, de agregação ou de composição. Além disso, podemos fazer mapeamento de formas diferentes dependendo da cardinalidade da associação, se é um para um, um para muitos ou muitos para muitos. Isso seria implementado pela transformação de PIM *Domain* para PIM UI. Uma extensão deste trabalho poderia começar pelo detalhamento desse mapeamento.

Referências bibliográficas

[1] Ahmed, K.; Umrysh, C. *Developing Enterprise Java Applications with J2EE and UML*, Addison-Wesley, 2002.

[2] Bauer, C.; King, G. *Hibernate in action*, Manning, 2005.

[3] Bettin, J. *Model-Driven Architecture – Implementation & Metrics*. Disponível em: http://www.softmetaware.com/mda-implementationandmetrics.pdf, Agosto, 2003, último acesso em fevereiro de 2005.

[4] Bettin, J. *Model-Driven Software Development, Versão 0.8*. Disponível em: http://www.softmetaware.com/mdsd-and-isad.pdf, Junho, 2004, último acesso em fevereiro de 2005.

[5] Bray, T.; Paoli, J.; Sperberg-McQu4een, C. M.; Maler, E. *Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation*. Disponível em: http://www.w3.org/TR/2000/REC-xml-20001006/, Outubro, 2000, último acesso em fevereiro de 2005.

[6] Courbis, A.; Lahire, P.; Parigot, D. *To build open and evolutive applications: an approach based on MDA and Generative Programming*, Technical report, INRIA, 2003.

[7] Czarnecki, K.; Eisenecker, U. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, Jun 2000.

[8] Dumoulin, C. *Tiles Advanced Features*, Novembro de 2002. Disponível em: http://www.lifl.fr/~dumoulin/tiles/tilesAdvancedFeatures.pdf, último acesso em novembro de 2005.

[9] Fowler, M. *Refactoring Home Page*. Disponível em: http://www.refactoring.com, último acesso em fevereiro de 2005.

[10] Budinsky, F.; Steinberg, D.; Ellersick, R.; Merks, E.; Brodsky, S.; Grose, T. *Eclipse Modeling Framework*, Addison-Wesley Professional, 2003.

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

[12] Gornik, D. *IBM Rational Unified Process: Best Practices for Software Development Teams*, IBM Corporation, 2001. Disponível em: http://www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/rup_bestpractices.pdf, último acesso em julho de 2005.

[13] Herst, D.; Roman, E., *Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) - Approach: A Productivity Analysis*, TMC Research Report, June 2003.

[14] Jacobson, I.; Rumbaugh, J.; Booch, G., *The Unified Software Development Process*, Addison Wesley, 1999.

[15] Jung, C. F. *Metodologia para pesquisa & desenvolvimento: aplicada a novas tecnologias, produtos e processos*. Rio de Janeiro/RJ: Axcel Books do Brasil Editora, 2004.

[16] Kleppe, A.; Warner, J.; Bast, W. *MDA explained, the Model Driven Architecture: Practise and Promise*, Addison-Wesley, 2003.

[17] Koch, N., Baumeister, H. and Mandel, L., *Extending UML to Model Navigation and Presentation in Web Applications*. In Modeling Web Applications, Workshop of the UML´2000. Ed. Geri Winters and Jason Winters, York, England, Outubro, 2000.

[18] Miller, J.; Mukerji, J. *MDA Guide Version 1.0.1*, document Number: omg/2003-06-01. Disponível em: http://www.omg.org/docs/omg/03-06-01.pdf, Junho, 2003, último acesso em fevereiro de 2005.

[19] Nunes, N.; Cunha, J. *Wisdom: A Software Engineering Method for Small Software Development Companies*. IEEE Software, vol. 17, no. 5, pp. 113-119, Sept/Oct, 2000.

[20] Paula, W. *Engenharia de Software: fundamentos, métodos e padrões*. LCT, 2ª edição, 2002.

[21] Pinheiro da Silva, P.; W. Paton, N. *User Interface Modeling in UMLi*. IEEE Software, Vol.20 No. 4, July/August 2003, pages 62-69.

[22] Pinheiro da Silva, P. *User Interface Declarative Models and Development Environments: A Survey*. In Proceedings of DSV-IS 2000, LNCS, Limerick, Ireland, June 2000. Springer-Verlag.

[23] Pleumann, J.; Haustein, S. *A Model-Driven Runtime Environment for Web Applications*, Sixth International Conference on the Unified Modeling Language - the Language and its applications, 2003.

