Universidade Federal de Juiz de Fora
Programa de Pós-graduação em Modelagem Computacional

Igor de Oliveira Knop

INFRAESTRUTURA PARA SIMULAÇÃO DE PROCESSOS DE SOFTWARE BASEADA EM METAMODELOS DE DINÂMICA DE SISTEMAS

Juiz de Fora
2009

Igor de Oliveira Knop

INFRAESTRUTURA PARA SIMULAÇÃO DE PROCESSOS DE SOFTWARE BASEADA EM METAMODELOS DE DINÂMICA DE SISTEMAS

Dissertação apresentada ao Curso de Mestrado do Programa de Pós-graduação em Modelagem Computacional: área de concentração em Sistemas Computacionais Aplicados da Faculdade de Engenharia da Universidade Federal de Juiz de Fora, como requisito parcial para obtenção do Grau de Mestre em Modelagem Computacional.

Orientador: Prof. Dr. Ciro de Barros Barbosa

Co-Orientador: Prof. Dr. Paulo Roberto de Castro Villela

Juiz de Fora

2009

INFRAESTRUTURA PARA SIMULAÇÃO DE PROCESSOS DE SOFTWARE BASEADA EM METAMODELOS DE DINÂMICA DE SISTEMAS

Igor de Oliveira Knop

DISSERTAÇÃO SUBMETIDA AO PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM COMPUTACIONAL DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M.SC.) EM MODELAGEM COMPUTACIONAL.

Aprovada por:

Prof. Ciro de Barros Barbosa, D.Sc.
(Orientador)

Prof. Paulo Roberto de Castro Villela, D.Sc.
(Co-Orientador)

Prof. Guilherme Horta Travassos, D.Sc.

Prof. Márcio de Oliveira Barros, D.Sc.

Prof. Rodrigo Weber dos Santos, D.Sc.

JUIZ DE FORA, MG - BRASIL
AGOSTO DE 2009

Knop, Igor

Infraestrutura para simulação de processos de software baseada em metamodelos de dinâmica de sistemas. / Igor Knop.
– 2009.
119 f. : il.

Dissertação (Mestrado em Modelagem Computacional) – Universidade Federal de Juiz de Fora, Juiz de Fora, 2009.

1. Engenharia de Software. 2. Softwares. I. Título.

CDU 004.41

À minha avó Ana Narcisa (*in memoriam*).

AGRADECIMENTOS

Nesta parte do texto fico sempre preocupado com a possibilidade de esquecer alguém. Então, desde já agradeço a todos que acompanharam, torceram ou possibilitaram o início e conclusão deste trabalho.

Agradeço a Deus, por acreditar que Sua mão se manifesta nos mais corriqueiros fenômenos do dia dia e estas pequenas bênçãos resultam na cadeia de eventos que me permite estar aqui hoje escrevendo a pessoas que, por meio Dele, contribuíram para o desenvolvimento deste trabalho.

Ao Prof. Ciro Barbosa e Prof. Paulo Villela, pela orientação e paciência durante todo o desenvolvimento desta dissertação. Em especial ao Prof. Ciro Barbosa, pelo entusiasmo contagiante, atenção e disponibilidade desprendida (muitas vezes em horas não tão convencionais). Em especial ao Prof. Paulo Villela, pela sabedoria e capacidade de fazer as coisas acontecerem, inspirando o progresso e aperfeiçoamento de todos com quem trabalha.

À Direção do Mestrado em Modelagem Computacional, pelo suporte constante e pela confiança depositada em nosso trabalho. Em especial ao Prof. Luis Paulo Barra e Prof. Rodrigo Weber que, com maestria, mantiveram um ambiente coeso e humanizado no curso.

Ao Prof. Guilherme Horta Travassos e ao Prof. Márcio de Oliveira Barros por participarem da banca de avaliação.

À Natália de Mira Braga, pela presteza e dedicação com que resolve todos os problemas que levamos à Secretaria do MMC.

À Universidade Federal de Juiz de Fora, pela infraestrutura e apoio financeiro.

Aos professores do MMC, por dividir conosco parte dos seus conhecimentos durante os créditos. Em especial à Prof. Regina Braga, pelos conselhos valiosos usados em boa parte deste trabalho.

Aos colegas do Mestrado, que durante os créditos dividiram as tensões, experiências e conhecimentos, sem nunca perder o ânimo nem o bom humor.

Aos meus amigos de longa data da Uikibiu que, mesmo estando longe em suas carreiras

profissionais, sempre arrumam um tempo para nos reencontrarmos para, por algumas poucas horas, retirarmos o peso do mundo de nossas costas.

À todos os membros do Projeto Ciência Viva, que dividiram comigo seus talentos e sonhos nas salas do Agrosoft.

Aos meus avós, pela fonte inesgotável de carinho e confiança.

Aos meus pais, pelas incontáveis horas de dedicação, suporte e amizade.

À Luzia Célia e família, pelo apoio e carinho dado em momentos cruciais.

E, finalmente à Maritza, por ter sido meu porto seguro repleto de amizade, amor, companhia e paciência. Sem o qual o caminho até aqui seria muito mais difícil.

RESUMO

Os resultados de projetos envolvendo desenvolvimento de software são melhores quando o gerente responsável possui uma certa experiência adquirida em projetos anteriores. Porém, é inviável para as instituições de ensino educar seus alunos criando projetos pilotos devido a problemas com a escala de tempo, custos e pessoal necessários. Uma alternativa para estudos de problemas que não podem ser reproduzidos dentro de uma escala viável é a modelagem. Este trabalho desenvolve uma infraestrutura computacional, independente de domínio, que serve como base para construção de aplicações que utilizam técnicas de modelagem e simulação. Esta infraestrutura é utilizada para estudos das causas e efeitos das dinâmicas encontradas em processos de desenvolvimento de software como nosso domínio de aplicação. O principal componente desta infraestrutura é a biblioteca JynaCore API, que implementa duas linguagens baseadas em Dinâmica de Sistemas para descrição dos modelos: os diagramas de estoque e fluxo e os metamodelos de Dinâmica de Sistemas. Como prova de conceito, um protótipo de simulador de uso geral é construído para realizar simulações com um conjunto de modelos encontrados na literatura sobre processos de software. Adicionalmente, apresentamos uma revisão das alternativas que permitem a modelagem de processos desenvolvimento de software em um computador e as bases teóricas para as duas linguagens de modelagem suportadas pela infraestrutura. A abordagem permite a construção de simuladores, modelos e cenários (variações de um modelo mais geral) onde os usuários das ferramentas podem experimentar diversas situações práticas em ambientes simulados.

Palavras-chave: Modelagem Computacional. Dinâmica de Sistemas. Modelos de Processos de Software.

ABSTRACT

Better results are achieved, in projects involving software development, when the responsible manager has some previous experience in projects. However, it is impossible for educational institutions to educate their students by creating pilot projects for each student, due to problems with the time scale, costs and staff. An alternative way study problems that are difficult to handle in a real scale is doing modeling. This work presents a computational infrastructure we have built, which is general purpose regarding application domain, and developed as a basis to build applications that use modeling and simulation techniques.This infrastructure is used to study the causes and effects of the dynamics found in software development processes, taken as our field of application. The main component of this infrastructure is the library JynaCore API, which implements two languages based on System Dynamics for describing the models: stock and flow diagrams and the System Dynamics Metamodels. As proof of concept, a general purpose prototype simulator is built to perform simulations with a set of models on the software processes literature. Additionally, we present a review of alternatives that allow modeling software development processes on a computer and the theoretical bases for the two modeling languages supported by the infrastructure. The approach allows the construction of simulators, models and scenarios (variations of more general models) where the users of the tool can experiment various practical situations in simulated environments.

