ALLAN GUERREIRO CARNEIRO

OBTENÇÃO DE MODELOS EXECUTÁVEIS DE PROCESSO DE SOFTWARE A PARTIR DA APLICAÇÃO DAS REGRAS DE TRANSFORMAÇÃO ENTRE MODELOS

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2011

ALLAN GUERREIRO CARNEIRO

OBTENÇÃO DE MODELOS EXECUTÁVEIS DE PROCESSO DE SOFTWARE A PARTIR DA APLICAÇÃO DAS REGRAS DE TRANSFORMAÇÃO ENTRE MODELOS

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 4 de fevereiro de 2011.

_____	_____
Prof. Alcione de Paiva Oliveira (Co-orientador)	Prof. Clarindo Isaías P. da Silva e Pádua

_____

Prof. José Luís Braga

(Orientador)

AGRADECIMENTOS

Agradeço a todas as pessoas que se fizeram presentes, que se preocuparam, foram solidárias e torceram por mim. Mas bem sei que agradecer é sempre difícil. Não gostaria de cometer nenhuma injustiça esquecendo pessoas que me ajudaram nesta etapa difícil e recompensadora da minha vida.

Entretanto, todos os que realizam um trabalho de pesquisa sabem que não o fazem sozinhos, existem diversas pessoas envolvidas e diversos fatores atenuantes, mesmo sendo um ato solitário de leitura e o do escrever. O resultado desses estudos se torna possível através da cooperação e do esforço de outros antes de nós.

Queria agradecer aos meus pais, irmão e familiares pelo incentivo, apoio, sem mencionar as cobranças.

Ao meu orientador Professor José Luis Braga, que foi de uma enorme paciência e tolerância as minhas falhas.

A Iulla Reis, por ter me ensinado que na vida que podemos aprender com os erros e crescer a partir deles.

Aos Professores Carlos de Castro Goulart e Mauro Nacif Rocha, pelas aulas e pelas recomendações ao programa de pós-graduação desta universidade maravilhosa.

Ao Professor Clayton Vieira Fraga Filho, por ter sido de imensa fonte de informação fornecendo a base de conhecimento para desenvolver este trabalho e por sua paciência e auxílio em todas as horas.

Ao Altino Alves de Souza Filho, secretário da Pós-graduação, por sua amizade, competência e ajuda em todos os trabalhos burocráticos do mestrado.

A UFV, e aos demais professores do Programa de Pós-Graduação em Ciência da Computação pelos momentos de aprendizado proporcionados.

Aos amigos do mestrado, pelas informações e por sua amizade.

A todos, muito obrigado!

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

RESUMO

CARNEIRO, Allan Guerreiro, M.Sc., Universidade Federal de Viçosa, Fevereiro de 2011. **Obtenção de modelos executáveis de processo de software a partir da aplicação das regras de transformação entre modelos.** Orientador: José Luis Braga. Co-Orientadores: Alcione de Paiva Oliveira e Vladimir Oliveira Di Iorio.

Identificar um modelo de processo de software, que possa melhorar a qualidade e a produtividade do desenvolvimento de software em uma organização, não é uma tarefa simples e direta, já que a realização de experimentos em computação é uma tarefa que consome muito tempo, gerando muitas despesas. A computação pode fazer uso de processos de simulação para testar hipóteses e experimentar a realidade evitando alto custo e perda desnecessária de tempo. A utilização de ambientes virtuais, que possam simular o aspecto real, está atraindo a atenção de diversos profissionais. Dessa forma, este trabalho teve como objetivo a transformação de um modelo, inicialmente, gerado de forma independente de tecnologia (e.g. modelos de classes UML) em modelos para uma tecnologia específica (e.g. Java). Para realizar a automação de um processo de desenvolvimento de software, foi desenvolvida uma aplicação Java que fornece um ambiente amigável, em que o usuário possa inserir os dados necessários, possibilitando a geração de uma simulação mais abrangente e completa. Entretanto, os resultados são exibidos através de um arquivo de texto contendo os passos da simulação, que é utilizado como entrada para outro aplicativo que interpreta e exibe o processo de simulação definido pelo usuário.

ABSTRACT

CARNEIRO, Allan Guerreiro, M.Sc., Universidade Federal de Viçosa, February 2011. **Obtaining executable models of software process based on the application of transformation rules between models.** Advisor: José Luis Braga. Co-Advisors: Alcione de Paiva Oliveira and Vladimir Oliveira Di Iorio.

Identify a process model that can improve quality and productivity of software development in an organization is not a straightforward task, since the experiments in a computation is a task that consumes a lot of time generating a lot of expenses. The computation can make use of process simulation to test hypotheses and try to avoid high-cost and unnecessary loss of time. The use of virtual environments that can simulate the real aspect is attracting the attention of many professionals. Thus, this study aimed to transformation a model, initially generated independently of technology (e.g. UML class models) to models for a specific technology (e.g. Java). To perform the simulation of a software development process, it was developed a Java application that provides a friendly environment in which the user can enter the necessary data, generating a broader and more complete simulation. However, the results are displayed trough a text file containing the steps of the simulation, which is used as input to another application that interprets and displays the simulation process defined by the user.

1 INTRODUÇÃO

A pesquisa empírica é essencial para o desenvolvimento de teorias de desenvolvimento de software ou de um processo de software, transformando a arte de desenvolvimento de software em uma disciplina de engenharia e, consequentemente, melhorando o desempenho global das atividades de desenvolvimento de software. Disciplinas de engenharia, por outro lado, exigem comprovações sobre a eficiência e a eficácia das ferramentas e técnicas em diferentes contextos de aplicação. Contudo, o número de ferramentas e técnicas está crescendo constantemente, e surgem cada vez mais ferramentas ou técnicas que podem ser aplicadas.

Ainda assim, a maioria das atividades em desenvolvimento de software é fortemente baseada nas pessoas, sendo que a real eficácia e efetividade de uma ferramenta ou técnica só podem ser determinadas através de experiências do mundo real.

Experimentos controlados são um meio para avaliar a eficiência e a eficácia local dos instrumentos ou técnicas. Os grandes problemas de se utilizar experimentos controlados envolvendo pessoas, ainda são custo e tempo. Por este motivo, as ferramentas de simulação estão ganhando cada vez mais espaço nas pesquisas.

De acordo com Park *et al.* (2007), existe um crescente interesse em melhorar a eficácia e a eficiência de um processo de software. Dessa forma, o gerenciamento de processos de software está se tornando uma das questões de maior importância.

As organizações têm procurado adotar a simulação de processos de software para gerenciar seus próprios processos. No entanto, é difícil adotar uma tecnologia de simulação. Uma das razões é que os modelos de simulação tendem a ser difíceis de serem construídos e mantidos (RAFFO, 2003).

Kellner (1999) acredita que o Modelo de Simulação Processo de Software começa a ser utilizado para tratar uma variedade de problemas, desde a gestão estratégica de desenvolvimento de software, como a melhoria dos processos de apoio e treinamento para o gerenciamento de projetos de software.

O Modelo de Simulação de Processo de Software desperta muito interesse entre pesquisadores e profissionais, como uma abordagem para a análise de negócios complexos e questões políticas. Embora os modelos de simulação tenham sido

aplicados em uma variedade de disciplinas ao longo dos anos, só recentemente foi aplicada à área de desenvolvimento de software.

A percepção geral é que as ferramentas de Simulação de Processo de Software são caras para construir e os resultados, em tempo útil, são difíceis de alcançar. Estas ferramentas podem ser desenvolvidas em diferentes níveis de abrangência e profundidade para atender às necessidades de uma organização. Quanto maior o nível de detalhes e fidelidade ao processo, maior a quantidade de esforço necessário para construir uma ferramenta específica.

Além disso, a maioria das ferramentas de simulação são muito gerais e não são adaptadas para atender aos requisitos de software de modelagem de processos. Esses requisitos incluem conceitos como modularização e reuso.

Quando se modela um sistema, principalmente, nos dias de hoje com o paradigma da Orientação a Objetos, é especificado um problema do mundo real da forma mais abstrata possível, e para isso existem técnicas formais e semi-formais. Com esse modelo definido o mais próximo da realidade, serão necessários diversos processos de refinamento para que se obtenha uma aplicação automatizada funcional.

Ainda é muito difícil descrever corretamente um problema do mundo real utilizando modelos. Por isso, diversas técnicas vêm sendo aprimoradas com exatidão matemática permitindo a verificação de propriedades desejadas no modelo. Pesquisadores estão trabalhando, por exemplo, com Máquinas de Estados Abstratas (ASM) para poder simular o comportamento de um modelo escrito de forma formal ou semi-formal. As ASM são máquinas de estado abstratas, onde os estados são álgebras e as mudanças de seus estados são as regras de transição que modificam a interpretação de funções de um determinado estado.

Com a utilização de um método que possa abstrair o máximo de detalhes e que tenha precisão matemática, diversos pesquisadores tentam implementar modelos de simulação para processo de software, com o objetivo de visualizar a execução antes da mesma ser colocada em prática. Esse é um assunto muito discutido, sendo criado até mesmo um Workshop de *Software Process Simulation Modeling* (ProSim) para mostrar a importância de se desenvolver modelos de simulação de processos. Em 2003, um ano marcado na história da computação com o surgimento da linguagem ASM.

1.1 O Problema e sua Importância

Como um esforço para trazer disciplina para a atividade de desenvolvimento de software, muitas organizações, na década de 70, começaram a aplicar os rigores da ciência e da engenharia em seus projetos de software. Surgiram, então, novas técnicas: projeto e programação estruturada, verificação formal, linguagens amigáveis, programação orientada a objetos, reutilização e outras que trouxeram ganhos para a indústria.

No entanto, Abdel-Hamid e Madnick (1991), acreditam que pouca atenção foi dada pela comunidade científica aos aspectos gerenciais do processo de desenvolvimento de software, e há necessidade de novas contribuições nessa área. A consequência dessa deficiência é a falta de entendimento do processo de gerenciamento de projetos. A falta de investimentos em pesquisa para aprofundar o conhecimento nessa área é considerada por alguns pesquisadores (Abdel-Hamid e Madnick, 1991) a responsável pela persistente dificuldade em produzir software.

Dessa forma, buscando melhorar a qualidade de software, a Engenharia de Software tem produzido ferramentas para auxílio ao desenvolvimento de software, assim como tem estudado e produzido formas de controlar o processo de desenvolvimento.

Brown (2004) afirma que os modelos elevam o nível de abstração do desenvolvimento dos sistemas, ajudando a planejá-los e entendê-los. Entretanto, existe um grande desafio no uso de modelos, já que é necessário sincronizar o código com o próprio modelo. Sendo assim, diversos enfoques estão sendo criados para resolver o problema da sincronia do software com o modelo. Uma delas é a geração automática de código a partir do modelo do sistema.

Para gerar automaticamente código a partir de um modelo, é preciso realizar um mapeamento do que será devidamente transformado. Para formalizar este procedimento, a *Object Management Group* (OMG) propôs uma solução através da arquitetura *Model Driven Architecture* (MDA), onde o processo de desenvolvimento de software deve ser direcionado pela atividade de modelagem do sistema, em nível conceitual, independente de qualquer plataforma ou implementação, através de transformações realizadas sobre esse modelo conceitual.

1.2 Hipótese de Trabalho

A geração de modelos executáveis de processos de software é beneficiada se for obtida a partir da definição e especificação de modelos semi-formais e das regras de mapeamento entre eles.

1.3 Objetivos

O objetivo deste trabalho é mostrar que a especificação semi-formal de modelos abstratos de processos de desenvolvimento de software favorece a qualidade do modelo executável final.

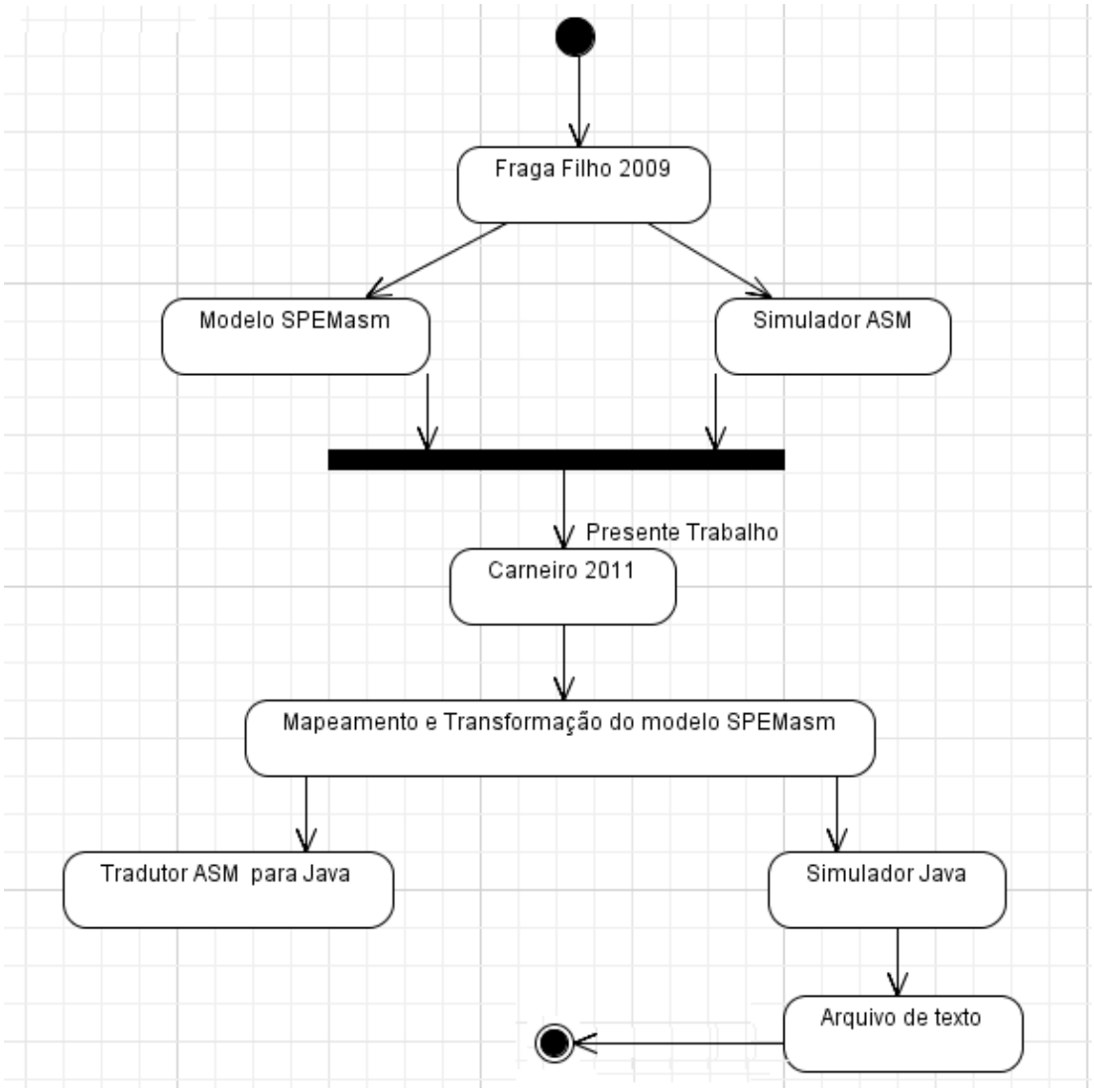
Os objetivos específicos são:

1. Determinar as linguagens que permitam a definição semi-formal de modelos de processos nos níveis Modelo Independente de Plataforma (PIM) e Modelo Específico de Plataforma (PSM) da arquitetura MDA;
2. Definir e especificar regras de transformação de modelos (PIM), com foco em pelo menos um (PSM) específico.
3. Demonstrar a especificação e a verificação destes modelos utilizando mapeamento de instâncias com marcas ou outra técnica;
4. Validar o modelo gerado pela transformação por meio de um estudo de caso, incluindo a execução do modelo final obtido.

1.4 Estrutura da Dissertação

Este trabalho tem como contribuição a realização um mapeamento conceitual do simulador, desenvolvido por Fraga Filho (2009), para um novo simulador, que deverá ser desenvolvido em Java. Este simulador deve produzir uma saída, que pode ser compatível com o formato gerado pelo simulador, desenvolvido por Fraga Filho (2009), para ser reconhecido pela ferramenta SPM Simulator.

A Figura 1.1 descreve, em síntese, o que foi realizado por Fraga Filho em (2009) e a proposta deste trabalho.



Fonte: Elaborado pelo autor.

Figura 1.1 Contribuição da Pesquisa.

Buscando alcançar os objetivos propostos neste trabalho, foram realizadas as seguintes atividades:

1. O Problema e a sua importância: através de pesquisas realizadas no Laboratório de Engenharia de Software (LES), da Universidade Federal de Viçosa (UFV), em 2009, foi identificada a necessidade de criar um simulador de processos de software, a partir de regras de mapeamento e transformação. Sendo assim, este trabalho realiza o mapeamento e a transformação de um modelo, visando à construção de um simulador de processos de software.