[24] Rosenberg, D. *Use Case Driven Object Modeling with UML: A practical Approach*, Third Edition, Addison-Wesley, 1999.

[25] Rumbaugh, J.; Jacobson, I.; Booch, G., *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.

[26] Santos, H.; Barros, R. *Utilizando o MOF na construção de metamodelos em um ambiente MDA*. SBES, Brasília, 2004.

[27] Schattkowsky, T.; Lohmann, M. *UML Model Mappings for Platform Independent User Interface Design*, MoDELS Satellite Events 2005: 201-209.

[28] Schlee, M., *Generative Programming of Graphical User Interfaces*, Diploma Thesis, University of Applied Sciences of Kaiserslautern, 2002 http://citeseer.csail.mit.edu/schlee02generative.html

[29] Singh, I.; Stearns, B.; Johnson, M. *Designing Enterprise Applications with the J2EE™Platform*, Second Edition, Addison-Wesley, 2002.

[30] Thiollent, M. *Metodologia da pesquisa-ação*. 13ª edição, Cortez, 2004.

[31] Van Harmelen, M. *Object Modeling and User Interface Design: Designing Interative Systems*, Addison-Wesley, 2001.

[32] Vanderdonckt, J. *A MDA-Compliant Environment for Developing User Interfaces of Information Systems*, CAiSE 2005: 16-31

[33] Warner, J.; Kleppe, A. *Object Constraint Language: Getting Your Models Ready for MDA*, 2nd edition, Addison-Wesley, 2003.

[34] Wohlin, C.; Runeson, P.; Host, M.; Ohlsson, M.; Regnell, B.; Wesslen, A. *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, 2000.

[35] *Agile Alliance*. Disponível em: http://www.agilealliance.org/, último acesso em julho de 2006.

[36] Carnegie Mellon Software Engineering Institute. *Product Line Hall of Fame*. Disponível em: http://www.sei.cmu.edu/productlines/plp_hof.html, último acesso em julho de 2006.

[37] Carnegie Mellon Software Engineering Institute. *Software Product Lines*. Disponível em: http://www.sei.cmu.edu/productlines/index.html, último acesso em julho de 2006.

[38] Java Community Process. *JavaServer Pages™ 1.2 Specification*, Agosto de 2001. Disponível em: http://www.jcp.org/en/jsr/detail?id=53, último acesso em novembro de 2005.

[39] Object Management Group. *Data Warehousing, CWM™ and MOF™ Resource Page*. Disponível em: http://www.omg.org/cwm, último acesso em fevereiro de 2005.

[40] *Object Management Group*. Disponível em: http://www.omg.org, último acesso em fevereiro de 2005.

[41] Object Management Group. *Metamodel and UML Profile for Java and EJB, v1.0*, Fevereiro de 2004. Disponível em: http://www.omg.org/cgi-bin/doc?formal/2004-02-02, último acesso em julho de 2005.

[42] Object Management Group. *MetaObject Facility (MOF) Specification*. Disponível em: http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf, Abril, 2002, último acesso em fevereiro de 2005.

[43] Object Management Group. *MOF QVT Final Adopted Specification*. Disponível em: http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf, novembro, 2005, último acesso em abril de 2006.

[44] Object Management Group. *OMG Model Driven Architecture*. Disponível em: http://www.omg.org/mda, último acesso em fevereiro de 2005.

[45] Object Management Group. *UML Profile for CORBA, v 1.0*, Abril de 2002. Disponível em: http://www.omg.org/cgi-bin/doc?formal/02-04-01, último acesso em julho de 2005.

[46] Object Management Group. *UML Superstructure Specification, v2.0*, Agosto de 2005. Disponível em: http://www.omg.org/cgi-bin/doc?formal/05-07-04, último acesso em novembro de 2005.

[47] Object Management Group. *XML Metadata Interchange (XMI) Specification*. Disponível em: http://www.omg.org/cgi-bin/apps/doc?formal/03-05-02.pdf, Maio, 2003, último acesso em fevereiro de 2005.

[48] Rational Software Corporation. *Rational Rose User's Guide.* Disponível em: http://www.rational.com, último acesso em novembro de 2004.

[49] SourceForge.net. *XDoclet – Attribute Oriented Programming*, Novembro de 2005. Disponível em: http://xdoclet.sourceforge.net/xdoclet/index.html, último acesso em novembro de 2005.