Keywords: Computational Modeling. System Dynamics. Software Processes Models.

Sumário

Lista de Figuras

Lista de Listagens

1 Introdução

Segundo Madachy (2008), um processo de software descreve grupos de pessoas, artefatos e tarefas cuja finalidade é desenvolver software. É difícil negar a existência e os efeitos dinâmicos em processos de software, principalmente quando se leva em consideração os efeitos combinados de pressões no cronograma, excesso ou falta de comunicação dentro de uma equipe, mudanças nas condições do negócio, volatilidade de requisitos, métodos de trabalho, controle de qualidade, mudanças organizacionais, eventos desmotivantes e outros. Com todas estas complicações é muito difícil prever os efeitos que pequenas mudanças, em uma parte do processo, produto ou projeto acarretarão, pois estes são frutos da interrelação de diversos fatores.

A maioria dos projetos de desenvolvimento de software usa ferramentas e métodos assumindo que os requisitos vão se manter relativamente estáticos: estes são congelados em um determinado momento, desenvolve-se o software e o entregam tempos depois assumindo que os requisitos ainda são relevantes e o software útil (BOEHM, 2008). Não é incomum que muitos projetos de desenvolvimento de software acabam por consumir mais recursos do que planejado, demoram mais tempo para serem realizados, possuem menos funções e menor qualidade do que esperado (BARROS, 2001).

Uma das maiores vantagens do uso de modelos e sua subsequente simulação é prover bases para experimentação virtual: torna-se possível prever efeitos causados por mudanças nas suas configurações e responder a questões relacionadas a seu comportamento, sem ser necessário interagir com um processo real (KELLNER; R; D, 1999). A modelagem e simulação podem ser usadas como ferramentas educacionais, onde um conhecimento sobre um determinado modelo pode ser passado aos alunos ou aos profissionais menos experientes. Nas quais os alunos irão interagir com os modelos durante seu estudo, através de simuladores de uso geral ou simuladores que procuram reproduzir algumas condições do ambiente do sistema real (utilizando os chamados “simuladores de vôo”). Em ambos os tipos de simuladores, a idéia é a mesma: experimentar os efeitos das variações de parâmetros ou estrutura a fim de compreender o comportamento do

modelo.

A presente dissertação apresenta como contribuição principal a biblioteca *JynaCore API*, desenvolvida para servir de infraestrutura para construção de aplicações que utilizam técnicas de modelagem e simulação para estudos das dinâmicas de modelos de processos de desenvolvimento de software encontrados na literatura. A JynaCore API implementa duas linguagens baseadas em Dinâmica de Sistemas para descrição de modelos dinâmicos: os diagramas de estoque e fluxo (FORRESTER, 1961) e os Metamodelos de Dinâmica de Sistemas (BARROS, 2001). A biblioteca, desenvolvida utilizando a linguagem Java, é disponibilizada como um projeto de código livre para maior interoperabilidade com outras aplicações.

Como prova de conceito, este trabalho também apresenta um protótipo de simulador de modelos de uso geral: o *JynacoreSim*. Construído utilizando a JynaCore API, este simulador realiza, em uma mesma interface, simulações a partir de modelos descritos por diagramas de estoque e fluxo ou Metamodelos de Dinâmica de Sistemas. A estrutura dos modelos é apresentada visualmente através de diagramas e os resultados da simulação em tabelas e gráficos de linhas.

Este capítulo apresenta na Seção 1.1 as motivações que levaram ao início deste trabalho. Na Seção 1.2 mostramos os objetivos que pretendemos alcançar através das propostas expostas na Seção 1.3. Já na Seção 1.4 expõe as nossas justificativas. Por fim, a Seção 1.5 apresenta toda a organização deste trabalho.

1.1 Motivação

Ao longo das últimas duas décadas, diversas técnicas de modelagem foram usadas para tentar capturar os comportamentos dinâmicos observados durante os processos de desenvolvimento de software. Essas foram usadas em uma ampla gama de aplicações, como: planejamento; controle e operação; gerenciamento estratégico; controle de qualidade; melhoria de processo; análise de riscos; transferência de conhecimento e construção de jogos gerenciais (KELLNER; R; D, 1999). Essas aplicações são frequentemente construídas utilizando como base um ambiente de modelagem e simulação, seja na forma de um editor e simulador de modelos ou na forma de bibliotecas desenvolvidas por terceiros.

O esforço necessário para coordenar a operação das ferramentas utilizadas como base para as aplicações acrescenta um fator complicador em projetos onde uma maior interação com a estrutura dos modelos é exigida. Muitas vezes exige lidar com os vínculos com licenças

proprietárias ou projetos que tiveram seu desenvolvimento interrompido. Esses fatores incluem uma limitação extra no desenvolvimento ou inviabilizam a reprodução dos resultados quando os autores do trabalho original não são acessíveis. De forma particular, aplicações desenvolvidas para uso educacional ou em ambientes experimentais podem se beneficiar de uma arquitetura de modelagem e simulação aberta, eliminando a necessidade de construção de uma estrutura para integração com soluções fechadas.

Acreditamos que fornecer uma biblioteca aberta ao invés de uma aplicação de uso geral, permite ao desenvolvedor dos modelos e aplicações usar sua criatividade para explorar outras possibilidades além das definidas inicialmente no momento de criação de um ambiente de uso geral.

Apesar das referências apresentadas estarem relacionadas ao domínio de processos de software, a infraestrutura interdiciplinar aqui proposta pode ser utilizada em outros domínios onde os modelos envolvidos possuam comportamento dinâmico, como os encontrados em Ciências Sociais, Econômicas, Biológicas, Química, Física e Engenharias em geral.

1.2 Objetivos desta Dissertação

O principal objetivo deste trabalho é criar uma infraestrutura computacional que disponibiliza o processo básico de modelagem e simulação na forma de uma biblioteca. A estrutura desta biblioteca deve ser flexível, independente de domínio e extensível. A flexibilidade deve ser alcançada de forma a permitir especialização dos elementos descritores do processo e possibilitar sua modificação para ser inserida em outras aplicações. A independência de domínio deve permitir que as estruturas dos modelos suportados não sejam vinculadas a um domínio específico. A extensibilidade deve permitir o suporte a novas linguagens de descrição de modelos para suprir as necessidades específicas do estudo em que estiverem sendo inseridas.

Esta biblioteca deve suportar, de forma transparente em um certo nível de abstração, duas linguagens de modelos de sistemas dinâmicos. A primeira delas, considerada a base da Dinâmica de Sistemas, descreve os diagramas tradicionais de estoque e fluxo com seu limitado conjunto de elementos. A segunda linguagem, os metamodelos de Dinâmica de Sistemas ou linguagem estendida de Dinâmica de Sistemas, apresenta uma estrutura mais complexa. Nela, os sistemas podem ser representados em até três tipos diferentes de modelo: modelos e domínio; modelos de instância e modelos de cenários.

Para exemplificar o uso desta biblioteca, este trabalho deve incluir uma ferramenta compu-

tacional que, através do uso de modelagem e simulação, auxilie o ensino de gerenciamento de processos de desenvolvimento de software. Para tanto, construímos um protótipo de aplicação que permite aos usuários interagir com modelos dinâmicos que representam comportamentos comuns encontrados em um processo de desenvolvimento de software.