2. Contexto: esta etapa fornece os meios para o entendimento dos assuntos utilizados neste trabalho. Foram exploradas as áreas Simulação de

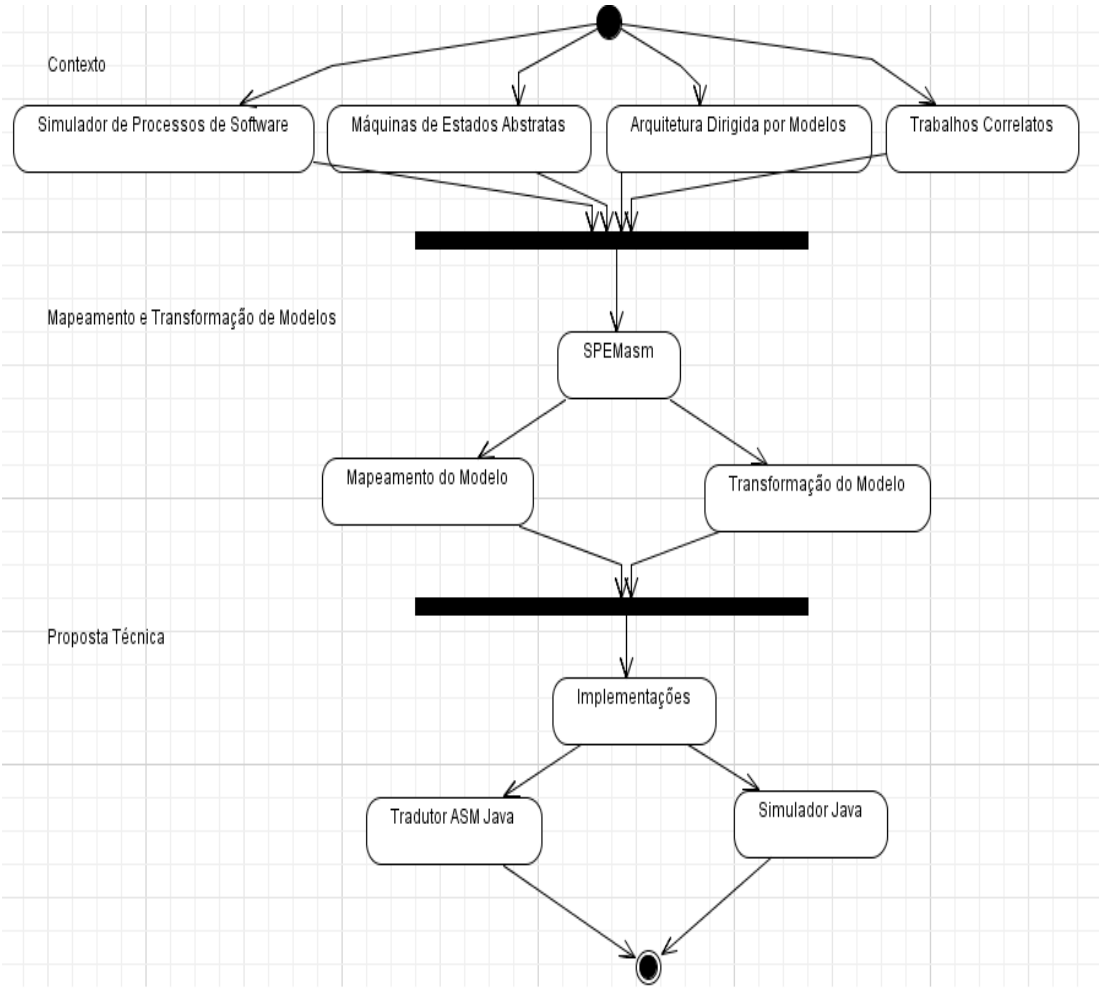
Processos de Software, Máquinas de Estado Abstratas e Arquitetura Dirigida por Modelos.

3. Transformação e Mapeamento do Modelo: para utilizar a metodologia de Arquitetura Dirigida por Modelos (MDA), era necessário ter um Modelo Independente de Plataforma (PIM). Este modelo foi proposto por Fraga Filho (2009), e a partir desse modelo foi gerado um Modelo Específico de Plataforma (PSM), utilizando a linguagem de programação Java.

4. Estudo de caso: foi criado para utilizar o simulador de processos, desenvolvido através do mapeamento e transformação.

5. Considerações Finais e Trabalhos Futuros: apresenta uma síntese das contribuições do trabalho e sugestões de trabalhos futuros.

A Figura 1.2 ilustra a estrutura da dissertação descrita anteriormente.



Fonte: Elaborada pelo autor.

Figura 1.2 Estrutura da Dissertação.

2 CONTEXTO

Este capítulo tem por objetivo apresentar, de forma sucinta, os principais conceitos relacionados a Modelos de Simulação de Processo de Software, Máquinas de Estado Abstratas e do Processo Dirigido por Modelos (MDA).

2.1 Modelos de Simulação de Processos de Desenvolvimento de Software

Ambientes de simulação têm como objetivo permitir a geração de cenários através de um aplicativo, com os propósitos de orientar o processo de tomada de decisão, proceder a análises e avaliações de sistemas e propor soluções para a melhoria de performance.

A informática teve avanços extraordinários, entre eles, equipamentos mais modernos e novas linguagens de programação. Com essas novas linguagens de programação, a área de simulação tem permitido empregar a técnica de simulação nas diversas áreas do conhecimento humano, fatos que têm propiciado:

- Projetar e analisar sistemas industriais;
- Avaliar desempenho de hardware e software em sistemas de computação;
- Analisar desempenho de armas e estratégias militares;
- Determinar frequência de pedidos de compra para recomposição de estoques;
- Projetar e administrar sistemas de transportes, como portos e aeroportos;
- Configurar sistemas de atendimento em hospitais, supermercados e bancos.

No caso específico das engenharias, a adoção da técnica de simulação tem trazido benefícios, como:

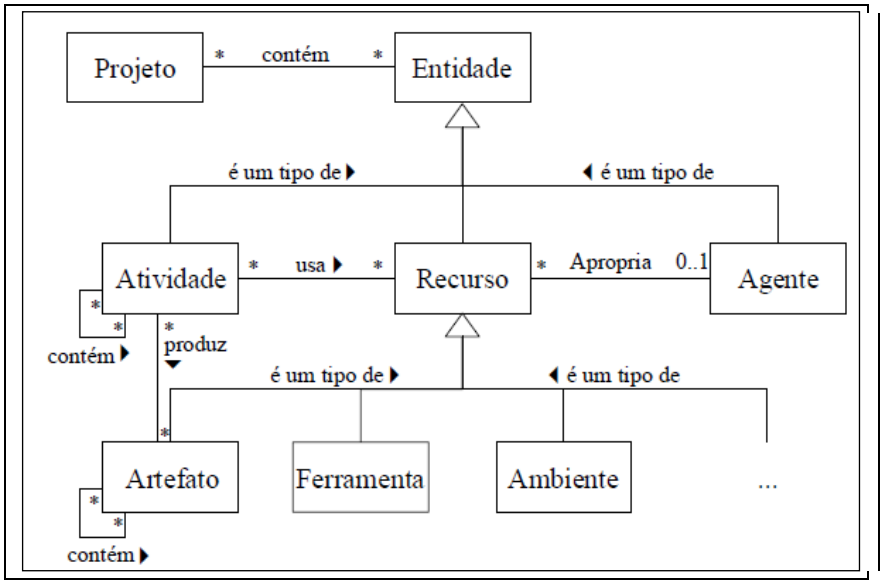
- A previsão de resultados na execução de uma determinada ação;
- A redução de riscos na tomada decisão;
- A identificação de problemas antes mesmo de suas ocorrências;
- A eliminação de procedimentos em arranjos industriais que não agregam valor a produção;

- A realização de análises de sensibilidade;
- A redução de custos com o emprego de recursos (mão-de-obra, energia, água e estrutura física);
- A revelação da integridade e viabilidade de um determinado projeto em termos técnicos e econômicos.

A simulação na área da engenharia de software tem grande importância, devido ao fato de que o software é um negócio competitivo, e os principais direcionadores que propiciam uma intensa competição na área de software são: custo, adequação de prazo e qualidade. Todos esses fatores intensificam-se, portanto, em uma rápida movimentação dos desenvolvedores para adotar práticas modernas de Engenharia de Software.

2.1.1 Processo de Desenvolvimento de Software

Um processo de desenvolvimento de software corresponde ao conjunto de fases relacionadas e necessárias executadas pelos desenvolvedores, desde a concepção até a liberação do produto. Requer a organização lógica de diversas atividades técnicas e gerenciais envolvendo agentes, métodos, ferramentas, artefatos e restrições, que possibilitam disciplinar, sistematizar e organizar o desenvolvimento e manutenção de produtos de software, conforme ilustra a Figura 2.1.



Fonte: (SOMMERVILLE, 2007).

Figura 2.1 Componentes de um processo de software.

De acordo com Fiorini (2001), um processo de software, quando bem definido, permite o acompanhamento dos resultados obtidos ao longo do ciclo de vida do software, além de prever resultados, estimar projetos e prover melhoria e qualidade. Conhecer os processos significa conhecer como os produtos são planejados e produzidos.

Para que possamos ter um processo de software bem definido, será necessário considerar alguns fatores, em particular, que estejam relacionadas à organização desenvolvedora de software e ao próprio sistema desenvolvido. Dentre essas particularidades, destacam-se:

- Adequação às tecnologias envolvidas;
- Tipo de software a ser desenvolvido;
- Domínio da aplicação;
- Grau de maturidade (ou capacitação);
- Características da equipe.

Essa quantidade e diversidade de variáveis leva ao entendimento de que, para cada projeto de desenvolvimento de software, é necessário que processos com características específicas sejam definidos.

O amadurecimento em processo de software é longo e custa caro. Entretanto, para analisar e amadurecer tal processo é necessária uma descrição, a qual consiste de um modelo. Um modelo é uma abstração da realidade usada para ajudar a compreender o objeto ou sistema que está sendo modelado. A modelagem é usada todo o tempo para tomar decisões, embora, geralmente, esta modelagem não seja feita de maneira formal.

Identificar o número de variáveis a serem inseridas no modelo, e definir qual o grau de robustez e complexidade que o modelo deve ter, são questões cujas respostas não são simples. Sendo assim, o uso de modelos torna-se essencial para que o gestor possa testar alternativas e tomar a decisão mais precisa. Para que os modelos possibilitem a tomada de decisão, é necessário que a etapa da modelagem seja executada da melhor forma possível.

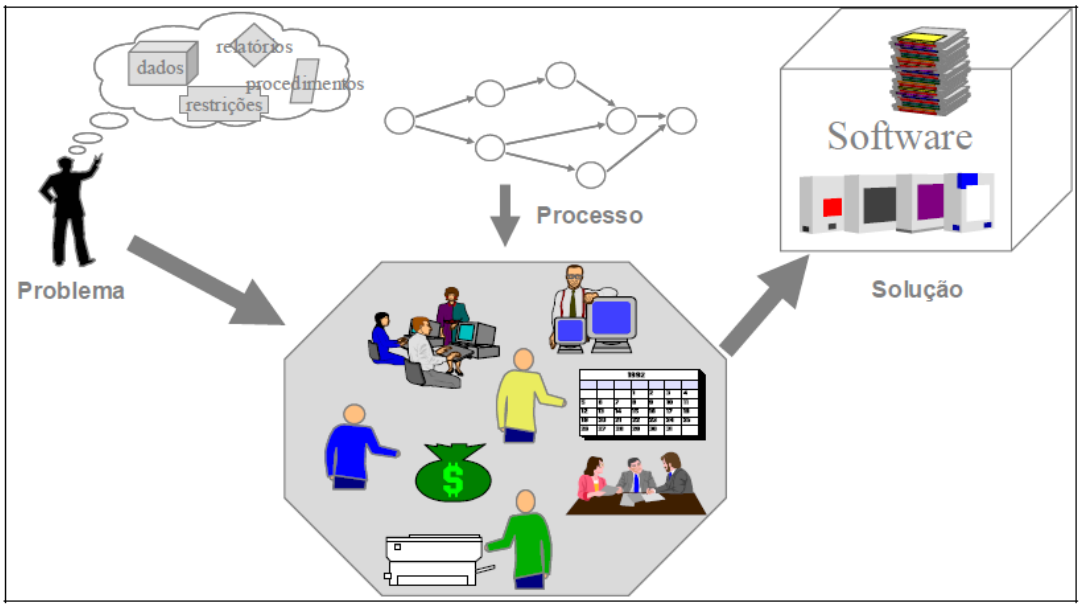
Com um modelo de processo pronto, é possível controlar a execução dos passos de forma automatizada. Na fase de execução de modelos de processo de software, devem ser levadas em consideração as questões de coordenação de

múltiplos usuários e a interação entre as ferramentas automatizadas e os desenvolvedores.

Modelos de processo de software são como perspectivas, sendo elas categorizadas em:

- Funcional;
- Organizacional;
- Comportamental;
- Informacional.

A categoria funcional possui componentes do processo que estão sendo realizados e os fluxos de produtos que são relevantes para esses componentes. A organizacional representa onde e por quem na organização os componentes do processo são realizados. A comportamental representa o tempo em que os componentes do processo são realizados. A informacional contém informações para a representação dos produtos produzidos ou manipulados pelos componentes do processo (CURTIS, 1992). Os componentes de um processo de software são ilustrados na Figura 2.2.



Fonte: (REIS, 2002).

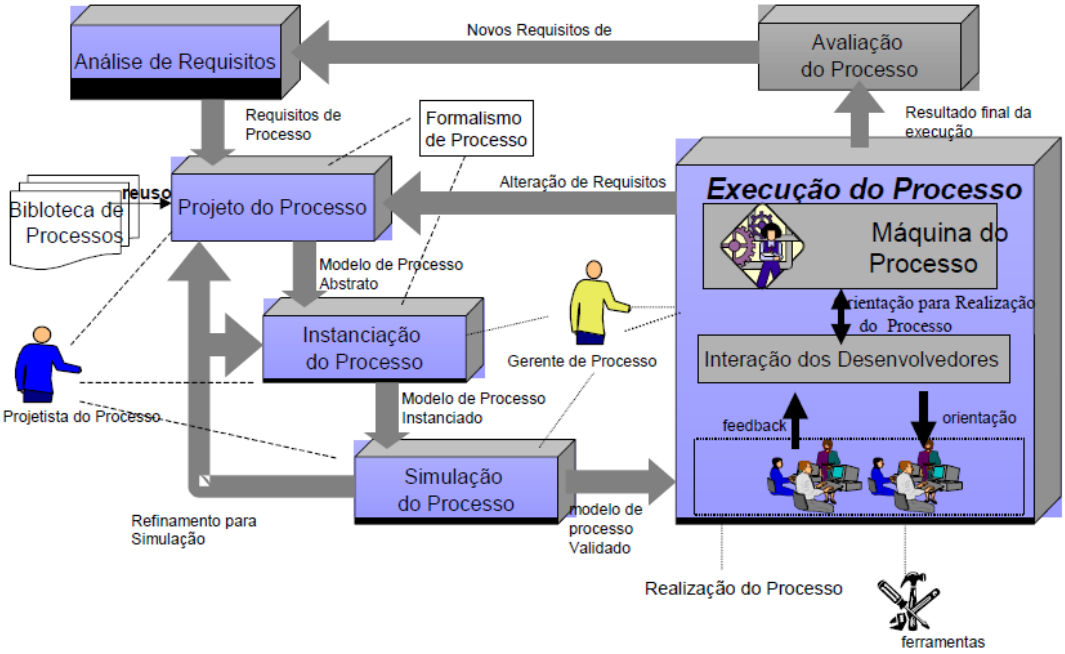
Figura 2.2 *Stakeholders* de um processo de software.

O ciclo de vida do processo de desenvolvimento de software é uma caracterização descritiva ou prescritiva de como o software é, ou de como deve ser

desenvolvido. Essas caracterizações sugerem que há uma variedade de propósitos para a articulação de modelos de ciclo de vida de software. Estas caracterizações servem de:

- Orientação para organizar e planejar: pessoal, orçamento, cronograma e gerir o trabalho de projeto de software, ao longo do tempo organizacional, espaço e ambientes de computação;
- Base para determinar quais ferramentas de engenharia de software e metodologias são mais adequadas e eficazes para apoiar as atividades do ciclo de vida diferente.
- Quadro de análise ou estimativa padrão de alocação de recursos e de consumo durante o ciclo de vida do software.
- Base para a realização de estudos empíricos para determinar o que afeta a produtividade de software, custo e qualidade global.

Existe um ciclo de vida para o desenvolvimento de processos de software, conforme mostra a Figura 2.3. Os retângulos representam atividades que constituem o meta-processo de software e as setas indicam o fluxo de modelos de processo, em diferentes níveis de detalhe. Nesse ciclo, a execução de processos opera sobre um modelo previamente validado e que tenha sido continuamente aperfeiçoado, desde a análise dos seus requisitos até a sua instanciação. Durante a execução, o processo pode ser alterado ou pode estar incompleto, então, as fases de projeto, instanciação e simulação podem ocorrer diversas vezes com o objetivo de aperfeiçoar o processo adotado.



Fonte: (LIMA, 1998).

Figura 2.3 Meta-modelo de processo de software.

Para Mendez (2005), em desenvolvimento de software, devem ser consideradas diferentes variáveis para definir um processo, entre elas:

- Modelo de ciclo de vida, métodos e ferramentas utilizadas pelo projeto;
- Os recursos humanos e materiais e suas responsabilidades dentro do processo;
- Artefatos consumidos e produzidos.

A influência desses fatores é refletida na adaptação do processo especializado para o projeto, o que nada mais é que a sua instanciação.

2.1.2 Modelagem de Processo de Software

A modelagem de processos fornece visão sistêmica aos gestores, permitindo que os gerentes possam compreender como o trabalho é executado e como suas tarefas influenciam no cliente final (VERNADAT, 1996).

A modelagem de processos precisa contemplar um grau adequado de clareza e formalização em seus níveis de abstração, ou seja, prover o conhecimento de seu

funcionamento, alinhando-o com a estratégia da organização e propondo a melhoria contínua dos seus processos de software.

Becker e Rosemann (2000) acreditam que os princípios e critérios geralmente aceitos em modelagem de processos são: aderência; relevância ou suficiência; custo/benefício; clareza; comparabilidade; estruturação sistemática.

A modelagem pode ser uma forma de apoiar a organização a explicitar e formalizar os seus processos de negócio, a fim de promover a representação e entendimento das atividades de trabalho, para minimizar problemas e maximizar a produção de produtos/serviços.

Vernadat (1996) afirma que os seguintes pontos são apresentados como motivação para a modelagem:

- Gestão de sistemas complexos; melhor gestão de todos os tipos de processos;
- Explicitação do conhecimento e *know how* organizacional;
- Reengenharia de processos;
- Integração empresarial propriamente dita.

Um dos objetivos da modelagem de processo é decompor estas descrições em detalhes suficientes para orientar a execução do processo (GIMENES *et al*, 2000).

De acordo com o Instituto de Engenharia de Software (SEI), a modelagem de processo de software deve possuir competências em três categorias principais:

- Representação;
- Análise global;
- Previsão.

Uma poderosa representação formal é necessária para lidar com as complexidades reais dos processos organizacionais. É importante ser capaz de representar as descrições atuais do processo, as prescrições de futuros processos desejados e restrições impostas pelos regulamentos e normas.