[50] The Apache Software Foundation. *Struts 1.2.x*, Novembro de 2005. Disponível em: http://struts.apache.org/struts-doc-1.2.x/index.html, último acesso em novembro de 2005.

[51] The Eclipse Foundation. Eclipse Java Development Tools (JDT) Subproject, Disponível em: http://www.eclipse.org/jdt/, último acesso em dezembro de 2005.

Apêndice

A.1 Contagem de linhas

Nesta seção, apresentamos detalhes da contagem de linhas do código-fonte do Caso de uso de gestão de usuários do sistema Merci e das transformações implementadas neste trabalho. Para todas as contagens foi usada a mesma aplicação de contagem de linhas. A seguir são apresentadas a contagem total e a de linhas por diretório e por extensão de arquivos.

A.1.1 Contagem do código-fonte do sistema Merci

Primeiro apresentaremos a contagem do Caso de uso de gestão de usuário do sistema Merci gerado pelo MDGI. A seguir é mostrada a saída da aplicação de contagem de linhas para as classes da aplicação:

```
*** Contagem de: \IBM\rationalsdp6.0\runtime-workbench-workspace\MerciWEB\src
Ignorar comentários: true
Contagem total: 392
Contagem por diretório:
src = 68
src\base = 14
src\com\uhackers\merci = 310
Contagem por extensão:
.java = 324
.xml = 68
```

A seguir, é mostrada a saída da aplicação de contagem de linhas para os arquivos específicos da parte WEB:

```
*** Contagem de: \IBM\rationalsdp6.0\runtime-workbench-workspace\MerciWEB\WebContent
Ignorar comentários: true
Contagem total: 305
Contagem por diretório:
WebContent\WEB-INF = 305
Contagem por extensão:
.jsp = 175
.xml = 130
```

Somando 392 com 305, temos um total de 697 linhas de código para a abordagem usando o MDGI.

Em segundo lugar, apresentaremos a contagem do Caso de uso de gestão de usuário do sistema Merci desenvolvido usando o processo Praxis. A seguir, é mostrada a saída da aplicação de contagem de linhas:

```
*** Contagem de: \merci.DI\CFSw\src
Ignorar comentários: true
Contagem total: 689
Contagem por diretório:
src\com\uhackers\merci = 28
src\com\uhackers\merci\controle = 39
src\com\uhackers\merci\controle\administração = 76
src\com\uhackers\merci\entidade\administração = 129
src\com\uhackers\merci\fronteira = 28
src\com\uhackers\merci\fronteira\administração\usuário = 145
src\com\uhackers\merci\fronteira\principal = 244
Contagem por extensão:
.java = 689
```

Portanto, temos um total de 689 linhas de código para a abordagem usando o Praxis.

A.1.2 Contagem do código-fonte das transformações

A contagem de linhas das transformações foi separada em 3 módulos:

1. Regras de transformação
2. Elementos reutilizáveis entre as regras
3. Manipulação de XML (gerado automaticamente)

As linhas dos dois primeiros módulos foram contadas juntas e depois separadas por pacote. A contagem total dos dois módulos é apresentada a seguir.

Saída da aplicação de contagem de linhas para os arquivos do tipo *template*:

```
*** Contagem de: \TransfHibernate___\templates
Ignorar comentários: true
Contagem total: 148
Contagem por diretório:
templates\psmtosrc\struts = 148
Contagem por extensão:
.jet = 148
```