Os objetivos deste trabalho podem ser sumarizados da seguinte forma:

- Infraestrutura para Modelagem e Simulação:
  - Capturar a estrutura básica de um processo de simulação:
    - Descrição de modelos;
    - Simulação via método numérico;
    - Descrição dos resultados.
  - Permitir mais de uma linguagem de modelagem:
    - Diagramas de estoque e fluxo de Dinâmica de Sistemas;
    - Metamodelos de Dinâmica de Sistemas.
- Simulador de Modelos
  - Ser construído usando a infraestrutura criada;
  - Simular modelos de duas linguagens diferentes;
  - Alterar o métodos de simulação como configuração;
  - Apresentar a estrutura dos modelos;
  - Apresentar os resultados graficamente;
  - Permitir a edição visual dos modelos.

1.3 Soluções Propostas

Este trabalho apresenta como contribuição a *JynaCore API*[1] biblioteca com código livre desenvolvida em Java (SUN MICROSYSTEMS INC., 2009a), a partir da descrição das linguagens de modelagem de estoque e fluxo e dos modelos estendidos da Dinâmica de Sistemas. Essa biblioteca apresenta as interfaces e implementações que permitem a descrição e simulação de modelos dinâmicos, independentes de domínio, utilizando uma das linguagens suportadas.

[1] Contração das palavras em inglês de Java (linguagem de programação), Dynamics (dinâmica), Core (núcleo) e API (sigla de interface para programação de aplicativos).

Como uma segunda contribuição que age como prova de conceito das funcionalidades básicas da biblioteca, criamos um ambiente de simulação *JynacoreSim* que trabalha com modelos de estoque e fluxo de Dinâmica de Sistemas e com os metamodelos de Dinâmica de Sistemas. Esse ambiente define um padrão de arquivos próprio, em XML, para trabalhar com os modelos e apresenta os resultados da simulação na forma de dados tabulares e gráficos de linha.

Este trabalho apresenta uma revisão dos trabalhos relacionados a modelagem de processos de software nas últimas duas décadas e uma revisão teórica das linguagens de modelagem implementadas. Nessa última, são expostos modelos básicos que exemplificam a construção de estruturas que servem de base para o entendimento de modelos mais complexos relacionados a processos de desenvolvimento de software disponíveis na literatura.

As contribuições deste trabalho podem ser sumarizadas da segunte forma:

- Revisão da literatura de modelagem de processos de software
  - Principais trabalhos;
  - Abordagens utilizadas;
  - Ferramentas disponíveis para Dinâmica de Sistemas.
- Desenvolvimento da JynaCore API
  - Biblioteca para Modelagem e Simulação;
  - Desenvolvido em Java;
  - Código Livre;
  - Modelos e Simuladores de Dinâmica de Sistemas;
  - Modelos e Simuladores de Metamodelos de Dinâmica de Sistemas;
  - Não realiza compilação dos Metamodelos;
  - Pode ser estendida para outras linguagens de modelos;
  - Pode ser estendida para dar suporte a outros métodos numéricos; (inclui: Euler explícito e Runge-Kutta ordem 4).
- Desenvolvimento do JynacoreSim
  - Ambiente para simulação de modelos;
  - Desenvolvido em Java;

- Simula modelos em Dinâmica de Sistemas e Metamodelos de forma transparente;
- Permite selecionar o método numérico de forma transparente;
- Exibe a estrutura em diagramas gráficos;
- Exibe resultados em tabelas e gráficos de linhas;
- Não permite a edição visual dos modelos.

1.4 Justificativa

Os problemas que motivaram o desenvolvimento deste trabalho foram observados, inicialmente, durante nossa participação no projeto Ciência Viva (VILLELA, 2005). O projeto aplica técnicas de modelagem e simulação como ferramenta didática para o ensino de ciências no ensino fundamental e médio. As opções comerciais disponíveis na época mostravam-se extremamente eficientes para a construção dos modelos, porém, deixavam a desejar quando queríamos um maior grau de liberdade para criar e adaptar modelos e simuladores a outras formas de apresentação (como jogos educativos). O fato da maioria das ferramentas disponíveis serem proprietárias também foi um fator limitante. Geralmente incluíam uma série de restrições estruturais como falta de acesso ao código fonte (necessário para se adaptar a ferramenta), uso estritamente acadêmico, complexidade dos modelos restrita ou um vínculo com um período de avaliação. As ferramentas comerciais disponíveis inseriam um custo extra com a compra de licenças ou assinaturas de serviços, complicando ainda mais o processo de aplicação das técnicas em escolas que não poderiam ou não queriam lidar com gastos extras. Isto nos levou a desenvolver aplicações dos processos de modelagem, inalteráveis pelos alunos e professores, que cumpriam sua finalidade, mas não permitiam modificações em seus modelos.

O uso de modelagem e simulação para estudo de processos de desenvolvimento de software utilizando Dinâmica de Sistemas, permitiu aproveitar nossa experiência anterior e abriu toda uma nova possibilidade de prover ferramentas para estudos nas áreas de engenharia de software, melhoria de processos, análise de riscos, métodos numéricos e representação do conhecimento. O trabalho de Barros (2001) nos permitiu expandir a nossa capacidade de representação dos modelos de Dinâmica de Sistemas através de sua nova linguagem estendida. As extensões propostas em tal trabalho, ao serem introduzidas no processo de construção de modelos, permitem que a modelagem realize “um salto” no nível de abstração dos modelos: pessoas com diferentes graus de conhecimento das técnicas de modelagem passam a poder discutir sobre o comportamento do sistema através de modelos diferentes, mas diretamente relacionados. Um especialista, no domínio e na linguagem de modelagem, descreve todo o comportamento

dinâmico, e um usuário, com conhecimento limitado ao domínio modelado, interage com a simulação sem a necessidade de conhecer as técnicas de modelagem empregadas.

Optamos por dar à nossa ferramenta a forma de uma nova biblioteca que encapsula o processo básico de modelagem e simulação. Acreditamos que, por estarmos inseridos em um curso multidisciplinar, ao fornecer uma ferramenta flexível o bastante para ser inserida em qualquer aplicação, contribuímos para a difusão do conhecimento e das técnicas empregadas. O estudo dos modelos de processos de desenvolvimento de software são usados para ilustrar a utilização de nossa solução. Isto, entretanto, não impede que ela seja aplicada em outros domínios de conhecimento que tenham como base de comportamento os sistemas dinâmicos. Porém, tais estudos estão fora do escopo deste trabalho.

A linguagem estendida de Dinâmica de Sistemas foi implementada originalmente, pelo compilador *Hector* (BARROS, 2001, p.221). Este converte os modelos estendidos[2] para os construtores padrão de Dinâmica de Sistemas. Uma vez compilados, são simulados ou convertidos para arquivos de outras aplicações proprietárias que lidam com modelos dinâmicos. A nossa implementação da linguagem estendida de Dinâmica de Sistemas foi feita de forma que não é necessária a compilação: os modelos estendidos são simulados diretamente a partir de sua descrição. Optamos por dar à linguagem estendida o mesmo status da Dinâmica de Sistemas, reescrevendo toda a estrutura de suporte e simulação. Isto é feito para validar que linguagens mais complexas estruturalmente, como a linguagem estendida, podem ser tratadas de forma transparente pela JynaCore API.

O ambiente de simulação *JynacoreSim* foi desenvolvido, primeiramente para ilustrar e avaliar o funcionamento da JynaCore API. Ele dá o primeiro passo no sentido de criar um ambiente onde alunos de um curso que envolva gerência de processos possam ter um contato com técnicas de modelagem de gerenciamento de projetos. O JynacoreSim permite a simulação de modelos dinâmicos, usando Dinâmica de Sistemas ou metamodelos, de forma transparente. Os modelos desenvolvidos durante a revisão teórica são descritos em arquivos XML e disponibilizados para estudo e modificação.