A capacidade de uma análise mais abrangente deve incluir uma larga variedade de testes nas áreas de consistência, integralidade e exatidão. Estes são fundamentais para determinar a validade do próprio modelo e do processo do mundo real que o modelo representa.

A capacidade de previsão inclui aspectos qualitativos e quantitativos. Testes qualitativos servem para analisar o comportamento do processo em resposta a vários

eventos e circunstâncias. Testes quantitativos podem prever os resultados numéricos ao longo das dimensões, como tempo para conclusão, necessidades de recursos humanos, ou medidas de qualidade (KELLNER, 1988).

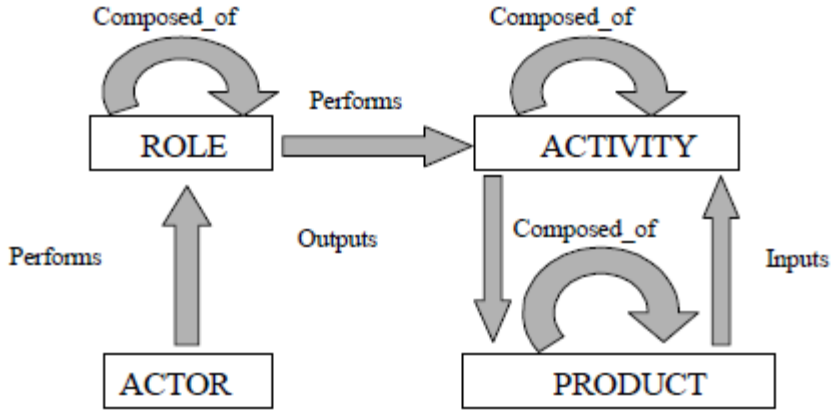
Como a modelagem de processos de software é um conjunto de atividades e resultados, que tem por finalidade a geração de um produto de software, Sommerville (2007) afirma que essas atividades são organizadas e estruturadas, podendo ter como entrada algum artefato, e devem gerar pelo menos um artefato, sendo executadas por um agente humano ou computacional. Em geral, devem-se considerar os seguintes pontos sobre um processo de software:

- Recursos - que são necessários para a execução das atividades;
- Atividades - que são executadas pelos recursos e geram algum artefato;
- Artefatos - que são produtos gerados pelas atividades, e que podem ser também matéria-prima para outras atividades do processo.

Entretanto, Acuña (2005) define uma visão com o mesmo propósito, mas com elementos diferentes no processo de desenvolvimento de software:

- Agente ou ator - é uma entidade que executa um processo.
- Papel - descreve um conjunto de agentes ou grupo de responsabilidades, direitos e competências necessárias para executar uma atividade do processo de software específico.
- Atividade - é a etapa de um processo que produz mudanças visíveis externamente no estado do produto de software.
- Artefato ou produto - é o produto ou subproduto, é a "matéria prima" de um processo.

Os elementos definidos por Acuña (2005) estão descritos e relacionados na Figura 2.4, a qual representa os relacionamentos entre os componentes envolvidos em um processo de software.



Fonte: (ACUÑA, 2005).

Figura 2.4 Processo simples de modelagem de componentes.

A modelagem de processo de software tem como objetivos principais (ARMENISE, 1992; CURTIS, 1992):

- Facilitar a comunicação e compreensão entre as pessoas;
- Facilitar o aperfeiçoamento do processo;
- Reutilização;
- Fornecer gerência do processo;
- Prover orientação automatizada do processo;
- Prover execução automatizada.

Modelagem de processos de software pode ser representada através de formalismos tradicionais da Engenharia de Software como diagramas UML, notação SPEM (*Software Process Engineering Metamodel*), modelagem BPMN (*Business Process Modeling Notation*), entre outros.

2.2 Máquinas de Estado Abstratas (ASM)

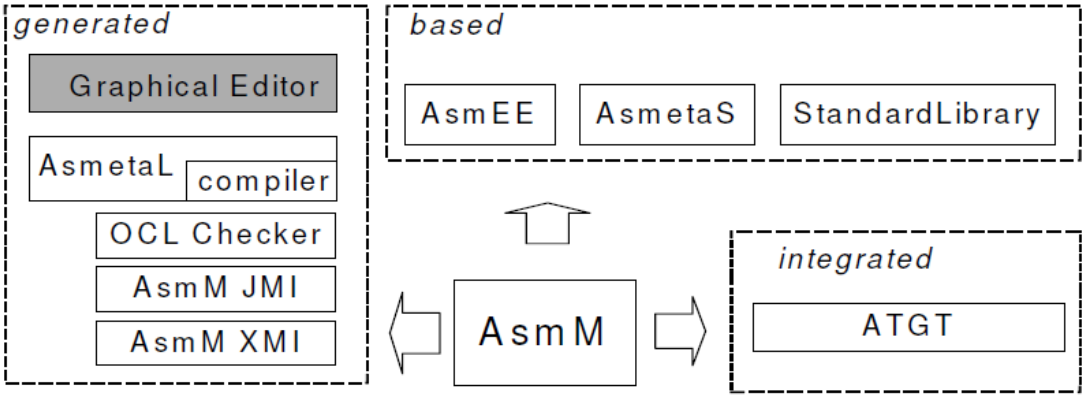
Há mais de 15 anos, *Abstract State Machines* (ASM) têm sido estudadas, praticadas e aplicadas na modelagem e especificação de sistemas, criando um forte relacionamento entre as abordagens formais e pragmáticas. ASM tornou-se uma técnica bem conhecida, e assumiu um papel importante no fornecimento de uma estrutura sólida e flexível para especificação e modelagem de praticamente todos os tipos de sistemas dinâmicos discretos. Portanto, com uma demanda crescente na confiabilidade dos sistemas de software e hardware, a utilização da ASM facilita a

validação e verificação de especificações abstratas, provando ser benéfico em todas as fases do processo de desenvolvimento.

A abstração e a formalização fornecem instrumentos eficazes para o estabelecimento de requisitos de sistema em sistemas de modelagem. Toda esta abstração é usada antes da construção, possibilitando analisar e calcular as especificações e opções de design e melhor compreender as suas implicações (BERRY, 2002).

2.2.1 O Projeto ASMETA

O projeto Asmeta possui um meta-modelo chamado AsmM, que foi desenvolvido seguindo as diretrizes da Engenharia Dirigida por Modelos (MDE). Esse conjunto de ferramentas, desenvolvido pelo projeto Asmeta, pode ser representado graficamente pela Figura 2.5 (GARGANTINI *et al*, 2007).



Fonte: (GARGANTINI *et al.*, 2007).

Figura 2.5 O Framwork Asmeta.

Gargantini *et al.*(2007) descrevem a Figura 2.5 da seguinte forma:

- O meta-modelo AsmM é baseado no *MetaObject Facility* (MOF) 1.4 da fundação *Object Management Group* (OMG), que representa conceitos em uma forma abstrata e construções do formalismo ASM, como descrito por Boerger e Staerk (2003).
- O verificador *Object Constraint Language* (OCL) do AsmM é baseado na ferramenta OCLE, e usado para verificar se um determinado modelo está bem especificado, ou não, no que diz respeito às restrições OCL definido sobre o meta-modelo AsmM.

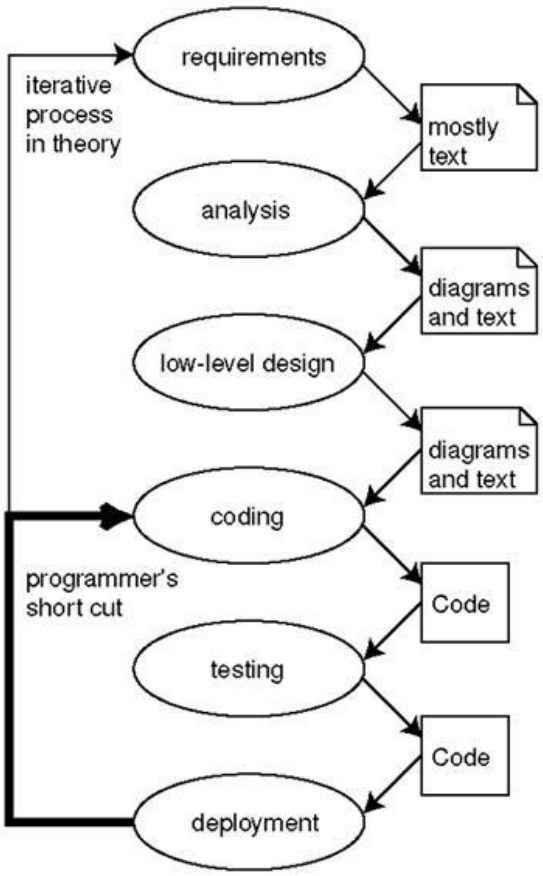
- O *Java Metadata Interface* (JMIs) são utilizados para gerenciar a criação, armazenamento, acesso, descoberta, e troca de modelos de ASM em termos de objetos Java.
- O formato XMI 1.2, fornecido em termos de um documento XML *Type Definition* (DTD), é gerado automaticamente a partir do AsmM, para a interação de modelos ASM para as ferramentas de serialização de XML.
- A AsmetaL (*Asmeta Language*) é uma notação textual AsmM, gerada a partir da gramática AsmM, como uma sintaxe para ser utilizada em modelos ASM de forma textual.
- A biblioteca padrão (*Standard Library*) é uma coleção declarativa de domínios ASM predefinidos (domínios de base para valores de dados primitivos como Boolean, Natural, inteiro, real, etc, e os domínios estruturados sobre outros domínios, como conjuntos finitos, sequências, entre outros) e funções que implementam um conjunto de operações em domínios.
- O AsmetaS é um simulador que torna os modelos AsmM em modelos executáveis.
- O gerador de testes *Tests Generation Tool* (ATGT) é uma ferramenta para a geração de casos de teste dos modelos, realizados em AsmM.
- Um plug-in para a ferramenta Eclipse, chamada AsmEE (*ASM Eclipse Environment*), atua no IDE para editar, manipular e exportar modelos de ASM usando todas as ferramentas/artefatos.

2.3 O Processo de Desenvolvimento MDA

Model Driven Arquitechture (MDA) é uma abordagem de desenvolvimento de sistema. É dirigida a modelos, uma vez que provê meios de usar modelos para direcionar o curso de entendimento, projeto, construção, distribuição, operação, manutenção e modificação. Esta técnica de desenvolvimento de software foi criada pela *Object Management Group* (OMG), onde diferentes modelos de objetos são ligados para construir um sistema completo. A OMG é um grupo formado por empresas de desenvolvimento de software do mundo todo, que tem como propósito definir padrões a serem adotados, tanto para MDA como para a linguagem de modelagem já estudada, a UML.

Essa técnica está associada a um conjunto de padrões de modelos, os quais são: o *Computacional Independent Model* (CIM) representa apenas requisitos do sistema e não mostra detalhes de sua estrutura. O CIM é usado como entrada para a elaboração da *Platform Independent Model* (PIM), definido com um alto grau de abstração, independente de qualquer tipo de tecnologia. Para finalizar, o PIM serve como entrada para a elaboração do *Platform Specific Model* (PSM). Cada PSM é gerado levando em conta detalhes específicos de uma determinada tecnologia a ser utilizada na implementação. Para cada PIM, vários PSMs podem ser gerados.

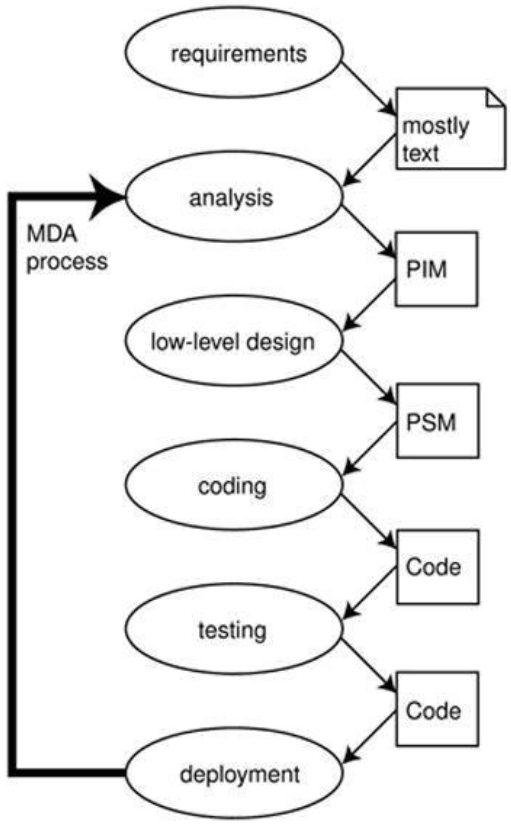
De acordo com Kleppe *et al.*, (2003), os documentos e os diagramas produzidos nas três primeiras etapas do ciclo de vida do desenvolvimento tradicional perdem o valor a partir do momento que a codificação tem início. Quando ocorrem mudanças ao longo do desenvolvimento, amplia-se cada vez mais a distância entre esses diagramas, pois estas mudanças são feitas apenas em código, onde, muitas vezes, não existe tempo disponível para atualização dos documentos de alto nível (Figura 2.6).



Fonte: (KLEPPE et al, 2003)

Figura 2.6 Ciclo de vida do desenvolvimento tradicional.

Entretanto, o ciclo de vida do desenvolvimento MDA difere bastante do modelo tradicional, pois o processo é voltado para a constituição dos modelos e definição de regras de transformações para geração automática de novos modelos e código-fonte, solucionando o problema descrito (KLEPPE et al., 2003), conforme ilustra a Figura 2.7.



Fonte: (KLEPPE et al., 2003)

Figura 2.7 Ciclo de vida do desenvolvimento MDA.

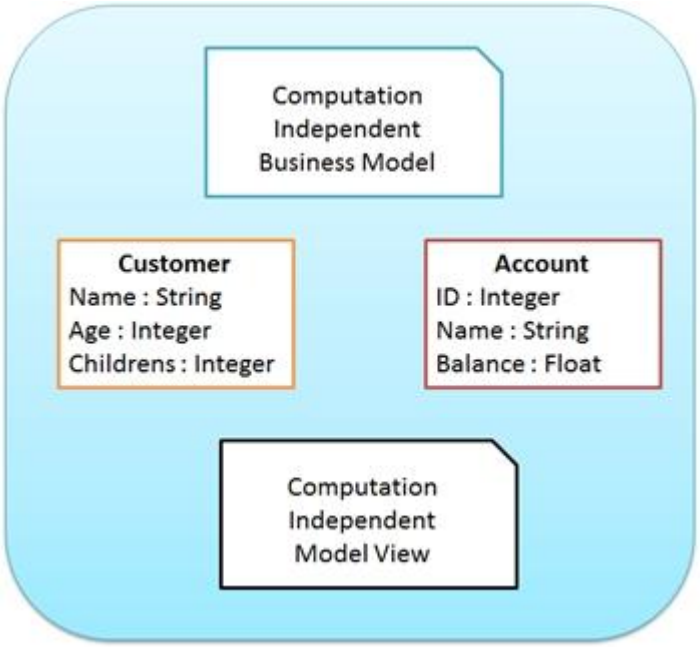
2.3.1 Modelo Independente de Computação

O CIM representa o modelo do sistema, desenvolvido pelo especialista de domínio, que muitas vezes não é um especialista em computação. Este modelo facilita a comunicação entre especialistas de domínio e os arquitetos de sistema, e pode incluir, descrições de casos de uso (por exemplo, como os usuários esperam para interagir com o sistema) e diagramas de sequência (por exemplo, as sequências em que certas ações são realizadas).

A obtenção do CIM é através do processo de documentação e especificação dos requisitos, ou seja, ao se especificar um modelo de requisitos para o sistema.

2.3.2 Modelo Independente de Plataforma

O PIM foca na operação do sistema, ou seja, no modelo computacional, contudo oculta os detalhes necessários para implantar esse modelo, em uma plataforma específica. Quando esse modelo é definido para um sistema, ele se torna único e não pode ser alterado, sendo que a sua alteração será realizada apenas se outra plataforma for especificada (Figura 2.8). O PIM pode ser especificado através de uma linguagem de modelagem como a *Unified Modeling Language* (UML) ou o *Systems Process Engineering Metamodel Specification* (SPEM).



Fonte: Adaptado de (BROWN, 2004)

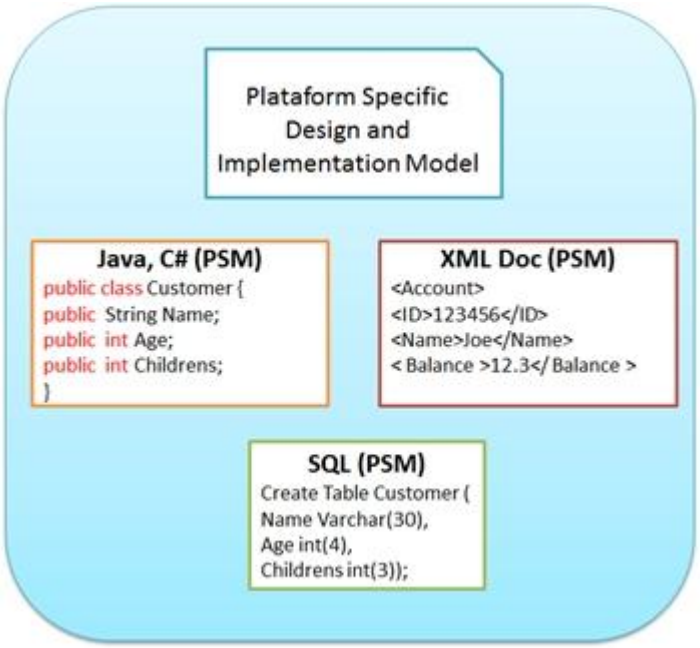
Figura 2.8 Modelo Independente de Plataforma.

A diferença básica entre CMI e PIM é que este já apresenta uma visão computacional do sistema (sem ligação a nenhuma plataforma), ausente no primeiro.

2.3.3 Modelo Específico de Plataforma

Esse modelo é feito especialmente para uma plataforma, agregando características e elementos representados de uma determinada plataforma, contendo informações da tecnologia utilizada na aplicação como a linguagem de programação, os componentes de middleware, a arquitetura de hardware e de software, conforme pode ser visualizado na Figura 2.9 (BUARQUE, 2009).