Saída da aplicação de contagem de linhas para as classes:

```
*** Contagem de: \TransfHibernate___\src
Ignorar comentários: true
Contagem total: 5,405
Contagem por diretório:
src\br\ufmg\dcc\gestus\anotacoesjava = 30
src\br\ufmg\dcc\gestus\anotacoesjava\hibernate2 = 46
src\br\ufmg\dcc\gestus\mda = 105
src\br\ufmg\dcc\gestus\mda\pimtopim\fronteira = 47
src\br\ufmg\dcc\gestus\mda\pimtopim\fronteira\regras = 371
src\br\ufmg\dcc\gestus\mda\pimtopsm\controle = 44
src\br\ufmg\dcc\gestus\mda\pimtopsm\controle\regras = 201
src\br\ufmg\dcc\gestus\mda\pimtopsm\dao = 52
src\br\ufmg\dcc\gestus\mda\pimtopsm\dao\regras = 317
src\br\ufmg\dcc\gestus\mda\pimtopsm\fronteira = 97
src\br\ufmg\dcc\gestus\mda\pimtopsm\fronteira\condicoes = 25
src\br\ufmg\dcc\gestus\mda\pimtopsm\fronteira\regras = 784
src\br\ufmg\dcc\gestus\mda\pimtopsm\hibernate = 109
src\br\ufmg\dcc\gestus\mda\pimtopsm\hibernate\condicoes = 145
src\br\ufmg\dcc\gestus\mda\pimtopsm\hibernate\regras = 561
src\br\ufmg\dcc\gestus\mda\pimtopsm\util = 288
src\br\ufmg\dcc\gestus\mda\pimtopsm\util\uml2 = 62
src\br\ufmg\dcc\gestus\mda\pimtopsm\validacao = 16
src\br\ufmg\dcc\gestus\mda\pimtopsm\validacao\regras = 355
src\br\ufmg\dcc\gestus\mda\psmtosrc\struts = 44
src\br\ufmg\dcc\gestus\mda\psmtosrc\struts\condicoes = 9
src\br\ufmg\dcc\gestus\mda\psmtosrc\struts\regras = 818
src\br\ufmg\dcc\gestus\mda\reuse\condicoes = 58
src\br\ufmg\dcc\gestus\mda\reuse\criacao = 327
src\br\ufmg\dcc\gestus\mda\reuse\estereotipo = 71
src\br\ufmg\dcc\gestus\mda\umltojava\transformations = 181
src\com\ibm\xtools\transform\samples\modeltomodel = 242
Contagem por extensão:
.java = 5,405
```

A seguir, é apresentada uma tabela com a separação em pacotes das linhas contadas referente ao primeiro e ao segundo módulos:

Pacote	Quantidade de linhas
1. Regras de transformação	
\mda\pimtopim\fronteira\regras	371
\mda\pimtopsm\controle\regras	201
\mda\pimtopsm\dao\regras	317
\mda\pimtopsm\fronteira\condicoes	25
\mda\pimtopsm\fronteira\regras	784
\mda\pimtopsm\hibernate\condicoes	145
\mda\pimtopsm\hibernate\regras	561
\mda\pimtopsm\validacao\regras	355
\mda\psmtosrc\struts\condicoes	9
\mda\psmtosrc\struts\regras	818
\mda\umltojava\transformations	181
\templates\psmtosrc\struts	148
Total	**3.915**
2. Elementos reutilizáveis entre as regras	
\anotacoesjava	30
\anotacoesjava\hibernate2	46
\mda	105
\mda\pimtopim\fronteira	47
\mda\pimtopsm\controle	44
\mda\pimtopsm\dao	52
\mda\pimtopsm\fronteira	97
\mda\pimtopsm\hibernate	109
\mda\pimtopsm\util	288
\mda\pimtopsm\util\uml2	62
\mda\pimtopsm\validacao	16
\mda\psmtosrc\struts	44
\mda\reuse\condicoes	58
\mda\reuse\criacao	327
\mda\reuse\estereotipo	71
\samples\modeltomodel	242
Total	**1.638**

Tabela 14 - Contagem de linhas dos módulos 1 e 2 das transformações

A contagem de linhas do terceiro módulo é apresentada a seguir. A saída da aplicação de contagem de linhas para as classes desse módulo foi:

```
*** Contagem de: \TransfHibernate___\src_xml
Ignorar comentários: true
Contagem total: 21,230
Contagem por diretório:
src_xml\br\ufmg\dcc\gestus\xml\binding\struts = 383
src_xml\br\ufmg\dcc\gestus\xml\binding\struts\impl = 11,282
src_xml\br\ufmg\dcc\gestus\xml\binding\struts\impl\runtime = 2,074
src_xml\br\ufmg\dcc\gestus\xml\binding\tiles = 203
src_xml\br\ufmg\dcc\gestus\xml\binding\tiles\impl = 5,214
src_xml\br\ufmg\dcc\gestus\xml\binding\tiles\impl\runtime = 2,074
Contagem por extensão:
.java = 21,230
```

Portanto, a contagem de linhas de cada módulo ficou da seguinte forma:

1. Regras de transformação: 3.915 linhas
2. Elementos reutilizáveis entre as regras: 1.638 linhas
3. Manipulação de XML (gerado automaticamente): 21.230 linhas