1.5 Organização da Dissertação

Este trabalho está organizado em seis capítulos e três apêndices. O primeiro Capítulo consiste nesta introdução, acrescido de uma revisão das técnicas utilizadas para modelagem e si-

[2]No trabalho original estes são chamados de metamodelos de Dinâmica de Sistemas. Entretanto, também usamos a denominação modelos em linguagem estendida de Dinâmica de Sistemas ou modelos estendidos.

mulação de desenvolvimento de processos de software, usados para aperfeiçoamento ou treinamento de gerentes de projetos.

O Capítulo 2 realiza uma revisão de alguns dos principais trabalhos relacionados a modelagem de processos de desenvolvimento software agrupados pelas técnicas utilizadas para a captura dos comportamentos de um processo por modelos formais.

O Capítulo 3 apresenta os diagramas causais e de estoque e fluxo que integram a linguagem de Dinâmica de Sistemas. Sendo apresentado um breve histórico e os elementos componentes de cada diagrama e exemplos que visam reproduzir comportamentos comuns encontrados na literatura de processos de desenvolvimento de software.

O Capítulo 4 expõe os modelos de domínio, instância e cenários que compõem os chamados metamodelos de Dinâmica de Sistemas, neste trabalho referenciados como linguagem estendida de Dinâmica de Sistemas. A extensão aos diagramas de estoque e fluxo é apresentada seguida de exemplos de modelos dos elementos de processos de software, a fim de ilustrar o uso da linguagem e servirem de base para construção de modelos mais complexos.

O Capítulo 5 mostra a arquitetura básica da biblioteca JynaCore API, que consiste em um conjunto de interfaces representando modelos em diagramas de estoque e fluxo de Dinâmica de Sistemas e modelos estendidos de domínio, instância e cenários. São apresentadas as interfaces auxiliares usadas para realizar um fluxo básico de simulação de modelos, incluindo, um exemplo de aplicação. Ao final do capítulo, apresentamos o JynacoreSim: protótipo de aplicação construído com a nossa biblioteca para ilustrar seu uso como ambiente educacional, onde alunos podem interagir com os modelos construídos nos capítulos anteriores a fim de construir seus conhecimentos sobre os comportamentos dinâmicos dos modelos de processos de desenvolvimento de software encontrados na literatura.

Por último, o Capítulo 6, trata das considerações finais deste trabalho, suas contribuições, limitações e sugestões para trabalhos futuros.

Construímos no Apêndice A um exemplo de aplicação que, a partir de um arquivo de modelo, apresenta o resultado da simulação na saída padrão. Esta aplicação pode trabalhar com modelos de estoque e fluxo e de instância. Exploramos também, uma opção para torná-la independente da linguagem do modelo.

No Apêndice B trabalhamos um modelo completo desenvolvido programaticamente para ilustrar a construção e simulação de um diagrama de estoque e fluxo usando as interfaces para Dinâmica de Sistemas da biblioteca Jynacore API. Apresentamos também, o mesmo modelo

utilizando um padrão de arquivo próprio no formato XML.

O Apêndice C apresenta um modelo completo para construção e simulação de modelos de domínio, instância e cenário de metamodelos desenvolvidos programaticamente usando os elementos para os modelos estendidos da biblioteca Jynacore API. Novamente, apresentamos o mesmo modelo construído utilizando um outro padrão próprio no formato XML.

2 *Modelos de Processos de Software*

Modelar processos é representar, de forma abstrata, a arquitetura, estrutura, comportamento ou definição de um conjunto de tarefas destinadas a cumprir um fim específico. O modelo surge para responder a um conjunto de questões bem definido sobre o sistema real. A modelagem abstrata de processos começa com o trabalho pioneiro de Osterweil (1987) onde o termo "programação de processos" passa a designar as técnicas sistemáticas e formalismos usados de forma a descrever conjuntos de tarefas para desenvolvimento de software.

Segundo Madachy (2008, p.23), modelos de processos de software são encontrados, frequentemente, na forma de modelos quantitativos. Um modelo foca um aspecto particular de um processo implementado (como é) ou planejado (como deverá ser). São usados quando os custos, riscos ou logística para manipular o processo real são proibitivos. Como uma abstração representa apenas os aspectos de maior interesse do estudo, os modelos são muito úteis quando utilizados em ambientes educacionais ou para aperfeiçoamento de gerentes. Assim, evita-se colocá-los diretamente em ação em um projeto real, ignorando os riscos inerentes à falta de experiência.

Barros (2001) destaca que o gerenciamento de projetos de software é uma atividade fortemente baseada em conhecimento. Gerentes usam suas habilidades e sua experiência para tomar decisões enquanto sua equipe executa o processo de desenvolvimento de software. Os gerentes mais experientes, geralmente, obtém mais sucesso no controle de projetos do que gerentes inexperientes. Os modelos de processos de software capturam as interrelações de grupos de pessoas, artefatos e tarefas e podem servir como meio de registro do conhecimento gerencial em uma linguagem formal. Desta forma, os comportamentos comuns encontrados na literatura podem ser descritos, guardados e experimentados em ambientes simulados. Assim, de posse dos modelos criados por gerentes experientes, outros gerentes podem experimentá-los em simuladores de processos e a diferença de experiência entre eles pode ser minimizada.

Este capítulo faz uma revisão dos trabalhos, técnicas e aplicativos que foram desenvolvidos ou adaptados para modelar processos de software. A Seção 2.1 fala sobre algumas tentativas de

classificar os trabalhos quanto à sua aplicação ou abstração da técnicas utilizadas. A Seção 2.2 faz o acompanhamento do histórico de trabalhos e técnicas utilizadas para modelar e simular processos de software. A Seção 2.3 apresenta alguns dos ambientes de simulação e simuladores de projetos existentes. Por fim, a Seção 2.4 apresenta as conclusões do capítulo.

2.1 Taxonomias de Modelos de Processos

A definição de uma taxonomia comum para modelos de processos de software é ainda uma questão em aberto. Segundo Kellner, R e D (1999), as abordagens podem ser classificadas quanto à seu propósito: gerenciamento estratégico; planejamento; controle e operação; melhoria de processos ou adoção de tecnologia; treinamento e aprendizagem. O gerenciamento estratégico envolve o impacto a longo prazo de políticas e iniciativas atuais ou propostas. O planejamento tenta prever esforço, custo, cronograma e qualidade no início do projeto para escolher, modificar e compor o melhor processo para um projeto específico. O controle e operação envolve em acompanhar um projeto que já teve início, comparando os indicadores atuais com os planejados para decidir quando atuar corretivamente. Em melhoria de processo e adoção de tecnologia compara os resultados de alternativas de processos e tecnologias tanto *a priori* quanto *ex post*. E por fim, treinamento e aprendizagem permite criar ambientes de instrução onde é possível praticar ou aprender os conceitos por simulações ao invés de exposição de casos particulares.

Para os casos como o uso educacional, projetos inovadores, novos processos ou pesquisa em ambientes acadêmicos, a construção e simulação de modelos e ferramentas é, antes de tudo, uma atividade experimental. Envolvendo tentativa e erro, mudanças de tecnologias, métodos e comparação de resultados. Estudos nestas áreas podem envolver esforços em diversos níveis de conhecimento, indo das tecnologias que dão suporte à descrição dos modelos e métodos de simulação até a definição de metodologias de gerenciamento de processos auxiliadas por ambientes de simulação integrados.

Em Lakey (2003), o autor classifica alguns modelos de processos em níveis que mostram o quanto as técnicas usadas estão próximas de um sistema real. A tabela 2.1 apresenta esta classificação partindo de modelos com alto nível de abstração (topo da tabela) para modelos mais concretos (final da tabela).