Outro exemplo de PSM seria a definição de classes para o controle de acesso ao sistema, com níveis de acessos atribuídos a usuários.



Fonte: Adaptado de (BROWN, 2004)

Figura 2.9 Modelo Específico de Plataforma.

Um PIM pode gerar mais de um PSM, em caso de sistemas multiplataforma, o que é comum nos dias atuais. Isso traz grande redução de custo com recodificação e faz o desenvolvedor focar sua atenção no PIM, ou seja, nas funcionalidades e lógica de negócio do sistema.

2.3.4 Mapeamentos na MDA

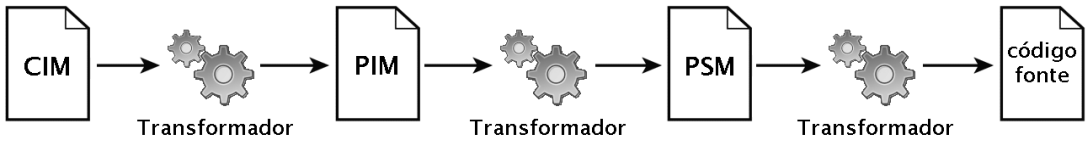
O conceito de mapeamento não é uma ideia nova e tem sido utilizada intensamente em vários campos, mais especificamente na área de bancos de dados. Entretanto, mapeamentos abrangem notações diferentes, como:

- Correspondências semânticas: correspondências explícitas entre os conceitos.
- Regras de transformação: conceito de esquema de transformação de outro.
- Mapeamento operacional: especificação de como calcular os elementos de um esquema com elementos de outro esquema.

Um mapeamento entre dois meta-modelos, raramente, especifica todos os conceitos em um meta-modelo. Em vez disso, os mapeamentos normalmente perdem alguma informação e podem se tornar parciais ou incompletos. Então, como calcular as relações entre os conceitos de meta-modelos associados. Automatizar sua descoberta é um dos desafios fundamentais não resolvidos para a interoperabilidade de dados. Na verdade, existem várias técnicas de mapeamento que podem ser classificadas como manuais ou semiautomáticas. O primeiro passo para mapear é definir uma linguagem apropriada.

2.3.5 Transformações na MDA

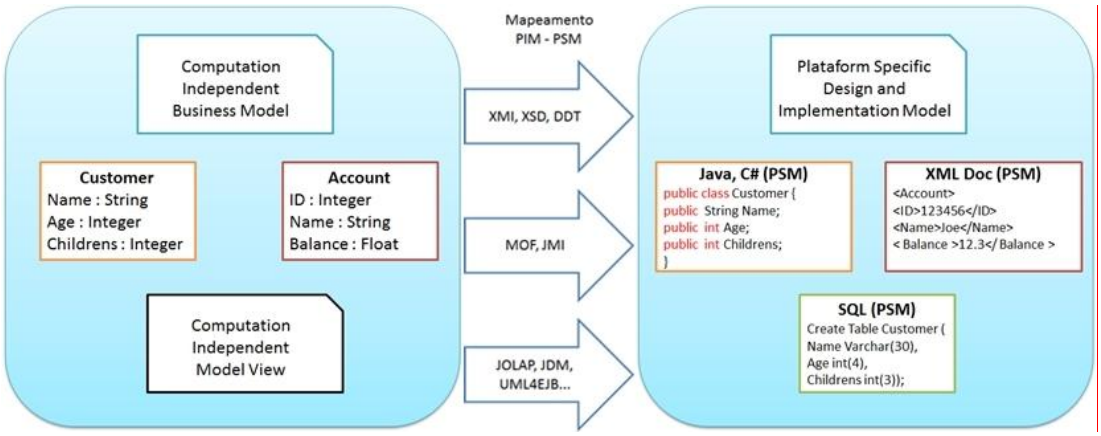
Uma transformação de modelos pode ser entendida como um conjunto de regras, condições e ações que se aplicam a um modelo origem com o objetivo de produzir um modelo destino. Em MDA, os modelos podem corresponder ao mesmo ou a distintos níveis de abstração MOF, e o sentido da transformação pode variar (MUKERJI e MILLER, 2008) como no exemplo da Figura 2.10.



Fonte: Adaptado de (KLEPPE *et al.*, 2003)

Figura 2.10 Etapas de transformação na MDA.

Mellor (2005) afirma que as transformações automáticas de modelos aumentam a portabilidade e a eficiência de desenvolvimento, que, por sua vez, aumentam a qualidade do software, enquanto reduz o tempo e o custo de desenvolvimento. A Figura 2.11 ilustra possíveis transformações e mapeamentos MDA.



Fonte: Adaptado de (BROWN, 2004)

Figura 2.11 Transformação de PIM para PSM.

2.4 Trabalhos relacionados

A realização de trabalhos e pesquisas envolvendo mapeamento e transformação de modelos não é algo recente. Harrisson *et al.* (2000) desenvolveram trabalhos para a IBM relacionados ao mapeamento de modelos UML e geração de código fonte Java e C++. Eles descrevem a importância de se possuir ferramentas para a geração de código, a partir de descrições de modelo para ajudar os desenvolvedores a manter a coerência entre um modelo e sua implementação. Então, foi desenvolvido um método novo para a geração de código Java, a partir da implementação de diagramas UML. Este método aceita diagramas UML especificados em um nível mais alto do que as ferramentas atuais, e colocam menos restrições aos modelos UML, como as generalizações e várias classes de associação, ao contrário das ferramentas atuais, que geram código de implementação desses sistemas em baixo nível de detalhe.

No trabalho de Cavarra *et al.* (2004), foi desenvolvido um framework de simulação de modelos UML, baseado em um esquema de mapeamento dos elementos do metamodelo UML para Máquinas de Estado Abstratas (ASM). Para realizar esse mapeamento, os elementos estruturais do modelo são traduzidos em um vocabulário ASM, como coleções de domínios e funções, enquanto a visão dinâmica é capturada por vários agentes escritos em ASM, que refletem o comportamento modelado em UML.

Entretanto, para desenvolver o presente trabalho, a contribuição mais significativa foi desenvolvida por Fraga Filho (2009), o qual realizou a

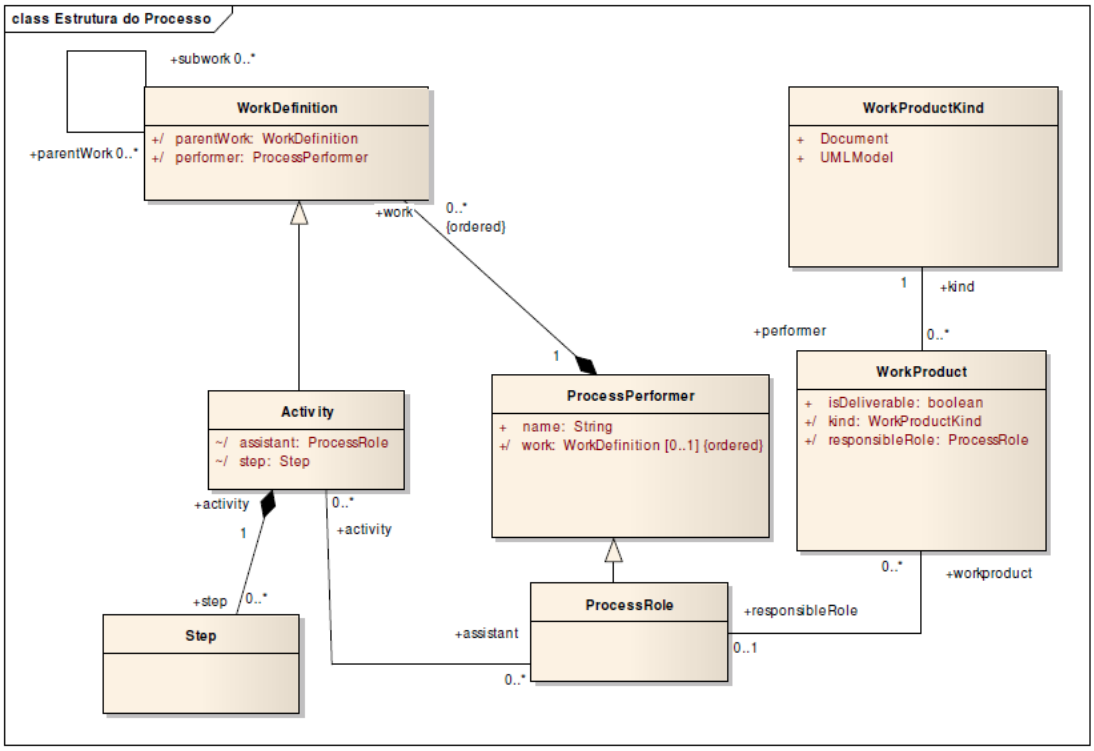
transformação de modelos de processo de software de um alto nível de abstração para um nível que permitisse sua execução. O modelo em questão foi nomeado para SPEMasm, adaptado do SPEM, sendo o mesmo mapeado para ASM.

O mapeamento utilizado por Fraga Filho (2009) foi o de instâncias, que especifica o relacionamento dos elementos entre os modelos estáticos e dinâmicos. A transformação do modelo proposta foi feita de forma manual e baseada na abordagem MDA da OMG, gerando um modelo executável de processo de software com ASM.

3 MAPEAMENTO E TRANSFORMAÇÃO DO MODELO

3.1 Modelo Proposto

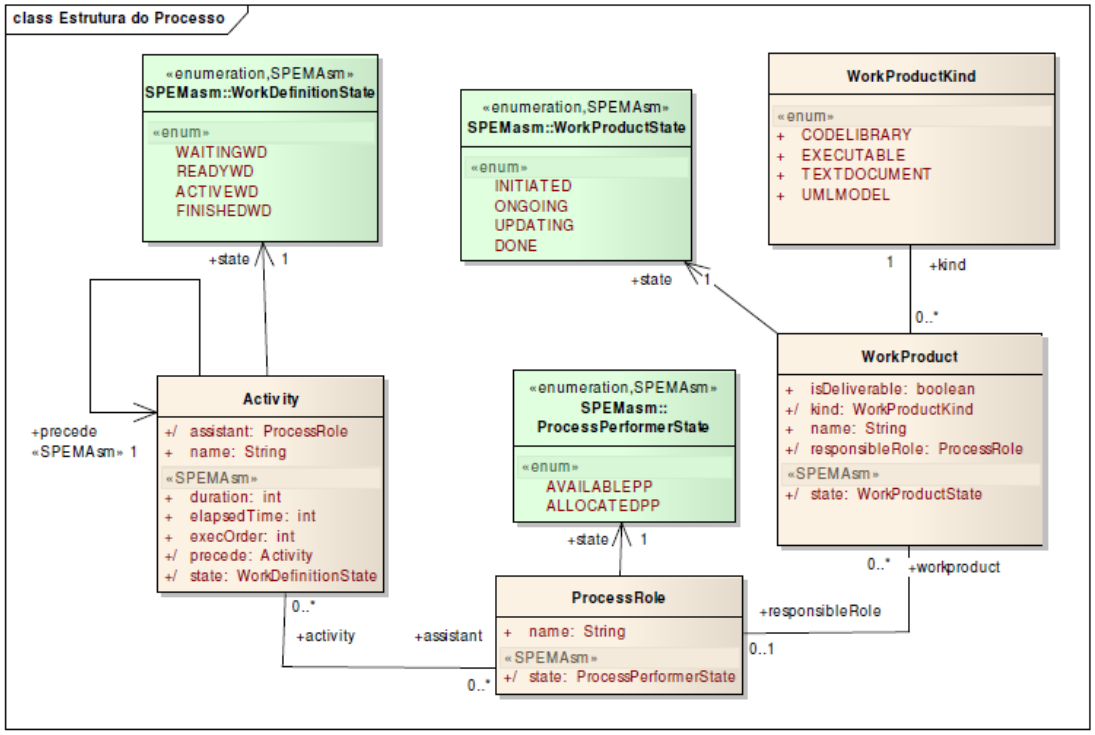
Fraga Filho (2009) propôs em seu trabalho a extensão do meta-modelo SPEM para simular processos utilizando Máquinas de Estado Abstratas. Esse simulador de processos foi criado a partir do pacote estrutural proposto no SPEM, versão 1.1, pois este pacote possui diversas estruturas que permitem a representação de qualquer processo de software (Figura 3.1).



Fonte: (OMG, 2005)

Figura 3.1 Modelo Estrutural de Processo.

Contudo, fez-se necessária uma adaptação ao modelo proposto no SPEM em sua versão 1.1, para que houvesse a possibilidade de criação de um simulador escrito em ASM. Essas adaptações estão ilustradas na Figura 3.2.



Fonte: (FRAGA FILHO, 2009)

Figura 3.2 Modelo Estrutural de Processo do SPEMasm.

A partir dessa adaptação, descrita na Figura 3.2, Fraga Filho (2009) desenvolveu um simulador em ASM que pudesse demonstrar comportamento para simular processos de desenvolvimento de software.

3.1.1 Objetivos

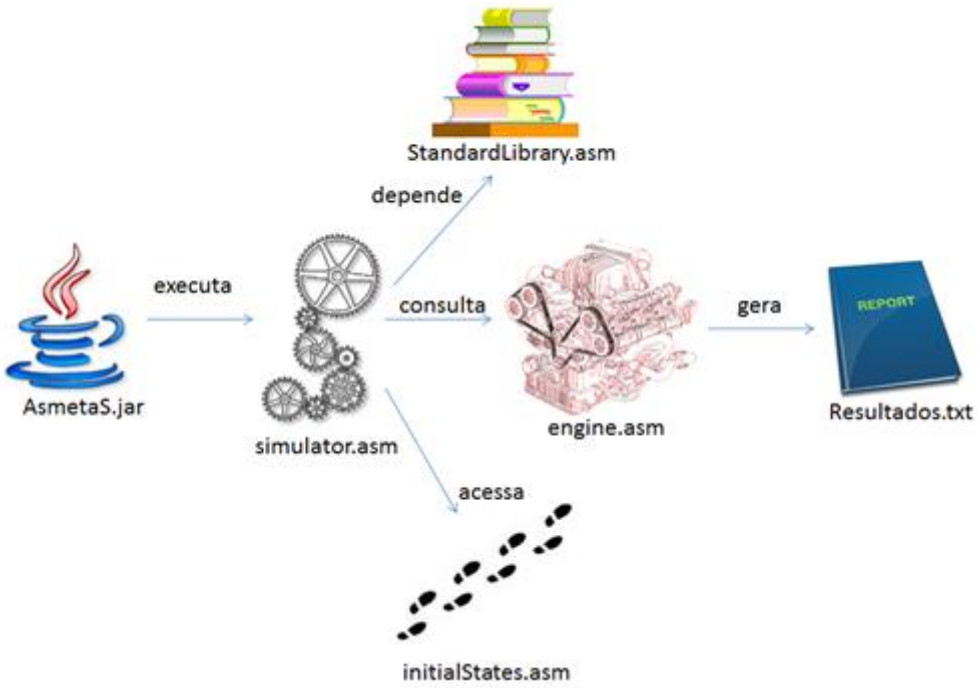
Máquinas de Estado Abstratas possuem um alto nível de abstração, contudo a falta de um ambiente mais amigável e flexível para desenvolver programas nessa linguagem deixa a sua programação mais complexa. A linguagem Java é uma das mais populares e robustas linguagens disponíveis no mercado. Realizar o mapeamento de ASM para a linguagem Java tornaria o simulador totalmente independente de plataforma, como também poderia fornecer uma interface de manipulação das variáveis envolvidas no processo de desenvolvimento de software.

A proposta desse trabalho é utilizar o modelo SPEMasm, desenvolvido por Fraga Filho (2009), que será o PIM. Será necessário também mapear o vocabulário escrito em ASM para a linguagem Java, realizando a transformação manual/automática das regras definidas em ASM gerando código Java (PSM). Para realizar a transformação de forma mais rápida, será desenvolvido um tradutor

AsmetaL para Java. Para finalizar, será implementado um ambiente que manipule as variáveis envolvidas em um processo de software (SCRUM, OpenUP, etc). O resultado da simulação, irá gerar um arquivo de texto, para que o mesmo possa ser interpretado pela ferramenta SPEMSIM, desenvolvida por Baia (2009).

A ferramenta SPEMSIM interpreta o resultado obtido de uma simulação criada em ASM, e exibe graficamente o comportamento gerado por uma simulação.

Fraga Filho (2009), em seu trabalho, define alguns elementos/arquivos que fazem parte do processo de execução do simulador escrito em ASM (AsmetaL). A Figura 3.3 descreve o relacionamento desses arquivos.



Fonte: Elaborada pelo autor

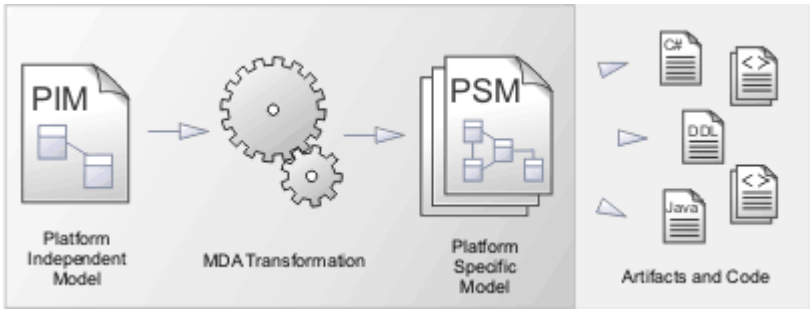
Figura 3.3 Processo de simulação de Fraga Filho (2009).

Para que uma aplicação, escrita em AsmetaL possa ser executada, é necessário um arquivo de extensão (.jar), de nome AsmetaS, que serve de simulador da linguagem. Em Fraga Filho (2009), foi desenvolvido um arquivo chamado de *simulator.asm*, que possui algumas regras, e o método principal (*main*) que invoca as funções dos arquivos *engine.asm* e *initialStates.asm*. O arquivo (*simulator*), assim como qualquer outro arquivo, que seja executado em AsmetaL, depende de um arquivo contendo bibliotecas específicas da linguagem, tal como o arquivo *StandardLibrary.asm*.