Inicialmente, temos modelos que não enxergam a estrutura do processo. Nos sistemas lineares tenta-se associar as saídas do processo às variáveis independentes (presumidas). Nesta

Modelo	Ferramenta
Equações Lineares Simples	Regressão estatística
	Variáveis independentes
Relações Multivariáveis Complexas	Monte Carlo
	Redes Neurais Artificiais
Modelos de Processos Estáticos	Dados históricos de cada fase
	Relacionamentos entre processos e variáveis
Modelos de Processos Dinâmicos	Dinâmica de Sistemas
	Representação em variáveis contínuas
Modelos de Processos Híbridos	Eventos Discretos e Dinâmica de Sistemas
Modelos de Processos Semiautomatizados	Sistema parcialmente integrado ao modelo
	Variáveis simuladas
Modelos de Processos Automatizados	Modelo integrado ao sistema em tempo real

Tabela 2.1: Tipos de simulação de modelos de processos de software baseados nas técnicas empregadas. Adaptado de Lakey (2003).

abordagem, o processo se comporta como uma caixa preta para os modelos. A próxima abordagem, que ainda trabalha enxergando apenas essa caixa preta, usa representações matemáticas mais sofisticadas com equações não-lineares ou redes neurais artificiais para realizar os relacionamentos das saídas com as variáveis. O próximo nível, decompõe a estrutura do processo nas suas fases ou atividades principais. Apesar de não ter acesso à estrutura de cada atividade, consegue fazer a relação entre os processos e variáveis por dados históricos obtidos em cada atividade do ciclo de vida. No nível seguinte, ainda segundo Lakey (2003), a estrutura das atividades é exposta e modelada de forma a apresentar seu comportamento endógeno[1]. Neste nível os modelos têm uma representação contínua das variáveis e são mais comumente encontrados descritos em Dinâmica de Sistemas. Já o próximo nível usa uma abordagem híbrida de Dinâmica de Sistemas e eventos discretos: cada fase do processo possui uma série de eventos discretos contendo um conjunto de modelos em Dinâmica de Sistemas. No nível posterior, os modelos semiautomatizados incluem parcialmente em sua descrição representações reais do sistema como parâmetros. Assim, conseguem realizar simulações das variáveis para reproduzir o comportamento a partir de um estado real. No último nível, o modelo do sistema praticamente é o sistema. Todas as variáveis relevantes ao processo são inseridas no modelo e atualizadas em tempo real.

Para que técnicas e ferramentas atinjam este nível, essas devem ser idealizadas visando um alto grau de interoperabilidade, de forma que, conectando-as umas às outras, o processo de modelagem e simulação se torne apenas um bloco básico dentro de um conjunto maior de tarefas. Acreditamos que isto possa ser alcançado se o usuário do ambiente de modelagem

[1] O comportamento externo do sistema emerge das relações dos seus elementos estruturais.

tiver a liberdade de adaptar a ferramenta para capturar diversos tipos de modelos ou definir seu próprio formato de dados. Além disso, ele deve ser capaz de alterar qualquer parte do processo de simulação que exija um ajuste mais fino, como uma maior descrição dos modelos ou maior velocidade nos métodos de simulação.

As aplicações que usam simulação de modelos podem ser construídas de duas formas: especializadas ou de uso geral. As especializadas focam a atenção do usuário nos conceitos mais importantes para um determinado propósito. Já as aplicações de uso geral, servem como ambientes para construção de modelos, expondo os construtores básicos sem ligá-los a um domínio de conhecimento específico. Em ambos os casos, internamente, há uma associação da descrição formal dos modelos às técnicas que permitem a simulação.

A figura 2.1 apresenta os conjuntos formados por técnicas de simulação quantitativa (regressão estatística, métodos numéricos, eventos discretos, etc.) e as de representação do conhecimento (linguagens formais, ontologias, diagramas, etc.). Algumas abordagens ficam descritas na interseção destes dois conjuntos provendo, ao mesmo tempo, a representação formal e suporte à simulação. As aplicações são construídas envolvendo um pequeno conjunto das técnicas disponíveis: selecionam-se as que apresentam melhor resultado para o propósito do modelo.

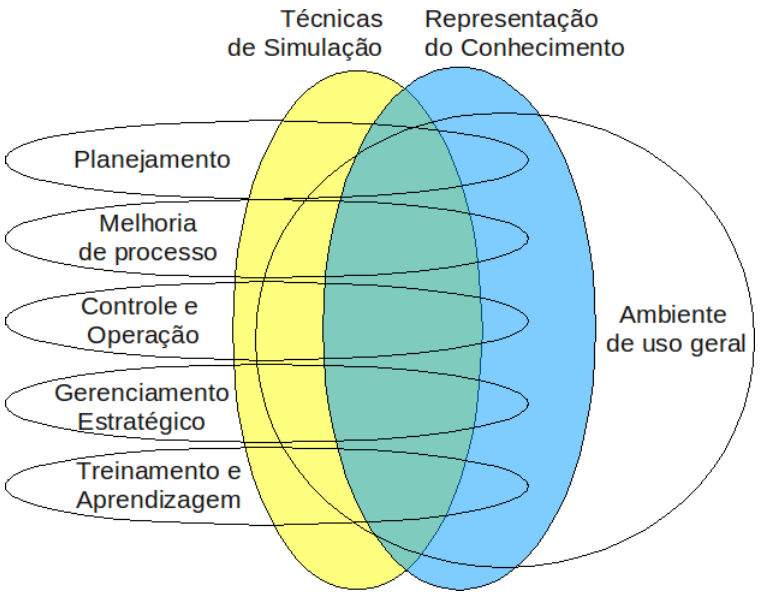

Figura 2.1: Aplicações que usam parte de um conjunto de técnicas de modelagem e simulação para um determinado propósito.

Como exemplo, incluímos, ainda na figura 2.1, um sumário com as classificações de propósitos para uso de simulação de modelos encontrados no domínio de processos de software: gerenciamento estratégico; planejamento; melhoria de processo ou adoção de tecnologia; treinamento e aprendizagem (MADACHY, 2008, p.26). Cada aplicação adota a abordagem que

lhe for mais conveniente, mas usa, pelo menos, uma técnica de modelagem e, quando for o caso, uma técnica de simulação associada. Frequentemente, as técnicas de modelagem e simulação são implementadas para cada aplicação e mantidas ocultas como uma característica não importante para o propósito do modelo. Esta abordagem atrasa a produção de aplicações experimentais e dificulta o reaproveitamento de soluções já encontradas.

O desenvolvimento de ferramentas que permitam encapsular o processo de descrição de modelos e técnicas de simulação pode contribuir para o surgimento de um conjunto de aplicações que explorem, de forma transparente, diversas técnicas e abordagens. O uso de um padrão, mesmo que em um nível mais conceitual, pode facilitar a criação de meios para integração de ambientes existentes à simulação de modelos.

2.2 Uso de modelagem e simulação em Processos de Desenvolvimento de Software

A modelagem abstrata de processos de Osterweil (1987) usa técnicas para produzir objetos de processos são muito parecidas com as de software: são geradas instâncias destes objetos componentes, de uso geral, e estes são executados em processos diferentes, mantendo o mesmo comportamento.

Modelos para estimar custos e cronogramas como Constructive Cost Model II, COCOMOII (BOEHM et al., 2000 apud MADACHY, 2008, p.25), SEER (GALORATH; EVANS, 2006 apud MADACHY, 2008), True-S (PRICE SYSTEMS, 2005 apud MADACHY, 2008, p.25) fornecem um valioso conjunto de idéias sobre as relações envolvidas no processo de software. Entretanto, a característica estática destes modelos, dificulta o entendimento de diversas situações dinâmicas complexas. Desta forma, modelos de processos devem ir além da representação estática para suportar análise de processos (MADACHY, 2008, p.25).

Curtis, Kellner e Over (1992) listaram cinco abordagens dinâmicas para representar informações de processos: modelos de programação (programação de processos); modelos funcionais; modelos baseados em planos; modelos em redes de Petri (redes de interações de papéis) e Dinâmica de Sistemas.O Articulator (MI; SCACCHI, 1993) é um ambiente de computação baseado em conhecimento, que usa inteligência artificial para agendamento de sistemas de produção durante a modelagem, análise e simulação de processos organizacionais. Classes de recursos organizacionais são modelados usando uma representação de conhecimento orientada a objetos.

Madachy (2008) lembra também que outras técnicas foram usadas para modelar processos como: as baseadas em estados; simulação de eventos discretos gerais; linguagens baseadas em regras; agendamento; modelos de fila e gerenciamento de projetos (PERT e CPM). A abordagem do grupo Software Engineering Institute - SEI é baseada em transições de estados com eventos e gatilhos, análise de diagramas e sistemas de projetos com modelagem de dados para dar suporte à representação, análise e simulação de processos (KELLNER, 1991; RAFFO, 1995). Essa abordagem permite uma simulação quantitativa que combina perspectivas funcionais, comportamentais e organizacionais.

Na última década, a modelagem de processos de software teve como técnicas mais utilizadas a Dinâmica de Sistemas e os eventos discretos (KELLNER; R; D, 1999). Entretanto, tem havido um aumento do número de trabalhos utilizando agentes autônomos (ZHAO; CHAN; LI, 2005). Abordagens híbridas entre Dinâmica de Sistemas e Eventos Discretos foram exploradas em Martin e Raffo (2000) e Lakey (2003). Um outro estudo que usa abordagem híbrida de dois níveis foi proposta por Donzelli e Iazeolla (2000) e combina métodos analíticos contínuos a eventos discretos. Em um nível mais alto de abstração, o processo é modelado por uma fila de eventos discretos que captura as atividades comportamentais (estados de serviços), suas interações e artefatos trocados. Detalhes de implementação são dados em baixo nível de abstração, onde métodos analíticos e contínuos são usados. Little-JIL (WISE, 2000) é uma linguagem gráfica para programação de processos que coordena as atividades de agentes autômatos e seu uso de recursos durante o cumprimento de uma tarefa (WISE et al., 2000; WISE, 2006).

O uso de Dinâmica de Sistemas (FORRESTER, 1961) em processos de software teve início com Tarek Abdel-Hamid, para sua tese de doutorado no Machassussets Institute of Technology. Esta tese foi defendida em 1984 e ele escreveu alguns artigos antes de publicar o livro Software Project Dynamics com Stuart Madnick em 1991 (ABDEL-HAMID; MADNICK, 1991). Desde então, uma centena de publicações lidaram com o processo de software usando Dinâmica de Sistemas (MADACHY, 2008, Apendix C). A Dinâmica de Sistemas facilita o entendimento e a comunicação da estrutura do processo a humanos. Os adeptos advogam que o entendimento do comportamento do sistema e da estrutura é o principal valor agregado ao uso da Dinâmica de Sistemas como ferramenta de predição.

Barros (2001) propõe uma metodologia onde o gerenciamento de projetos de desenvolvimento de software e técnicas de análise de risco são auxiliadas pelo uso de ferramentas de modelagem e simulação. O trabalho cria uma extensão da linguagem de Dinâmica de Sistemas, chamada de *metamodelos*, que inclui elementos que aumentam a abstração dos modelos, facilitando a transmissão de conhecimento gerencial em uma linguagem formal. A abordagem provê

um compilador que traduz os metamodelos para os construtores de Dinâmica de Sistemas para, então, serem simulados.

2.3 Simuladores de Processos de Software

O uso de elementos visuais presentes em jogos como alternativa aos gráficos e tabelas, comuns nas ferramentas tradicionais de simulação, apresenta uma outra abordagem para construção de ambientes para treinamento e ensino na forma de "simuladores de vôo".

O *Manager Master* (BARROS, 2001, p.228) é um emulador de projetos usado para realizar o estudo experimental para análise de viabilidade das técnicas de modelagem e simulação propostas para treinamento de gerentes de projetos de software. Utiliza um motor de simulação dos metamodelos de Dinâmica de Sistemas para capturar as decisões quanto à equipe de desenvolvedores dentro um conjunto de candidatos e as quais atividades estes estão vinculados.

O *Incredible Manager* (DANTAS, 2003) é um jogo de simulação usado como instrumento para treinamento de gerentes de projetos de software. Construído em Java, utiliza um motor de simulação que orquestra, internamente, o compilador e simulador dos metamodelos de Dinâmica de Sistemas. Através da interface do jogo, as interações do usuário são associadas à recompilação de metamodelos. O modelo resultante obtém seus parâmetros e estados a partir do resultado prévio da simulação.

O SimSE (NAVARRO; van der Hoek, 2004; NAVARRO, 2008) é um ambiente de simulação gráfico, interativo, educacional, extensível, baseado em jogos, desenvolvido para ensinar processos de engenharia de software. Ele utiliza uma linguagem própria de modelagem para descrição dos objetos, regras, figuras e atividades de processos. Os modelos podem ser criados através de um editor para representar uma grande gama de processos de software. Inclui, também, a representação gráfica que simula o ambiente de trabalho com seus desenvolvedores e clientes. O SimSE cria aplicações Java independentes, contendo o jogo, quando os modelos de processos são compilados.

2.4 Conclusões Parciais

Durante as duas últimas décadas observou-se que modelos e simulações de processos de desenvolvimento de software se tornaram uma alternativa viável para capturar e estudar os comportamentos dinâmicos inerentes de um processo real.

Foram utilizadas diversas abordagens para a confecção e simulação de modelos de processos, como técnicas de análise algorítmica, transições de estados, eventos discretos, redes de Petri, inteligência artificial, agentes autômatos, variáveis contínuas, modelos estocásticos e Dinâmica de Sistemas.

As aplicações variam desde de ambientes de modelagem de uso geral, onde é possível construir modelos e visualizar os resultados da simulação na forma de gráficos e tabelas até ambientes que realizam a simulação de forma a reproduzir os elementos de um processo real, nos chamados “simuladores de vôo”.

Não há uma solução que seja considerada superior em todas as aplicações e sua escolha é diretamente ligada às questões que o modelo deve responder e experiência dos condutores do estudo com as ferramentas e recursos disponíveis.

As técnicas de modelagem de processos de software baseadas em Dinâmica de Sistemas e em sua linguagem estendida são de especial interesse deste trabalho. Portanto, revisamos os conceitos teóricos dos diagramas de Dinâmica de Sistemas no Capítulo 3 e sobre os metamodelos de Dinâmica de Sistemas no Capítulo 4.

3 *Introdução à Dinâmica de Sistemas aplicada aos Processos de Software*

Durante a década de 60, Jay Forrester, então professor da Escola de Administração do Massachusetts Institute of Technology, desenvolveu a Dinâmica de Sistemas (FORRESTER, 1961), uma ferramenta para expressar comportamentos econômicos e sociais complexos. Forrester notou que as pessoas não estavam prontas para interpretar respostas e comportamentos de sistemas complexos, onde um grande número de variáveis está envolvido. E, uma má identificação das causas de um efeito, pode tornar um processo de tomada de decisão propenso a erros, no qual uma intervenção mais intuitiva sobre uma parte do sistema pode conduzir a resultados muito distantes dos desejados.