O arquivo (*simulator*), também, importa o arquivo (*initialStates*) que possui algumas regras e os estados e as variáveis que compõem o modelo do processo a ser simulado. É nesse arquivo que as modificações devem ser realizadas de forma manual.

Para finalizar, o arquivo (*engine*) contém as regras de negócios determinadas pelo simulador. Com as regras bem definidas, só há necessidade de se alterar as variáveis de entrada do processo, para gerar novas simulações.

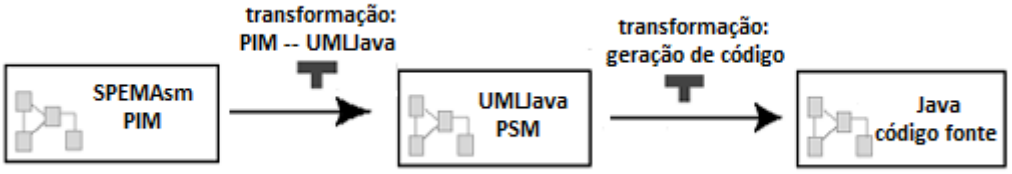
De acordo com a abordagem MDA, é necessário ter um PIM, o qual será transformado para PSM, e irá gerar um código a partir dele, de acordo com a Figura 3.4.



Fonte: (KLEPPE *et al.*, 2003)

Figura 3.4 Transformação MDA.

A primeira etapa da modelagem do processo de software realizada neste trabalho é a transformação das classes, que foram modeladas no PIM do SPEMasm. As definições das classes especificam todos os seus atributos e métodos com seus respectivos tipos, que durante a transformação são mapeados para os tipos correspondentes no PSM da linguagem de programação Java como plataforma alvo da transformação. A Figura 3.5 representa a etapa da modelagem do processo de software deste trabalho.



Fonte: Elaborada pelo autor

Figura 3.5 Transformação SPEMAsm para UMLJava.

3.1.2 Transformação

A abordagem MDA não especifica como as transformações podem ser desenvolvidas, entretanto auxilia com informações gerais, quatro maneiras de transformações de modelo. Esses métodos são explicados a seguir (MUKERJI, MILLER, 2008):

- **Transformação manual** – Trata-se da engenharia convencional. A abordagem MDA agrega valor de duas maneiras: a distinção explícita entre um modelo independente de plataforma e transformar o modelo de plataforma específica.
- **Transformação de PIM preparado com perfil** – Um PIM pode ser preparado utilizando uma plataforma, através de um perfil UML independente. Esse modelo pode ser transformado em um PSM usando uma segunda plataforma de perfil UML específica. A transformação pode envolver a marcação do PIM usando marcas fornecidas através de um perfil específico da plataforma.
- **Transformação utilizando padrões e marcações** – Padrões podem ser utilizados na especificação de um mapeamento. O mapeamento inclui um padrão e marcas correspondentes a alguns dos elementos do padrão. Os elementos de um PIM são marcados e transformados de acordo com o padrão estabelecido no mapeamento, produzindo um PSM.
- **Transformação automática** – Há contextos em que o PIM pode fornecer todas as informações necessárias para a implementação, e não há necessidade de acrescentar marcas ou usar os dados adicionais, a fim de ser capaz de gerar código. Uma delas é a de desenvolvimento baseado em componentes maduros, onde um middleware fornece um conjunto completo de serviços, e onde as decisões necessárias sobre a arquitetura são feitas uma vez por certo número de projetos. Essas decisões são implementadas em ferramentas, processos de desenvolvimento, modelos, bibliotecas de programa, e geradores de código.

A transformação será realizada, tendo por base o modelo de Fraga Filho (2009), ilustrado pela Figura 3.2, pois ele define um modelo UML da estrutura do

processo SPEMAsm. Com isso, podemos utilizar o padrão definido pela OMG de transformação de PIM preparado com perfil. Para realizar a transformação entre os modelos, serão criadas algumas regras de transformação descritas abaixo, entre o modelo de Fraga Filho (2009) e o modelo proposto.

- Para cada classe do PIM, uma classe Java é gerada.
- Cada classe PIM que possua uma associação unária, binária, etc, será transformada em uma associação igual em Java, pois estas associações especificam que objetos de uma classe estão ligados a objetos de outras classes.
- Cada classe PIM que possua dependência entre classes indica que os objetos de uma classe usam serviços dos objetos de outra classe. Essas classes serão transformadas em classe Java.
- Cada generalização no PIM deverá ser transformada em generalização em Java, para que os relacionamentos entre um elemento mais geral e um mais específico possam herdar as propriedades e métodos do elemento mais geral.
- Cada agregação no PIM será transformada em um tipo de associação Java (é parte de, todo/parte), onde o objeto Java parte é um atributo do todo e onde o objeto parte somente é criado se o todo ao qual estão agregados seja criado.
- Uma composição no PIM se transformará em relacionamento entre um elemento Java (o todo) e outros elementos Java (as partes), em que as partes só podem pertencer ao todo, e são criadas e destruídas com ele.
- Cada atributo de uma classe PIM é transformado em um atributo Java da classe de dados mapeados.
- Cada operação PIM é transformada em uma operação Java do componente gerado, através de uma classe do PIM.

De acordo com Czarnecki e Helsen (2003), as abordagens de transformação do tipo modelo-modelo podem ser classificadas em seis diferentes categorias:

- **Manipulação direta:** funcionam como um framework orientado a objetos, possivelmente fornecendo uma infraestrutura mínima para organizar as transformações.
- **Relacionais:** são caracterizadas pelo uso de relações matemáticas. Os elementos de entrada e de saída das transformações são definidos, de forma declarativa, por relações e restrições.
- **Baseadas em transformações de grafos:** utilizam grafos para representar modelos UML e, com isso, é possível transformá-los utilizando regras de transformação de grafos.
- **Orientadas pela estrutura:** possuem duas fases distintas: a primeira fica encarregada de criar a estrutura hierárquica do modelo de destino; a segunda fase configura as propriedades e referências dos elementos no destino.
- **Híbridas:** basicamente são aquelas que combinam características de algumas das categorias anteriores.

Para cada uma das classes modeladas no PIM, o processo primeiramente verifica sua categoria, que pode ser modelo, visão ou controle, de acordo com o padrão arquitetural conhecido como *Model-View-Controller* (MVC). Dependendo da categoria de classe que está sendo transformada, são adicionados novos atributos e métodos necessários para a sua implementação dentro do sistema modelado em sua plataforma alvo.

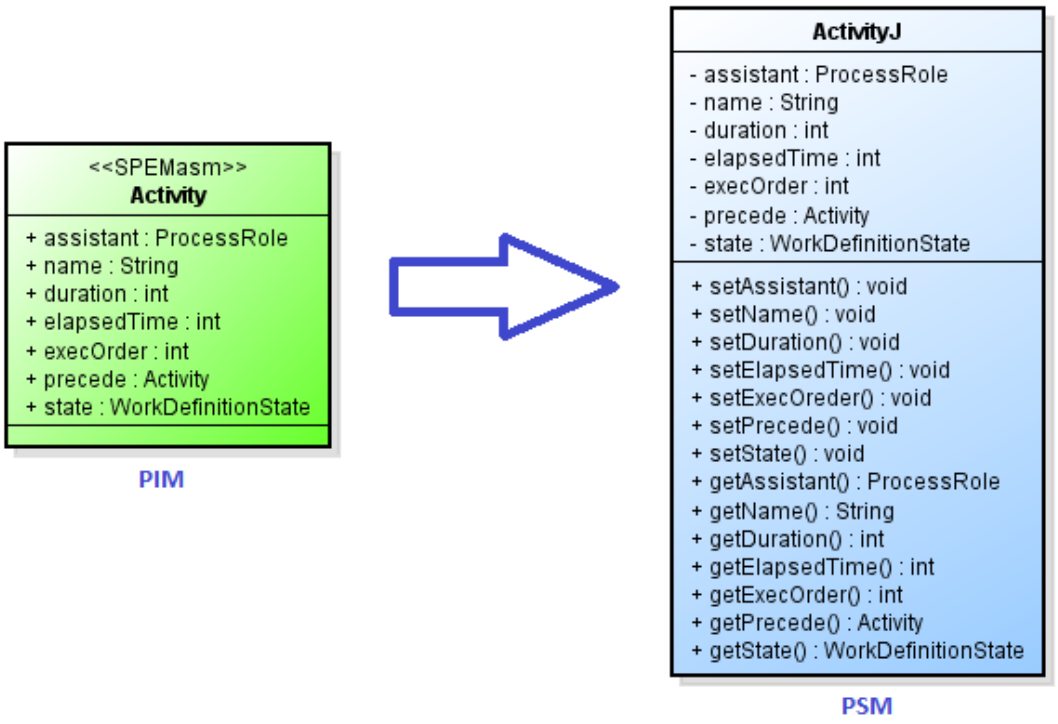
Tais atributos e métodos a serem acrescentados a cada um dos diferentes tipos de classe também são parametrizáveis de forma simples, através do arquivo de configuração. Dessa forma, caso o projetista tenha necessidade de acrescentar um novo método ou atributo comum a todas as classes de modelo, visão ou controle, ele deverá simplesmente adicionar o método atributo em questão no local correto do arquivo de configuração e a própria transformação se encarrega de acrescentá-lo a todas as classes da categoria determinada no PSM.

3.1.2.1 Classes de Modelo

No MVC, o modelo é o código que carrega alguma tarefa. Ele deve ser desenvolvido sem nenhuma preocupação em como ele será apresentado ao usuário.

O modelo possui um conjunto de métodos públicos que podem ser acessados para alcançar as funcionalidades.

Para cada uma das classes da categoria modelo no PIM, será gerada uma classe no modelo PSM, como mostra a Figura 3.6. Na primeira delas, são colocados todos os atributos da classe original, respeitando a regra de que todo o atributo que tenha sido modelado como público no PIM se torna privado no PSM e são gerados seus respectivos *getter* e *setter*, que são chamados de métodos que encapsulam o atributo da classe e oferecem interfaces de acesso e de alteração do mesmo pelas outras classes do sistema.

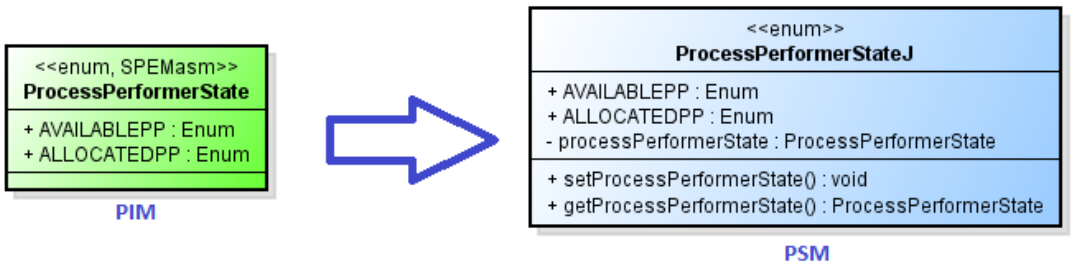


Fonte: Elaborado pelo autor.

Figura 3.6 Transformação de classe de modelo ActivityJ.

Para cada atributo do modelo PIM, um atributo e dois métodos assessores são criados no modelo PSM. A classe ***Activity*** foi transformada em ***ActivityJ*** para representar uma classe Java.

Todas as classes do modelo PIM que possuem apenas atributos (***ProcessRole*** e ***WorkProduct***) se comportam da mesma maneira da Figura 3.6, contudo as classes contendo atributos enumerados possuem outra estrutura, como mostra a Figura 3.7.



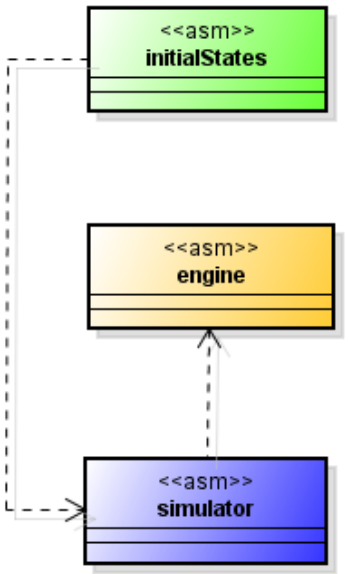
Fonte: Elaborado pelo autor.

Figura 3.7 Transformação de classes de modelo ProcessPerformerStateJ.

O modelo MVC não possui um padrão para as declarações de valores enumerados. Todavia, para definir um atributo enumerado, será necessária a criação de outro atributo, que possa manipular os dados de um ***Enum***. Então, foi criado um atributo em cada classe do modelo PSM, para manipular as informações contidas em um ***Enum***, através dos métodos assessores.

3.1.2.2 Classes de Visão

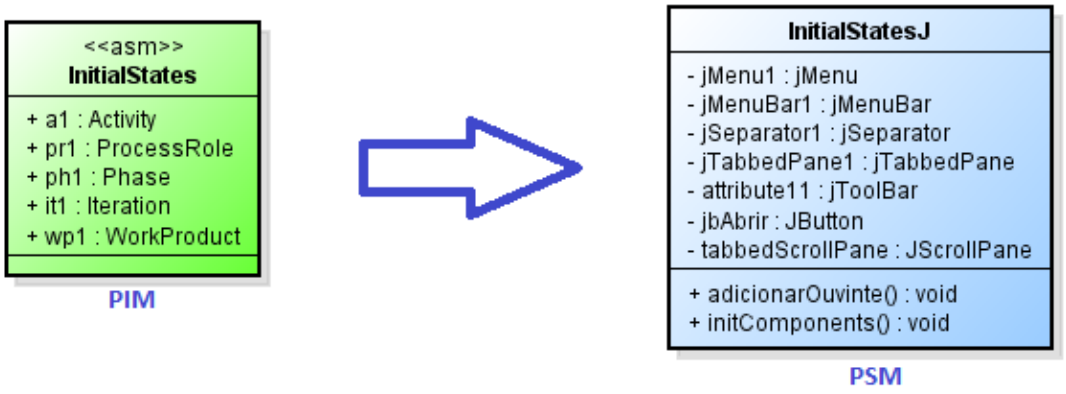
Para as classes de visão, serão utilizadas "classes" que compõe o simulador. Para isso, será modelada a estrutura final de seu simulador, pois é através desse outro modelo que poderemos definir as classes da visão e de controle.



Fonte: Elaborado pelo autor.

Figura 3.8 Estrutura do Simulador ASM.

No ASM, não há definição de modificadores de acesso. Sendo assim, os atributos modelados originalmente são transformados em privados. Além disso, no caso do PSM Java, cada classe de visão recebe um método (operador) público *adicionarOuvinteX*, onde X será um identificador qualquer ou o nome da classe e um método privado *initComponents*(), que irá executar os componentes da interface. A Figura 3.9 mostra um exemplo de como uma classe de visão do PIM é representada no PSM Java, após a transformação.



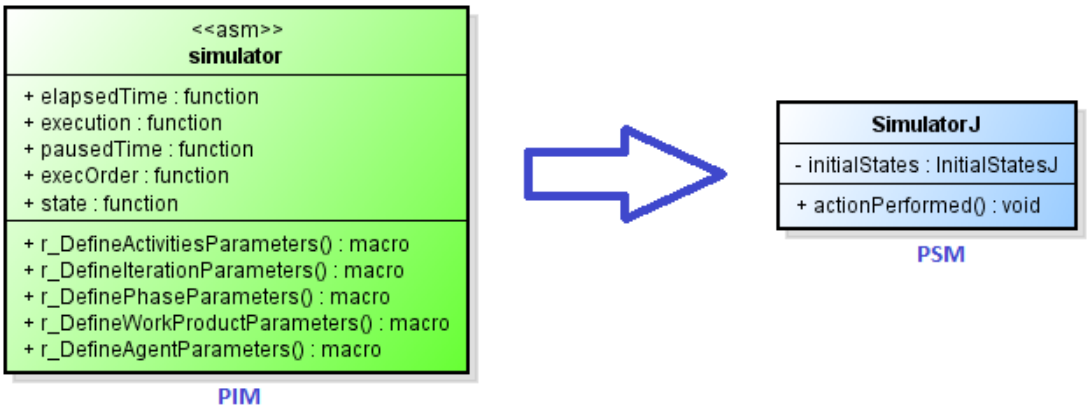
Fonte: Elaborado pelo autor.

Figura 3.9 Classe visão InitialStatesJ.

A camada de apresentação ou visão fornece componentes de interface gráfica, para que o usuário seja capaz de modificar o modelo. Recebe os resultados do modelo e especifica como estes serão projetados para o usuário. É responsabilidade das visões manter a consistência em sua apresentação quando o modelo muda, ou seja, quando o usuário define um novo processo de desenvolvimento de software, a interface permanece inalterada.

3.1.2.3 Classes de Controle

As classes de controle do PSM Java irão instanciar as suas respectivas classes visão, implementarão a interface *ActionListener* e terão métodos do tipo *actionPerformed*, responsáveis por receber os eventos provenientes de uma classe de visão. Um exemplo do que acontece durante as transformações de classes do tipo controladoras pode ser visualizado na Figura 3.10.



Fonte: Elaborado pelo autor.

Figura 3.10 Classe controlador SimulatorJ.

Os controladores traduzem interações com as visões nas ações a serem executadas pelo modelo. Baseado nas interações do usuário e no resultado das ações do modelo, o controlador responde selecionando uma visão apropriada.

As visões em MVC são associadas diretamente aos controladores, que são responsáveis por atualizar o modelo quando um usuário interage com uma visão. Então, o controlador invoca os métodos que modificam o modelo, mantendo o mesmo sempre atualizado.

3.1.3 Mapeamento

Toda transformação é definida através de mapeamentos. De acordo com a abordagem MDA, são definidas três categorias básicas de mapeamentos. São elas:

- **Mapeamentos de Tipos do Modelo** – um mapeamento entre PIM e PSM é descrito a partir dos tipos existentes no PIM e os tipos existentes no PSM.
- **Mapeamentos de Instâncias do Modelo** – na transformação de um PIM para um PSM, o mapeamento pode definir uma marcação que representa um conceito do modelo PSM, e que poderá ser aplicada aos elementos do modelo PIM, para informar como o elemento marcado deve ser transformado.
- **Mapeamentos Combinados entre Tipos e Instâncias** – esta categoria envolve mapeamentos que combinam as características dos dois tipos anteriores.