Forrester criou, em seu livro, ferramentas de modelagem que permitiram às pessoas compreender e aperfeiçoar sistemas sociais da mesma forma que engenheiros utilizam princípios de Engenharia de Controle[1] para desenvolver e melhorar modelos mecânicos ou eletrônicos (ARONSON, 1998). Amparada por linguagens gráficas e simulações por computador, a Dinâmica de Sistemas permitiu que sistemas complexos pudessem ser modelados e ter seu comportamento estudado sem o uso direto de equações diferenciais, tornando mais simples a criação de modelos e permitindo aos especialistas de domínio se concentrarem na sua construção e nos efeitos que as mudanças pontuais causam no comportamento geral, sem se preocupar em resolver de forma analitica os sistemas de equações.

A Dinâmica de Sistemas trabalha baseada no conceito de *sistema*. Segundo Senge (1990), um sistema é um subconjunto da realidade que é o foco da análise. Já para Madachy (2008, p.6), um sistema é tido como um conjunto de componentes que interagem entre si para cumprir uma função que não são capazes de desempenhar sozinhos.

Um sistema é descrito por parâmetros e variáveis. Os parâmetros são medidas quantificadas independentes que influenciam nas entradas ou mudam a estrutura do sistema. As variáveis

[1] A tradicional análise e síntese do comportamento quantitativo de sistemas é objeto de estudo da Teoria de Controle. As bases da chamada Teoria de Controle Moderno são encontradas em Ogata (1982)

são medidas indiretas que são tomadas em relação aos parâmetros ou a outras variáveis. Ao conjunto de variáveis e parâmetros necessários para se descrever um sistema em qualquer ponto do tempo damos o nome de estado do sistema. Já o conjunto de informações trocado pelas variáveis recebe o nome de estrutura do sistema.

Sistemas são considerados "abertos" quando suas saídas não influenciam no seu comportamento. Ou seja, não possuem conhecimento de seu estado passado. Os sistemas "fechados", retroalimentados ou realimentados são aqueles que suas saídas influenciam a sua operação através de um laço de realimentação de informação[2], usando as saídas anteriores para controlar suas saídas futuras.

Modelos são construídos como uma simplificação da realidade para responder a um conjunto de questões bem definido. No caso de modelos que representam sistemas dinâmicos, estes conduzem a modelos matemáticos que descrevem como seu estado varia com o tempo, ou seja, seu comportamento devido a uma certa configuração da estrutura e variáveis. Ao ato de acompanhar as mudanças do estado de um modelo de sistema, seja analiticamente ou seja via métodos numéricos em um computador, chamamos de simular o sistema ou simplesmente, simulação (MADACHY, 2008).

Modelos de sistemas podem ser construídos em vários níveis de abstração e representar qualquer nível de complexidade. Podemos tomar uma pessoa como um sistema complexo de funções fisiológicas e mentais e modelar como seu humor afeta algumas funções vitais. Em um modelo de uma cidade ou um país, podemos assumir o fluxo de pessoas que se mudam em função das condições econômicas locais como um sistema. Veremos, mais detalhadamente, como analisar e modelar um processo como um sistema, para estudar como este se comporta com a variação de seus parâmetros e estrutura.

Este capítulo expõe os conceitos básicos de Dinâmica de Sistemas como metodologia de construção de modelos em suas duas linguagens gráficas: os *diagramas causais* e o *diagrama de estoque e fluxo*. Os modelos expostos apresentam comportamentos comuns encontrados na literatura e servem como exemplos básicos para nosso estudo. A Seção 3.1 mostra os conceitos que relacionam o pensamento sistêmico a processos de desenvolvimento de software. A Seção 3.2 apresenta os diagramas da Dinâmica de Sistemas: a Seção 3.2.1 apresenta os diagramas causais, usados para uma modelagem mais conceitual de um sistema e a Seção 3.2.2 apresenta as bases para construção de modelos dinâmicos através de diagramas de estoque e fluxo. Por fim, a Seção 3.4 apresenta as conclusões do capítulo.

[2]Do inglês "*information feedback loops*".

3.1 Pensamento Sistêmico em Processos de Software

Segundo Richmond (apud MADACHY, 2008, p.9), o *Pensamento Sistêmico* é a arte e ciência de se fazer inferências confiáveis sobre o comportamento de um sistema através da construção e aprofundamento do entendimento de sua estrutura fundamental. O pensamento sistêmico é ao mesmo tempo um paradigma e um método de aprendizagem que provê processos, linguagens e tecnologias para se criar o modelo. O pensamento sistêmico é uma forma de buscar e descobrir a estrutura de um sistema e inferir seu comportamento a partir dela. Pode ser descrito como um paradigma geral que usa princípios de Dinâmica de Sistemas para compreender a estrutura de um sistema qualquer.

De acordo com Madachy (2008, p.8), a idéia central é substituir o pensamento unidirecional por um laço de causa e efeito, considerando a interdependência entre as partes componentes do sistema (comportamento endógeno). Então busca-se uma orientação operacional ao invés da correlacional, onde os avanços das interrelações variam dinamicamente e não estaticamente. O autor enfatiza que o Pensamento Sistêmico é uma metodologia e a Dinâmica de Sistemas é uma ferramenta. Senge (1990, p.4) descreve o Pensamento Sistêmico como a quinta disciplina necessária para se integrar o domínio de pessoal, os modelos mentais, a visão compartilhada e a aprendizagem em equipe em uma organização capaz de aprender e evoluir. Colocar o conhecimento organizacional em modelos formais compartilhados pela equipe é a base para a melhoria do processo (SENGE, 1996 apud MADACHY, 2008, p.9).

A construção de modelos e sua simulação permitem que o gerente de um processo de software avalie *a priori* quais efeitos uma decisão pode criar no andamento de um projeto. Essas decisões gerenciais podem ser uma alteração nos prazos, nos custos ou nos atos que afetam diretamente o moral da equipe. As simulações por métodos numéricos são de propósito geral e podem ser utilizadas quando soluções analíticas são de difícil resolução ou mesmo impraticáveis devido à complexidade do sistema.

Um processo de desenvolvimento de software (incluindo evolução e gerência) pode ser visto como um sistema realimentado. Podemos considerar as especificações de requisitos (pontos funcionais, cronograma, orçamento, etc), recursos (pessoal e maquinário para produção e manutenção) e padrões de processo (métodos, políticas, procedimentos, etc.) como entradas deste sistema e os artefatos produzidos (incluindo defeitos) como suas saídas. Um sensor aparece na forma de um acompanhamento e análise de métricas de software. Uma ação gerencial age como um atuador no sentido de alinhar os resultados do processo atual com os resultados esperados. Além do processo de gerenciamento interno ao Processo de Software, Madachy

(2008) cita um processo de realimentação global, que aparece quando há mudanças de requisitos por parte dos usuários, tendências de mercado ou qualquer outra volatibilidade proveniente de uma fonte externa.

3.2 Diagramas básicos de um modelo em Dinâmica de Sistemas

A Dinâmica de Sistemas utiliza duas linguagens gráficas, na forma de diagramas, para extrair e modelar o comportamento de um sistema: os diagramas causais e os diagramas de estoque e fluxo. Esta seção apresenta a definição de cada uma das linguagens e apresenta a identidade visual mais comum dos diagramas encontrados na maioria das ferramentas de modelagem e simulação.