A Tabela 1 exibe o mapeamento de tipos, realizado neste trabalho com base no modelo SPEMasm. Então, temos uma coluna que representa os elementos envolvidos, sejam eles estruturais ou comportamentais, o modelo SPEMasm, o mapeamento para a linguagem ASM e o mapeamento para a linguagem Java.

Tabela 1 Extensão do mapeamento SPEM para ASM (adicionando Java)

Elemento	**SPEMAsm**		**ASM**	**Java**
Estrutura	*ProcessStructure*	*WorkDefitinitionState, WorkProductState, ProcessPerformState, WorkProductKind*	Domínio estático enumerado pré-definido.	**Lista enumerada de valores pré-definidos.**
		Activity, WorkProduct	Domínios abstratos	**Classes**
		ProcessRole	Agente	**Tipo**
		Atributos de Activity, WorkProduct e ProcessRole	Funções estáticas e dinâmicas	**Métodos**
		Associações	Funções estáticas	**Métodos**
		Instâncias	Funções estáticas	**Classes**
	ProcessLifeCycle	*Iteration, Phase*	Domínios abstratos	**Classes**
		Precondition e Goal	Funções dinâmicas	**Métodos**
		Associações	Funções estáticas	**Métodos**
Comportamento	Diagramas de Estados	*WorkDefitinitionState para Activity, Phase e Iteration,*	Regras de Transição	**Métodos**
		WorkProductState, ProcessPerformState,	Regras de Transição	**Métodos**
		Restrições	Axiomas (ou Predicados)	**Métodos**

Fonte: Adaptado de (FRAGA FILHO, 2009)

A Tabela 1 exibe uma extensão da proposta feita por Fraga Filho (2009), adicionando estruturas escritas na linguagem Java, para suportar a transformação manual das funções ASM. Contudo, a linguagem AsmetaL precisa ser mapeada, pois ela sofrerá a transformação de ASM para Java, proposta neste trabalho.

Tabela 2 Mapeamento AsmetaL para Java.

AsmetaL	Propósito	Java
asm <nome do modelo>	palavra-chave asm seguido pelo nome do modelo, que deve ser igual ao nome do arquivo (*case sensitive*).	**<modificador de acesso> class <nome da classe> <delimitadores>**
import <*extModule*>	import é uma palavra-chave e *extModule* é o caminho relativo do módulo que deseja importar.	**import java.<módulo>.<especificação>**
signature	Palavra chave que define as assinaturas de domínios e funções	**Declaração de variáveis globais da classe.**
Tipos de Domínios		
Real	Destinado aos tipos de ponto flutuante	**double.**
String	Domínios de texto.	**String**
Char	char é integral mas sem sinal.	**char**
Boolean	Domínio lógico.	**boolean**
Undef	Domínio não especificado.	**Object.**
Integer	Valores integrais inteiros.	**long**
Declaração de Funções		
dynamic controlled <nome da função> : Prod(Lift, Floor) -> <tipo>	As funções controladas dinamicamente podem assumir diversos valores ao longo do processo.	**Declaração de Métodos**
function <identificador> (<corpo>)	Função simples.	**Declaração de Métodos**
derived <identificador>: Prod(NumCard, Integer) -> <tipo>	Função derivada é uma função cujo valor de retorno é submetido a suas entradas.	**Declaração de Métodos**
Regras		
rule r_<identificador>	As regras são parecidas com as funções mas quando chamadas a ordem é importante.	**Declaração de Métodos.**
main rule r_Main	Função principal do sistema necessária para execução.	**public static void main (String args[])**
monitored function	Função necessária para	**Scanner(System.in)**

<identificador>: <tipo>	receber dados do usuário.	
Axiomas		
axiom <identificador> over id_function,...,id_rule : term	Axiomas são utilizados para expressar as restrições sobre as funções e regras.	**assert**

Fonte: Elaborada pelo autor.

A Tabela 2 define os mapeamentos mais importantes entre a linguagem AsmetaL e a linguagem Java. Com esses dados bem definidos, as regras escritas em AsmetaL podem ser transformadas para a linguagem Java. Essa tabela foi desenvolvida para expressar o mapeamento de alguns tipos da linguagem AsmetaL. Esse mapeamento foi necessário para criar o tradutor da linguagem ASM para Java.

Os elementos Atividades (*Activity*), Produtos de Trabalho (*WorkProduct*), Iteração (*Iteration*) e Fase (*Phase*) são representados como domínios abstratos por Fraga Filho (2009). Entretanto, domínios abstratos em ASM estão sendo mapeados como classes abstratas Java.

Tabela 3 Mapeamento dos Elementos Estáticos do ASM

ASM	JAVA
abstract domain ACTIVITY	public abstract class Activity { }
abstract domain WORKPRODUCT	public abstract class Workproduct { }
abstract domain ITERATION	public abstract class Iteration { }
abstract domain PHASE	public abstract class Phase { }

Fonte: Elaborada pelo autor.

Os domínios enumerados controlam os ciclos de vida de elementos especializados. Esses domínios foram transformados em classes Java, que expressam o seu comportamento através dos domínios enumerados.

Tabela 4 Mapeamento dos Domínios Enumerados

ASM	JAVA
enum domain WORKDEFINITIONSTATE = { WAITINGWD | READYWD | ACTIVEWD | FINISHEDWD }	public enum StateWorkDefinition { WAITINGWD, READYWD, ACTIVEWD, FINISHEDWD }
enum domain PROCESSPERFORMERSTATE = { AVAILABLEPP | ALLOCATEDPP}	public enum StateProcessPerformer { AVAILABLEPP, ALLOCATEDPP }

enum domain WORKPRODUCTSTATE = { INITIATED | ONGOING | UPDATING | DONE } enum domain WORKPRODUCTKIND = { TEXTDOCUMENT | UMLMODEL | EXECUTABLE | CODELIBRARY }	public enum StateWorkProduct { INITIATED, ONGOING, UPDATING, DONE } public enum KindWorkProduct { TEXTDOCUMENT, UMLMODEL, EXECUTABLE, CODELIBRARY }

Fonte: Elaborada pelo autor.

Cada elemento estrutural do SPEM possui atributos que foram etiquetados. Esses atributos são mapeados e transformados de acordo com a regra definida. A Tabela 5 exemplifica um dos elementos estruturais do SPEM e as suas etiquetas. Essa marcação foi definida por Fraga Filho (2009) e adaptada para suportar as especificações deste trabalho.

Tabela 5 Transformação do elemento estrutural *Activity*

Valores Etiquetados	Marcas do AsmM	Declaração com AsmetaL	Declaração com Java
name	StaticFunction	static name: Activity → String	**private String name;**
duration	StaticFunction	static duration: Activity → Integer	**private int duration;**
elapsedTime	ControlledFunction	dynamic controlled elapsedTime: Activity → Integer	**private int elapsedTime;**
execOrder	ControlledFunction	dynamic controlled execOrder: Activity → Integer	**private int execOrder;**
precede	ControlledFunction	dynamic controlled precede: Activity → Activity	**private Activity precede;**
state	ControlledFunction	dynamic controlled state: Activity → WorkDefinitionState	**private WorkDefinitionState state;**
assistant	StaticFunction	static assistant: Activity → Agent	**private Agent assistant;**
produce	StaticFunction	static produce: Activity → Seq(WorkProduct)	**private WorkProduct produce;**

Fonte: Adaptado de (FRAGA FILHO, 2009).

Assim como o elemento *Activity*, todos os outros elementos, como *ProcessRole*, *WorkProduct*, *WorkProductState*, *WorkDefinitionState*,

ProcessPerformerState, *Iteration* e *Phase*, possuem os seus valores etiquetados em forma de atributos de classe e mapeados da mesma maneira exibida na Tabela 5.

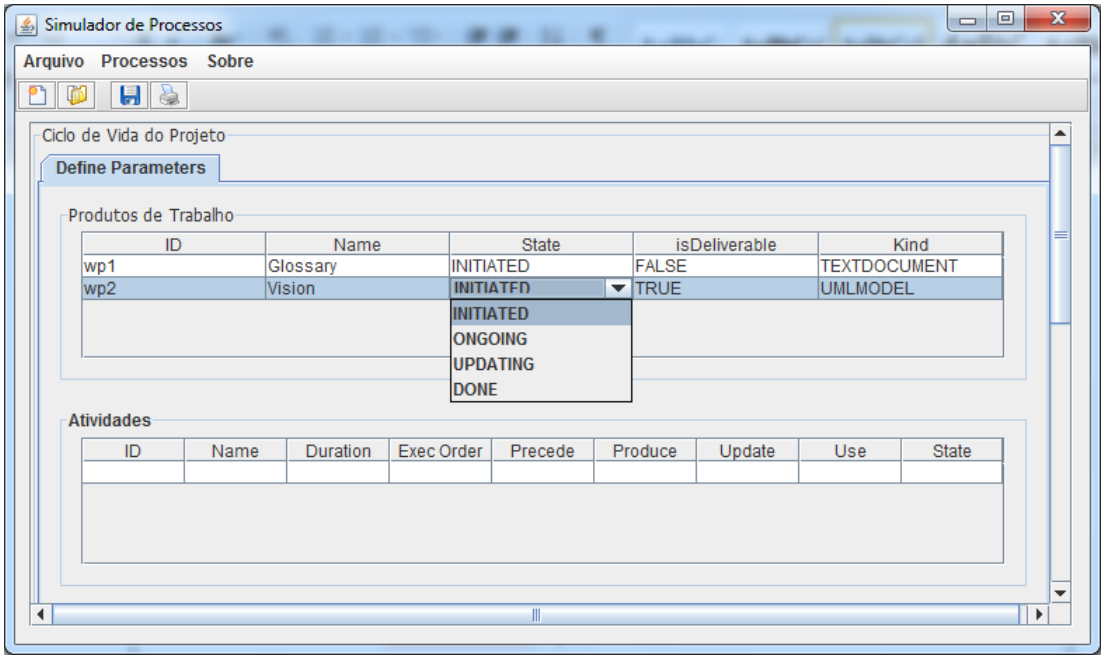
Após os mapeamentos estruturais do simulador, é importante analisar as regras definidas para simulação de um processo de desenvolvimento de software. Fraga Filho (2009) define cinco elementos como estados iniciais a serem processados pelo simulador, os quais são: *Activity*, *ProcessRole*, *Phase*, *Iteration* e *WorkProduct*.

3.1.4 Simulador

No simulador desenvolvido, em ASM, por Fraga Filho (2009), o grande problema era ter que especificar cada produto de trabalho diretamente em linha de código. Neste trabalho, a especificação dos dados é facilitada através de uma interface gráfica. O usuário só terá que acrescentar o que deseja em uma tabela, sendo que alguns desses dados podem ser pré-selecionados.

Essa regra mapeada para Java não necessita de um método próprio, pois ela ficará armazenada em um arquivo de texto contendo os dados salvos da tabela específica.

Na Figura 3.12, o usuário pode preencher os dados diretamente em uma tabela, o estado pode ser do tipo iniciado, em curso, atualizando e terminado. No final do preenchimento, o usuário poderá salvar e executar a aplicação para que a mesma armazene os dados das tabelas e os utilize nos métodos que irão gerar o arquivo de saída.



Fonte: Elaborado pelo autor.

Figura 3.11 Adicionando dados na tabela Produtos de trabalho.

A Figura 3.13 ilustra como os dados são armazenados em um arquivo de texto. Nessa Figura, os dados presentes são apenas os que foram inseridos, de acordo com a Figura 3.11, sendo que o restante permanece vazio.

```
1  ID|Name|State|isDeliverable|Kind|
2  wp1|Glossary|INITIATED|FALSE|TEXTDOCUMENT|
3  wp2|Vision|INITIATED|TRUE|UMLMODEL|
4  ID|Name|Duration|Exec Order|Precede|Produce|Update|Use|State|
5
6  ID|Name|State|Iterations|
7
8  ID|Name|State|Activities|
9
10 ID|Name|Perform|State|
11
```

Fonte: Elaborado pelo autor.

Figura 3.12 Dados armazenados em um arquivo de texto.

O simulador contém regras que atribuem valores para os parâmetros no arquivo ***initialStates***. Os parâmetros são instâncias e cada instância possui diversos valores atribuídos a elas, como: nome, estado, distribuído, tipo, dentre outros. Em Java, foi criada uma classe que possui tabelas (*JTables*), para que os dados pudessem

ser inseridos diretamente sem a necessidade de criar métodos que pudessem atribuir os valores, como mostra o Quadro 3.1.

```
macro rule r_DefineWorkProductParameters =
        forall $w in WorkProduct with isUndef(name($w)) = true  do
        par
            name(wp1) := "Glossary"
            name(wp2) := "Vision"
            state($w) := INITIATED

            isDeliverable(wp1) := false
            isDeliverable(wp2) := false
            kind(wp1) := TEXTDOCUMENT
            kind(wp2) := UMLMODEL
        endpar
```

Fonte: (FRAGA FILHO, 2009)

Quadro 3.1 Regra DefineWorkProductParameters.

Essa regra não precisa ser mapeada em Java, pois no simulador Java os dados são armazenados em um arquivo de texto, que são recuperados quando necessários. Entretanto, as regras do arquivo ***engine***, criado por Fraga Filho (2009), possui as regras de negócio do simulador, precisam ser mapeadas.

```
rule r_ActiveActivities =   forall $a in Activity with state($a) = READYWD do
    seq
        state(first(asSequence(assistant($a)))) := ALLOCATEDPP
        state($a) := ACTIVEWD
    endseq
```

Fonte: (FRAGA FILHO, 2009)

Quadro 3.2 Regra *ActiveActivities*.

A regra acima significa que todas as atividades que entram no estado READYWD passam para o estado ACTIVEWD, e o ator que executa a atividade fica alocado (ALLOCATEDPP), para realizar a atividade em questão.

A mesma regra mapeada em Java é demonstrada abaixo e segue os padrões descritos nas tabelas de mapeamentos (Quadro 3.3).

```
public void ActiveActivities() {
    if (jtActivities.getValueAt(0, 8) == READYWD) {
       jtProcessRole.setValueAt("ALLOCATEDPP",0, 3);
       jtActivities.setValueAt("ACTIVEWD",0, 8);
     }
  }
```

Fonte: Elaborado pelo autor.

Quadro 3.3 Regra mapeada para Java *ActiveActivities*.

Como está sendo tudo trabalhado em tabelas, todo objeto tabela possui um "jt + identificador", os métodos *getValueAt* e *setValueAt*, servem para obter um valor em uma determinada posição ou alterá-lo de acordo com a regra estabelecida. Com isso, os dados são alterados ou capturados em tempo de execução.

A Tabela 6 define como os dados são inseridos pelo usuário do sistema.

Tabela 6 Definição dos parâmetros das atividades.

DefineActivitiesParameters							
STATE:	***WAITINGWD***						
ID	***NAME***	***DURATION***	***EXECORDER***	***PRECEDE***	***PRODUCE***	***UPDATE***	***USE***
A1	*Define Vision*	25	0	*NULL*	WP1	*NULL*	*NULL*
A2	*Plan Project*	20	*NULL*	A1	WP2	WP3	WP1
A3	*Plan Iteration*	12	*NULL*	A2	WP1, WP2	*NULL*	WP1
A4	*Manage Iteration*	9	*NULL*	A3	WP3	WP3	WP3
A5	*Test*	5	*NULL*	A4	WP3	WP3	WP2
A6	*Integration*	10	*NULL*	A5	WP4	*NULL*	*NULL*

Fonte: Elaborada pelo autor.

Os dados da Tabela 6 são criados pelo autor, somente para exemplificar. Todavia, os nomes (*names*) são oriundos do processo unificado aberto *OpenUP* (IBM, 2010) um *framework* de processo *open source*, desenvolvido pela *Eclipse*

Foundation. Este *framework* fornece práticas de desenvolvimento de software. O *OpenUP* é baseado no processo unificado da *Rational* (RUP/*Unified Process*, 2010), que inclui desenvolvimento iterativo e incremental, entre outras características. A duração é definida em dias e a ordem de execução pode ser alterada pelo usuário, assim como a precedência das atividades, os processos de trabalho produzidos, atualizados e utilizados.

O estado da regra *DefineActivitiesParameters* pode ser escolhido através de um domínio enumerado *WorkDefinitionState*, sendo eles: { *WAITINGWD* | *CANCELEDWD* | *READYWD* | *ACTIVEWD* | *FINISHEDWD* | *PAUSEDWD*}.

Tabela 7 Definição dos parâmetros dos produtos de trabalho.

DefineWorkProductParameters			
STATE:	***INITIATED***		
ID	***NAME***	***ISDELIVERABLE***	***KIND***
WP1	*Glossary*	*FALSE*	*TEXTDOCUMENT*
WP2	*Vision*	*FALSE*	*TEXTDOCUMENT*
WP3	*Project Plan*	*FALSE*	*TEXTDOCUMENT*
WP4	*Iteration Plan*	*FALSE*	*TEXTDOCUMENT*
WP5	*Use-Case Model*	*TRUE*	*UMLMODEL*
WP6	*Architecture*	*FALSE*	*TEXTDOCUMENT*

Fonte: Elaborada pelo autor.

A regra que define os parâmetros de produtos de trabalho possuem uma identificação (WPn), um nome (baseado no ciclo de vida do *OpenUP/Basic*), se é passivel de entrega, e o tipo (*kind*) que pode ser um documento de texto, um diagrama UML, um executável ou uma biblioteca. Os estados para os produtos de trabalho são obtidos de um domínio enumerado *WorkProductState*, que podem ser: { *INITIATED* | *ONGOING* | *UPDATING* | *DONE* }.

A Tabela 8 define os parâmetros das iterações inseridas pelo usuário do sistema. Para essa regra, são definidos os valores de identificação, o nome e as atividades relacionadas com aquele identificador. O valor de estado é baseado no domínio enumerado *WorkDefinitionState*, que poderá assumir: { *WAITINGWD* | *CANCELEDWD* | *READYWD* | *ACTIVEWD* | *FINISHEDWD* | *PAUSEDWD*}.

Tabela 8 Definição dos parâmetros das iterações.

DefineIterationParameters		
STATE:	***ACTIVEWD***	
ID	***NAME***	***ACTIVITIES***
IT1	*Initiate Project*	A1, A3, A4
IT2	*Plan and Manage Iteration*	A2
IT3	*Requirements and Architecture*	A3, A5

Fonte: Elaborada pelo autor.

De forma análoga à descrição das regras de iteração, os parâmetros das fases envolvem as fases de um processo de desenvolvimento de software, como: Concepção, Elaboração, Construção e Transição (Tabela 9). Estes são os ciclos de vida de um projeto.

Tabela 9 Definição dos parâmetros das fases.

DefinePhaseParameters		
STATE:	***WAITINGWD***	
ID	***NAME***	***ITERATIONS***
PH1	*Inception*	IT1
PH2	*Elaboration*	IT2, IT3
PH3	*Construction*	IT3
PH4	*Transition*	IT3

Fonte: Elaborada pelo autor.

A última regra que sofre alteração direta é descrita por Fraga Filho (2009) como um agente que inicia o processo (*DefineAgentParameters*). Este agente inicia o processo, desde que existam atividades com estado em espera e que o agente esteja disponível (Tabela 10).

Tabela 10 Definição dos parâmetros do agente.

DefineAgentParameters		
STATE	***AVAILABLEPP***	
ID	***NAME***	***PERFORM***
PR1	*Analyst*	A1, A2, A3
PR2	*Project Manager*	A4
PR3	*Architect*	A5
PR4	*Developer*	A6

Fonte: Elaborada pelo autor.

3.1.5 Considerações Finais

Toda a formalização das regras de mapeamento e das suas transformações em cima do processo *OpenUP/Basic* (IBM, 2010) poderão ser utilizadas em diversos outros formalismos, como: eXtreme Programing (IBM, 2010), RUP (IBM, 2010), Scrum (SCHWABER, 2010), dentre outros. Basta especificar o ciclo de vida das suas fases e iterações.

No capítulo seguinte, é realizado um pequeno estudo de caso com a utilização das ferramentas criadas para validar o mapeamento realizado neste trabalho, assim como uma proposta de implementação de um ambiente amigável de simulação de processos.

4 ESTUDO DE CASO

O desenvolvimento deste trabalho gerou duas aplicações. Uma delas é um tradutor de regras AsmetaL para Java, a outra é um painel de controle, onde o usuário poderá manipular variáveis, com o intuito de descrever o comportamento de um ambiente de desenvolvimento de software que utilize um processo de desenvolvimento de software como framework. Para este estudo de caso o *OpenUP*, que é um framework de processo para desenvolvimento de software, será utilizado.

O painel de controle, quando executado, irá invocar métodos que realizam o processamento dos dados inseridos pelo usuário, via interface gráfica. Após a execução dos dados pelo simulador, um arquivo de texto será gerado. Este arquivo servirá de entrada para outro aplicativo, chamado de SPEMSIM, o qual recebe os dados gerados pela simulação e interpreta-os criando a simulação visual, utilizando a notação SPEM.

4.1 Modelo de Processo de Desenvolvimento de Software

O modelo de processos é o *OpenUP*, o qual fornece as melhores práticas baseadas nas opiniões de líderes em desenvolvimento de software e da comunidade de desenvolvimento de software, que cobre um conjunto diverso de perspectivas e necessidades de desenvolvimento.

O *OpenUP* é baseado em 4 princípios básicos que são: colaborar para alinhar interesses e compartilhar conhecimento, focar na articulação da arquitetura, balancear prioridades concorrentes com o retorno de valor para o *Stakeholder* e envolvimento continuo para obtenção de feedback e melhorias.

A Figura 4.1 descreve de forma simplificada o ciclo de vida do processo *OpenUP*.

Fonte: Elaborado pelo autor.

Figura 4.1 Ciclo de Vida do OpenUP.

A escolha do *OpenUP* é baseada no fato de que ele é um subconjunto do Processo Unificado da Rational (RUP), com as fases de Concepção, Elaboração, Construção e Transcrição entre outras coisas boas do RUP.

O mais interessante do *OpenUP* (IBM, 2010), é que ele uniu o planejamento e controle de marcos do RUP (IBM, 2010), com o gerenciamento de iteração e micro-ambiente do Scrum (SCHWABER, 2010). Então temos o gerenciamento de micro-ambiente com o *work*-item, que são as tarefas que o desenvolvedor utiliza, nesse ponto pode-se adicionar algumas das práticas do *Extreme Programming* (XP).

4.2 Modelo de Processo

Para que possamos executar a simulação do processo de desenvolvimento de software, devem ser preenchidos os campos das tabelas, de acordo com os parâmetros desejados, pelo usuário do sistema.

A Tabela 11 foi preenchida com informações definidas para executar um processo de simulação, esses dados foram escolhidos pelo autor. A aplicação em Java recebe todos esses dados como entrada, podendo armazená-los em um arquivo

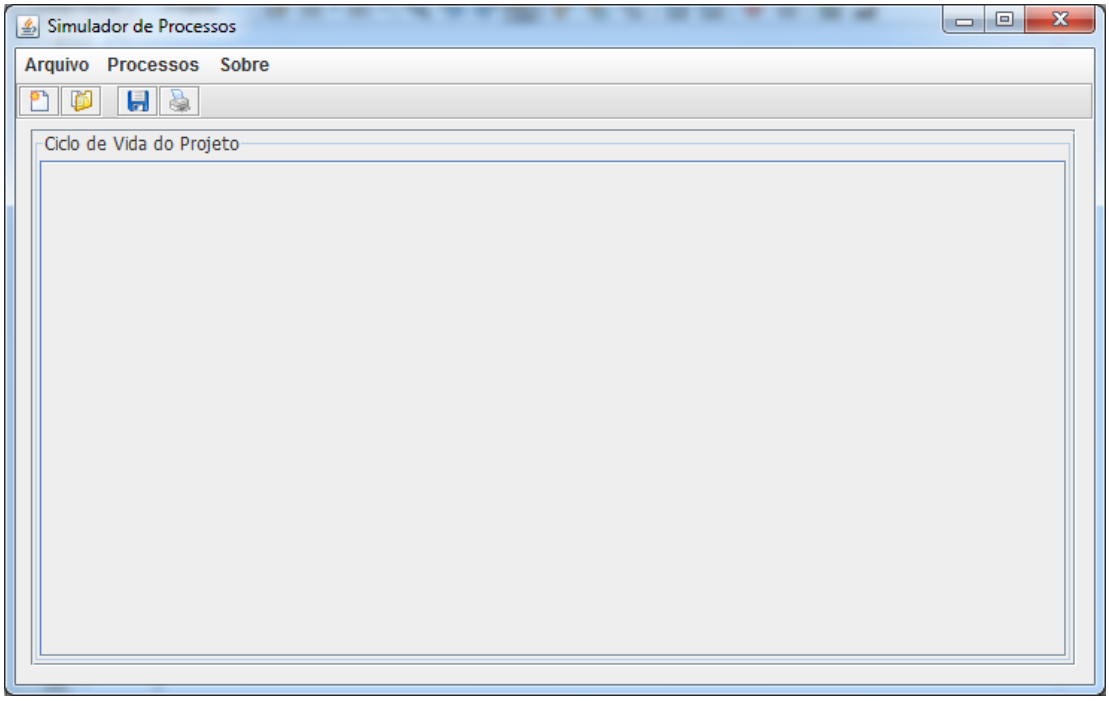
de texto para futuras utilizações, contudo o usuário do sistema poderá executar a simulação a partir dos dados preenchidos na tabela do sistema, como mostra a Figura 4.3.

Tabela 11 Estudo de Caso

TASKS	*ID*	*PRECEDE*	*STAKEHOLDERS*
Concepção	PH1		Arquiteto
Escopo do Sistema	A1		Analista
Documento de visão	WP1		Desenvolvedor
Plano de Iteração	WP2		Desenvolvedor
Requisitos do Sistema	A2		Analista
Lista de itens de trabalho	WP3	WP1	Desenvolvedor
Plano de projeto	WP4	WP1	Desenvolvedor
Protótipos	WP5	WP2	Desenvolvedor
Elaboração	PH2		Arquiteto
Baseline da arquitetura	A3		Analista
Protótipo	WP6		Desenvolvedor
Componentes do sistema	A4		Analista
Modelo de Dados	WP7		Desenvolvedor
Modelo de Implantação	WP8		Desenvolvedor
Construção	PH3		Arquiteto
Qualidade do Sistema	A5		Analista
Casos de Testes	WP9	WP8	Desenvolvedor
Versões Alfa e Beta	A6		Analista
Release do Sistema	WP10	WP9	Desenvolvedor
Transição	PH4		Arquiteto
Teste Beta	A7		Analista
Casos de Testes	WP11	WP10	Desenvolvedor
Distribuição	A8		Analista
Pacote de Distribuição	WP12	WP10	Desenvolvedor

Fonte: Adaptado de (Fraga Filho, 2009).

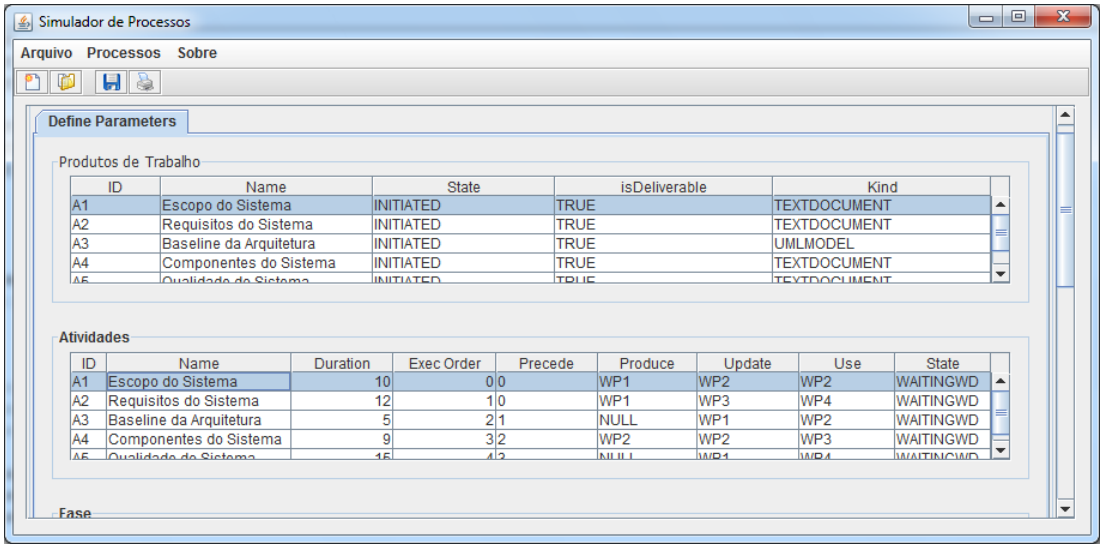
A Figura 4.2 descreve a interface principal do sistema. No menu “Arquivo”, o usuário possui as opções: novo, abrir, salvar e imprimir. No menu “Processos”, o usuário possui as opções: *OpenUP, Scrum* e *eXtremeProgramming*. O menu “Sobre” descreve os envolvidos no projeto.



Fonte: Elaborado pelo autor.

Figura 4.2 Interface Principal do Simulador.

A Figura 4.3 representa uma parte das tabelas a serem preenchidas pelo usuário. Antes, isso só era possível em linha de comando.



Fonte: Elaborado pelo autor.

Figura 4.3 Ciclo de Vida do Processo OpenUP.

Antes, cada informação era escrita em linha de comando e armazenada no próprio código fonte, contudo este procedimento pode conter falhas por parte do usuário, que poderá se esquecer de colocar alguma informação. Já através da

interface gráfica, fica visível o que deve ser preenchido, evitando erros desnecessários.

O Quadro 4.1 exibe a regra que define os parâmetros dos produtos de trabalho, sendo que essas informações são utilizadas no decorrer da simulação. Em Java, não havia necessidade de se atribuir valores às variáveis através de um método, pois os dados são acessados e alterados na interface gráfica, que podem ser armazenados em um arquivo de texto.

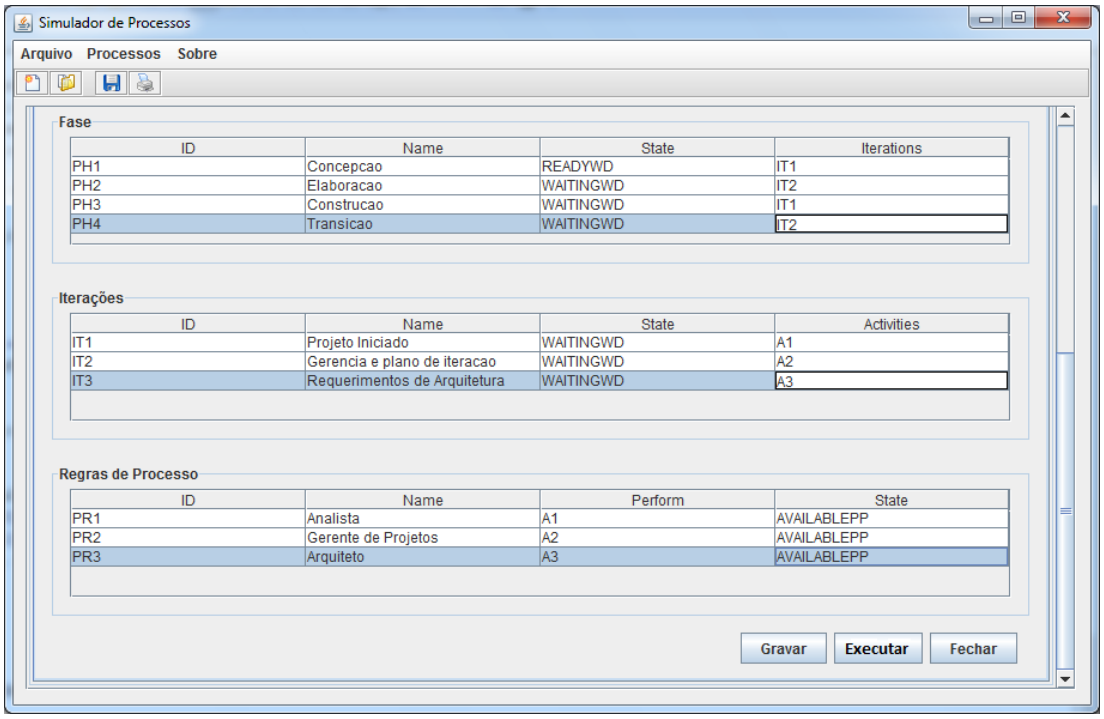
```
macro rule r_DefineWorkProductParameters =
        forall $w in WorkProduct with isUndef(name($w)) = true  do
        par
                name(wp1) := "Escopo do Sistema"
                name(wp2) := "Requisitos do Sistema"
                name(wp3) := "Baseline da Arquitetura"
                name(wp4) := "Componentes do Sistema"
                name(wp5) := "Qualidade do Sistema"
                state($w) := INITIATED
                isDeliverable(wp1) := true
                isDeliverable(wp2) := true
                isDeliverable(wp3) := true
                isDeliverable(wp4) := true
                isDeliverable(wp5) := true
                kind(wp1) := TEXTDOCUMENT
                kind(wp2) := TEXTDOCUMENT
                kind(wp3) := UMLMODEL
                kind(wp4) := TEXTDOCUMENT
                kind(wp5) := TEXTDOCUMENT
        endpar
```

Fonte: Adaptado de (FRAGA FILHO 2009).

Quadro 4.4 Definição dos parâmetros de produtos de trabalho.

A Figura 4.5 completa os dados escritos na Tabela 11, com os parâmetros de Fase, Iteração e o agente Regra de Processo. Cada uma dessas tabelas possui a sua

própria regra em ASM, o que foi modificado em Java. Essa classe faz parte da Visão do MVC.



Fonte: Elaborado pelo autor.

Figura 4.4 Ciclo de Vida do Processo OpenUP.

O Quadro 4.2 representa a regra que define ou atribui valores para as iterações. Através da interface gráfica, diversos tipos de usuário podem utilizar o sistema sem que o mesmo necessite saber algo de programação. Os botões “Gravar, Executar e Fechar” executam, respectivamente: gravar os dados em arquivo texto para futuras utilizações; executar o modelo descrito nas tabelas; e fechar a aba do modelo em questão.

```
macro rule r_DefineIterationParameters =
        forall $it in Iteration with isUndef(name($it)) = true  do
        seq
            state($it) := WAITINGWD
            name(it1):= "Projeto Iniciado"
            name(it2):= "Gerencia e plano de iteração"
            name(it3):= "Requerimentos de Arquitetura"
            activities(it1) := [a1]
            activities(it2) := [a2]
```

```
                    activities(it3) := [a3]
                endseq
```

Fonte: Adaptado de (FRAGA FILHO 2009).

Quadro 4.5 Definição dos parâmetros de iterações.

No simulador criado por Fraga Filho (2009), a descrição dos parâmetros era todo feito através de linhas de comando, em um bloco de notas. Com este painel de controle, o usuário do sistema poderá manipular os parâmetros via interface gráfica.

É através das tabelas que o usuário do sistema poderá determinar os parâmetros de entrada no sistema, esses parâmetros irão interagir com as regras de monitoramento e restrições responsáveis pela execução do simulador, após o processamento um arquivo de registro dessa simulação será gerado um arquivo de texto, onde o mesmo, serve de entrada para o aplicativo SPEMSIM. O Quadro 4.3 mostra um trecho do arquivo gerado.

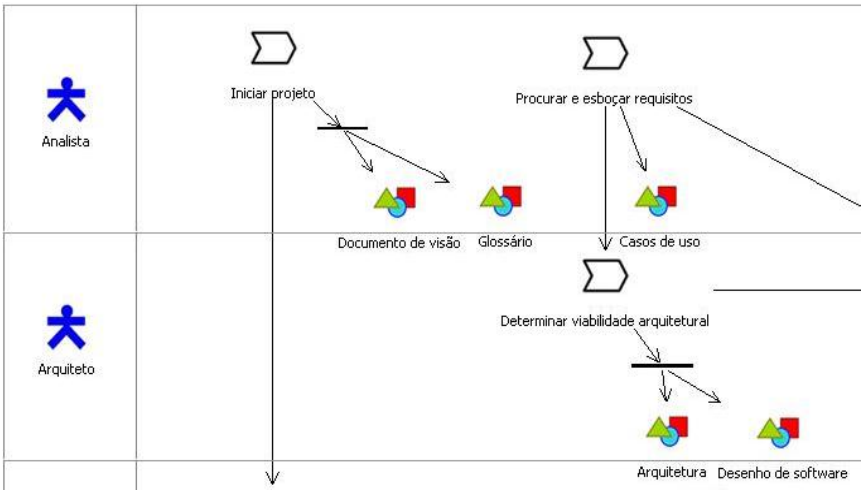
```
xxxxxxxxxxxxxxxx
<Transition>
<State>Activity={a1,a2,a3,a4,a5}
Agent={self}
Iteration={it1,it2,it3}
Phase={ph1,ph2, ph3, ph4}
ProcessRole={pr1,pr2,pr3}
WorkProduct={wp1,wp2,wp3,wp4,wp5}
activities(it1)=[a1,a2]
activities(it2)=[a3,a4]
activities(it3)=[a5,a6]
duration(a1)=25
duration(a2)=27
duration(a3)=13
duration(a4)=16
duration(a5)=25
duration(a6)=36
elapsedTime(a1)=1
elapsedTime(a2)=1
```

elapsedTime(a6)=1
execOrder(a1)=1
execOrder(a2)=1
execOrder(a3)=2
execOrder(a4)=2
execOrder(a5)=2
execOrder(a6)=1

Fonte: Elaborado pelo autor.

Quadro 4.6 Registro da Saída de uma Simulação.

Com um arquivo de saída em mãos, os dados da simulação poderão ser vistos graficamente através da ferramenta SPEMSIM, como mostra a Figura 4.6.



Fonte: (Ferramenta SPEMSIM).

Figura 4.5 Representação gráfica do modelo de processo OpenUp com o SPEM.

A ferramenta SPEMSIM interpreta o arquivo de saída e monta o gráfico da simulação. Ela também executa o modelo, e o usuário pode verificar como o processo de simulação acontece, passando por cada uma das fases.

4.3 Considerações Finais

Existem diversas ferramentas e abordagens que auxiliam no processo da MDA, elas criam modelos (PIM), transformam e até mesmo geram código. Contudo, elas não são fáceis de manipular. A aplicação desenvolvida neste trabalha, possui uma interface simples e limpa, permitindo ao usuário, fácil manipulação.

O próximo capítulo descreve as considerações finais deste trabalho e os trabalhos futuros que podem ser desenvolvidos.

5 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho foi realizado com objetivo de mapear um modelo independente de plataforma (PIM), transformar este modelo PIM para um modelo específico de plataforma PSM, realizar a transformação entre os modelos e gerar código Java.

Seguindo a ideia básica do processo de desenvolvimento de software de acordo com a MDA, este trabalho mostrou um processo de transformação que visa reduzir o nível de abstração dos modelos, caminhando em direção ao objetivo final, que é o código fonte executável do sistema de informação.

O mapeamento do modelo foi baseado nas regras da MDA, utilizando o mapeamento de tipos do modelo e no trabalho de Fraga Filho (2009). A transformação foi realizada manualmente, através das abordagens de transformação do tipo modelo-modelo com a manipulação direta, e utilizando a técnica de Transformação de PIM preparado com perfil.

Através do modelo proposto por Fraga Filho (2009), foi determinada uma definição semi-formal de modelos de processos nos níveis (PIM) e (PSM) da arquitetura MDA, sendo o PIM o modelo SPEMasm e o PSM o diagrama de classes UML, que utilizou o padrão MVC para estruturar o PSM.

Através do PSM gerado automaticamente pelo processo descrito neste trabalho, foi possível permitir ao projetista a visualização do resultado final da aplicação dentro da plataforma alvo selecionada por ele para a geração de código. Além disso, é possível que ele trabalhe em cima do PSM gerado, adicionando características da aplicação que tiram proveito de capacidades específicas da plataforma selecionada e que, por este motivo, não poderiam ser representadas e modeladas em um nível de abstração mais alto.

A linguagem Java se mostrou satisfatória, pois através dela, ganhou-se independência de plataforma, velocidade, conforto e desempenho. A transformação de PIM preparado com perfil alcançou os objetivos específicos propostos neste trabalho, pois um PIM pode ser preparado utilizando uma plataforma, através de um perfil UML independente. Esse modelo pode ser transformado em um PSM usando uma segunda plataforma de perfil UML específica.

O estudo de caso representado neste trabalho demonstrou o uso do conhecimento obtido para desenvolver essa pesquisa, ele mostrou o funcionamento

das regras de um modelo de processo de software específico o *OpenUP*, contudo, através das regras definidas pode-se expandir para outros modelos de processo de desenvolvimento de software.

O estudo de caso demonstra que o modelo de processo pode ser modificado e revisto constantemente, já que se configura a cada projeto, e com o auxílio do ambiente gráfico, torna-se mais fácil e confortável a sua manipulação.

O uso da linguagem Java como produto é outra grande contribuição deste trabalho. Java é apoiada por duas grandes empresas, a Oracle e a IBM. Ambas têm criado inúmeras e contínuas melhorias na linguagem e criado ferramentas que agilizam o trabalho dos desenvolvedores, que, consequentemente, conseguem construir mais com menos recursos. Através da sua portabilidade, ou seja, por não estar presa a um sistema operacional e ambiente específico, Java pode ser utilizado em diversos ambientes, desde pequenas empresas até grandes corporações, desde supercomputadores até celulares e computadores de mão. Com isso, empresas que adotam esta tecnologia têm maior portabilidade de seus aplicativos e podem utilizar uma única linguagem em diferentes computadores e dispositivos.

A linguagem Java fornece maior dinamismo, por ser um projeto, criado com o paradigma orientado a objetos. O dinamismo faz parte da natureza do Java, o que permite a extensibilidade durante a execução, alto desempenho, uma vez que suporta vários recursos de alto desempenho, como *multithreading*, compilação *just-in-time* e utilização de código nativo.

5.1 Trabalhos futuros

O modelo SPEMasm ainda está em fase de amadurecimento. Seria importante aprimorar esse modelo, para que o mesmo se torne menos específico e possa englobar novas funcionalidades. Além disso, durante o processo de transformação proposto, podem ser gerados novos diagramas, onde quase todas as classes são desenhadas umas sobre as outras. Um algoritmo de visualização dos objetos dentro dos diagramas, para facilitar a vida do projetista e tornar os diagramas gerados mais legíveis, foge ao escopo deste trabalho, mas também é visto como um trabalho futuro e algo que deve ser aprimorado.

O simulador feito em Java é uma ferramenta de apoio à tomada de decisão, mas ainda está em seu nível inicial, podendo ser adicionadas novas funcionalidades, bem como aprimorar as que ele já possui.

Pode ser desenvolvido um portal WEB colocando a ferramenta à disposição, para que outras organizações possam popular um banco de dados de informações de diversos tipos de modelos de desenvolvimento, gerando informação a ser interpretada por técnicas de *Data Mining* ou outra técnica.

APÊNDICE A

Este apêndice é responsável por exibir o mapeamento estático do ASM para a linguagem Java.

Atividades (Activity), Produtos de Trabalho (WorkProduct), Iteração (Iteration) e Fase (Phase), que são representados como domínios abstratos, foram mapeados em instâncias (classes) Java separadas, cada uma contendo atributos e os métodos *getters* e *setters* de seus atributos.

- abstract domain ACTIVITY
- abstract domain WORKPRODUCT
- abstract domain ITERATION
- abstract domain PHASE

Cada um desses elementos foi mapeado, respectivamente, para:

- public class ActivityJ
- public class WorkproductJ
- public class IterationJ
- public class PhaseJ

O domínio enumerado *WorkDefinitionState* é utilizado para controlar o ciclo de vida dos elementos especializados de definição de trabalho, como: Activity, Phase e Iteration. Esse domínio foi transformado em um Enum Java, em uma classe somente destinada a esse domínio.

A classe recebeu o nome *WorkDefinitionStateJ*, entretanto, o nome de seu enum teve que ser alterado.

```
public class WorkDefinitionStateJ {
   public enum WorkDefinitionState {
      WAITINGWD, CANCELEDWD, READYWD, ACTIVEWD, FINISHEDWD, PAUSEDWD
   }
}
```

Da mesma forma, aconteceu com os outros domínios enumerados em ASM, que foram mapeados em Java. Cada um ganhou uma classe, contendo um Enum.

A classe *ProcessPerformerStateJ* representa o ciclo de vida do ator durante a execução (computação) da máquina de estado abstrata, sendo que ele pode ficar disponível, alocado ou indisponível, dependendo da situação.

```
public class ProcessPerformerStateJ {
   public enum ProcessPerformerState {
      AVAILABLEPP, ALLOCATEDPP, UNAVAILABLEPP
   }
}
```

Os produtos de trabalho possuem um ciclo de vida definido, que são resultados das atividades desempenhadas pelos atores. Essas atividades são domínios enumerados.

```
public class WorkProductStateJ {
   public enum WorkProductState {
      INITIATED, ONGOING, UPDATING, DONE
   }
}
```

Os produtos de trabalho desenvolvidos durante um projeto de software têm tipos variados, e ainda podem ser complementados. Esses produtos estabelecem um ajuste com o modelo de processo de software. Sendo assim, eles serão representados por domínios enumerados.

```
public class WorkProductKindJ {
   public enum WorkProductKind {
      TEXTDOCUMENT, UMLMODEL, EXECUTABLE, CODELIBRARY
   }
}
```

REFERÊNCIAS BIBLIOGRÁFICAS

ABDEL-HAMID, T.; MADNICK, S. E. **Software Project Dynamics: an Integrated Approach,** Ed. Prentice Hall, 1991. 264 p.

ACUÑA, S. T.; FERRÉ, X. **Software Process Modelling**, Universidad Nacional de Santiago del Estero, Argentina 2005.

ARMENISE, P. et al. **Software Processes Representation Languages: Survey and Assessment.** In: International Conference on Software Engineering and Knowledge Engineering, 4.,1992, Capri, Italy. Proceedings... [S.l.]: IEEE Press, Junho de 1992.

ASMETAL. Disponível em: http://asmeta.sourceforge.net/index.html. Acessado em: 06 nov. 2010.

AZOFF, M. **The Benefits of Model Driven Development: MDD in Modern Web-based Systems**. Mar, 2008.

BAÍA, J. W. **SPEMSIM: um simulador visual de modelos de processos de desenvolvimento de software**, Ubá - Minas Gerais, 2009.

BECKER, J., ROSEMANN, M., VON UTHMANN, C. **Guidelines of Business Process Modeling**. Volume 1806 Computer Science., Springer Verlag (2000) 30-49

BERRY, D. M. **Formal Methods: the very idea - Some thoughts about why they work when they work**. Science of Computer Programming, 42(1):11{27, 2002.

BOERGER, E. STAERK, R. **Abstract State Machines: A Method for High-Level System Design and Analysis**. Springer Verlag, 2003.

BROWN, A. **An introduction to Model Driven Architecture**: MDA and today's systems, 17 Feb 2004. Disponível em:< http://www.ibm.com/developerworks/rational/library/3100.html>. Acessado em 21 de Novembro de 2010.

BUARQUE, A. da S. M. **Desenvolvimento de Software Dirigido por Modelos: Um Foco em Engenharia de Requisitos**. Recife, julho 2009.

CAVARRA, A.; RICCOBENE, E.; SCANDURRA, P. **Mapping UML into Abstract State Machines: A Framework to Simulate UML Models**, J. Studia Informatica Universalis, 3(3):367--398, 2004.

CURTIS, B. et al. **Process Modelling**. Communications of the ACM, New York, Vol. 35, Num. 9, 1992, pp. 75-90.

CZARNECKI, K., HELSEN, S. **Classification of Model Transformation Approaches**. In: Online Proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA, Anaheim, California, October 2003.

FIORINI, S.T. **Arquitetura para Reutilização de Processos de Software**, tese (doutorado em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, orientador Júlio Cesar Sampaio do Prado Leite, 2001.

FRAGA FILHO, C. V. **Simulação de Modelos de Processo de Software Utilizando Máquinas de Estado Abstratas**, Dissertação (Mestrado) - Curso de Ciência da Computação, Universidade Federal de Viçosa, Minas Gerais, 2009.

FRAGA FILHO, C. V.; BRAGA, J. L.; OLIVEIRA, A. de P.; DI IORIO, V. O. **Transformação de Modelos aplicada a Processos de Desenvolvimento de Software: Um estudo de caso com Máquinas de Estado Abstratas**. VIII Simpósio Brasileiro de Qualidade de Software - SBQS, 2009.

GARGANTINI, A.; RICCOBENE, E.; SCANDURRA, P. **AsmEE: an Eclipse plug-in in a metamodel based framework for the Abstract State Machines**, in First International Conference on Eclipse Technologies ECLIPSE-IT 2007Cuzzolin Editore (2007).

GARGANTINI, A.; RICCOBENE, E.; SCANDURRA, P. **Model-driven language engineering: The ASMETA case study**. In Software Engineering Advances, 2008. ICSEA '08. The Third International Conference on Software Engineering Advances, pages 373{378, 2008

GIMENES IMS, TANAKA S, OLIVEIRA JPM. **An Object Oriented Framework for Task Scheduling**. Proceedings of TOOLS Europe 2000, 2000 Mont St. Michel, France (Tools 33 Technology of Object-oriented Languages and Systems), vol. 1. IEEE Computer Society Press: Los Alamitos, CA, 2000; 383–394.

HARRISON, W.; BARTON, C.; RAGHAVACHARI, M. **Mapping UML designs to Java**, Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, p.178-187, October 2000.

IBM (Org.). **OpenUP/Basic**. Disponível em: <http://epf.eclipse.org/wikis/openup/>. Acessado em 12 de Novembro de 2010.

IBM. **Rational Unified Process Support**. Disponível em: http://www.01.ibm.com/software/awdtools/rup/support/doc.html. Acessado em 08 de Novembro de 2010.

KELLNER, M. I., HANSEN, G. A. **Software Process Modeling**. Technical Report CMU/SEI-88-TR-009, ESD-TR-88-010, Maio de 1988.

KELLNER, M. I.; MADACHY, R. J.;RAFFO, D. M. **Software Process Simulation Modeling: Why? What? How?**, Journal of Systems and Software, Vol. 46, No. 2/3, 15, 1999.

BECK, Kent: **Extreme Programming Explained: Embrace Change – 2ª Edição**, Addison-Wesley, 2004.

KLEPPE, A. et al. **MDA Explained: The Model Driven Architecture Practice and Promise**. 1st.ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

LIMA, C. D. Q.; JÚNIOR, J. P. da S. **Relacionando Engenharia de Requisitos à Engenharia de Software Orientada a Modelos**. Recife-PE, 2009.

LIMA, Carla A. G. **Um gerenciador de Processos de Software para o Ambiente PROSOFT**. Porto Alegre: CPGCC da UFRGS, 1998. Dissertação de Mestrado.

LITOIU, M. **Reduce Complexity With Model-Driven Development**: Use the IBM Software Development Platform to develop end-to-end solutions, Technical Designer, IBM, Software Group, 01 Sep 2004. Disponível em:< https://www.ibm.com/developerworks/ibm/library/i-modev1/>. Acessado em 20 de Novembro de 2010.

MADACHY, Raymond J.; BOEHM, Barry W. **Software Process Dynamics**. Early draft version 4/99. Los Angeles, California: IEEE Computer Society Press, 1999. Disponível em:<http://sunset.usc.edu/people/ray/spd/>. Acessado em 18 de Novembro de 2010

MALTEMPE, F. G. **Estudo do MDA e SOA.** Londrina – Paraná 2006.

MANLEY, J. H. **CASE: Foundation for Software Factories**, COMPCON Proceedings, IEEE, Setembro 1984, pp. 84-914.

MARIA, A. **Introduction to Modeling and Simulation**, Department of Systems Science and Industrial Engineering, State University of New York at Binghamton: Proceedings of the 1997 Winter Simulation Conference.

MELLOR, S. et. al. **MDA Destilada: Princípios da Arquitetura Orientada por Modelos** – 1.ed. Ciência Moderna. Rio de Janeiro. 2005.

MENDES, R. C. **Modelagem e Avaliação do CMMI no SPEM para Definição de um Meta-Processo de Software.** UFPE, Recife, Brasil, 2005.

MUKERJI, J., MILLER, J. **MDA Guide Version 1.0.1**. 2008. Disponível em:< http://www.omg.org/cgi-bin/doc?omg/03-06-01>. Acessado em 20 de Novembro de 2010.

OBJECT MANAGEMENT GROUP. **Software Process Engineering Metamodel Specification (SPEM) – version 2.0**. 2005. Disponível em: < http://www.omg.org/spec/SPEM/2.0/PDF/>.

PARK, S.; CHOI, K.; YOON, K.; BAE, D. **Deriving Software Process Simulation Model from SPEM-based Software Process Model**. APSEC 2007: 382-389.

PRESSMAN, R.S. **Engenharia de Software**, Makron Books, S.P., 1995, pp. 13-19

RAFFO, D. M.; SPEHAR, G.; NAYAK, U. **Generalized Simulation Models: What, Why and How?**, ProSim '03, 2003.

ROCHA, A.R.C.; MALDONADO, J.C.; WEBER, K.C. **Qualidade de Software**, São Paulo: Prentice Hall, 2001.

SCHWABER, Ken - **Scrum Guides**. Scrum.org. 2009. Disponível em: < http://www.scrum.org/ >. Acessado em: 12 nov 2010.

SOMMERVILLE, Ian. **Engenharia de Software** - 8a edição - Versão em Português, Addison Wesley. 2007.

TAJIMA, D.; MATSUBARA, T. **Inside the Japanese Software Factory**, Computer, vol. 17, No. 3, Março 1984, pp. 34-43

VERNADAT, F. B. **Enterprise Modeling and Integration**. London: Chapman & Hall. 1996