3.2.1 Diagrama Causal

Os Diagramas Causais são utilizados para representar de forma qualitativa os conceitos importantes (variáveis ou parâmetros) de um sistema e como esses afetam uns aos outros ao longo do tempo. Foram criados como forma de serem mais acessíveis que os diagramas de estoque e fluxo (vistos em detalhes na Seção 3.2.2) por não apresentarem as equações que vão descrever em detalhes os sistemas. Eles atingem um público maior pois dão suporte a uma forma mais livre de pensamento criativo e possuem um conjunto muito limitado de componentes. São compostos por palavras que expressam os conceitos do sistema, conectados por setas que representam as influências mútuas entre eles. Os laços de influência representam uma cadeia fechada de causa e efeito. Usados corretamente, esses diagramas podem ser utilizados de forma muito eficiente para explicar padrões de comportamentos dinâmicos e são ótimos para uma fase inicial de estudo de um sistema.

Segundo Madachy (2008, p.79), o uso de diagramas causais costuma ser muito debatido. Dependendo do estado e contexto do estudo, eles podem ser considerados muito úteis ou praticamente desnecessários. Sua utilidade depende dos objetivos, restrições e das pessoas envolvidas na fase da modelagem. A principal crítica ao uso de diagramas causais é que eles carecem de uma representação para diferenciar os conceitos que podem ser acumulados e transportados dentro do sistema (fluxos conservativos) dos que são obtidos indiretamente a partir de outros dados (fluxos não conservativos). A existência dessa simplificação pode levar a interpretações diferentes e errôneas do comportamento de um modelo, mesmo por especialistas.

Ao construir os Diagramas Causais, utiliza-se palavras para representar as variáveis do sistema a ser modelado. Em um Processo de Software podemos destacar conceitos como *Produtividade*, *Tarefas Concluídas*, *Qualidade*, *Retrabalho*[3]. Em seguida, as influências são traçadas como setas ligando os conceitos. E finalmente, símbolos de "+" e "-" adicionados ao lado das pontas de setas para indicar uma influência positiva ou negativa, respectivamente. A figura 3.1 apresenta o uso desses elementos em um exemplo dos comportamentos de um processo de software.

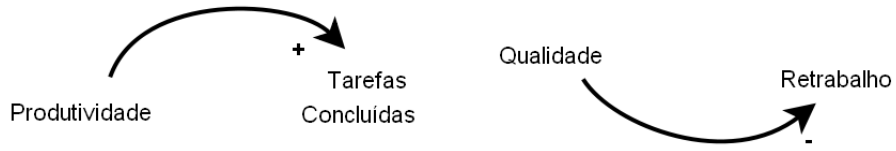

Figura 3.1: Diagrama causal: Relações entre algumas variáveis. À esquerda: Produtividade "aumenta" a quantidade de Tarefas Concluídas. À direita: Qualidade "diminui" a necessidade de Retrabalho. Adaptado de Madachy (2008, p.81).

A influência positiva significa que *Produtividade* "aumenta" a quantidade de *Tarefas Concluídas* ou afirmar que "mais" *Produtividade* gera "mais" *Tarefas Concluídas*. Uma influência negativa aparece na forma da *Qualidade* "diminui" o *Retrabalho*, ou que "mais" *Qualidade* implica em "menos" *Retrabalho*.

Segundo Madachy (2008), as relações expostas no diagrama da Figura 3.2 podem ser encontradas em um processo de desenvolvimento de software[4]. O conceito *Pessoal* representa a quantidade de membros de uma equipe de desenvolvimento de software. Consideraremos esse conceito como nosso único parâmetro de controle, ou seja, nossa única ação será decidir adicionar ou não pessoas à equipe do projeto.

A variável *Atrasos de Comunicação* indica a dificuldade dos membros em passar uma nova política, ação ou comunicar o estado de suas atividades para que a equipe trabalhe em sincronia. Isso ocorre devido ao excesso de canais de comunicação e está diretamente relacionado ao tamanho da equipe. Assim, assumimos que Pessoal influencia positivamente Atrasos de Comunicação, ou seja, "mais" Pessoal gera "mais" Atrasos de Comunicação. Por sua vez, Atrasos de Comunicação influencia negativamente a Produtividade pois pode inserir erros nos artefatos em

[3]Do inglês "rework".

[4]Este modelo representa a chamada Lei de Brooks (BROOKS, 1975 apud MADACHY, 2008): *"Adicionar tardiamente pessoal a um projeto atrasado o atrasa ainda mais"*. Tradução nossa. A Lei de Brooks é uma simplificação que expõe possíveis efeitos acumulados de uma série de fatores encontrados em um processo de software.

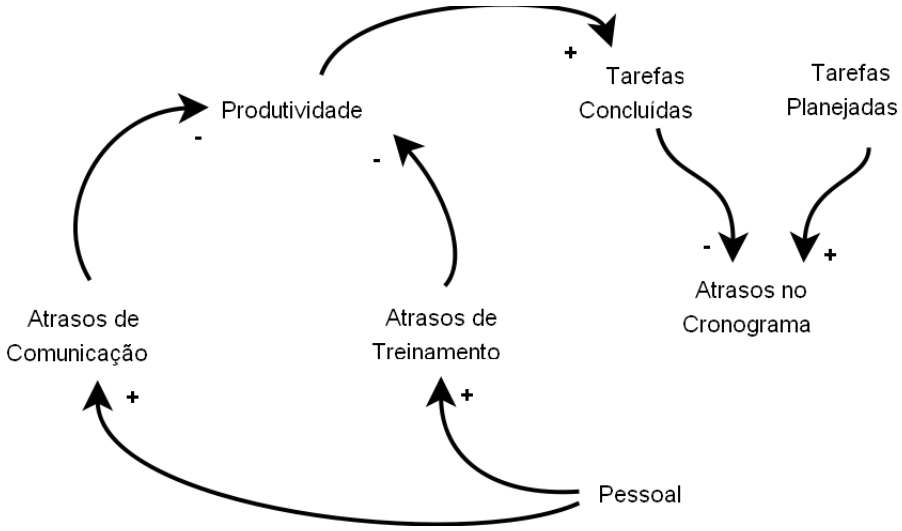

Figura 3.2: Diagrama causal: Alguns comportamentos encontrados em um processo de software. Adaptado de Madachy (2008, p.83).

produção, exigindo que a equipe direcione esforços para corrigí-los.

Outro efeito é devido aos novos membros que não contribuem com sua plena capacidade logo que integram um projeto. Os novatos necessitam de tempo para se habituarem com o trabalho que foi realizado anteriormente. Frequentemente os membros destacados para a produção são retirados temporariamente de suas funções para liderar o treinamento ou durante a produção têm de parar suas atividades para ajudar os novatos. Estes efeitos são modelados pela influência negativa de *Treinamento* para a Produtividade.

A quantidade de *Tarefas Planejadas* influencia positivamente *Atrasos no Cronograma*, pois se há um aumento na quantidade de tarefas e a produtividade continuar constante acarretará um atraso. O mesmo raciocínio pode ser empregado a *Tarefas Concluídas*, porém considerando uma influência negativa a *Atrasos no Cronograma*. Devido à conclusão de uma tarefa implicar à diminuição para o atraso final do projeto.

Uma decisão de aumentar a equipe geralmente parte do gerente quando este percebe que os prazos para entrega do produto estão se aproximando e ainda existem muitas tarefas a serem concluídas, então, ele tenta aumentar a produtividade incluindo novas pessoas ao projeto. Podemos incluir esta política em nosso modelo se considerarmos que *Atrasos no Cronograma* influencia positivamente em *Pessoal* como no diagrama da Figura 3.3. Ou seja, "mais" atraso pede "mais" pessoal para aumentar a produtividade.

Pela análise desses diagramas, observamos que as setas vão da causa para o efeito e podemos, então, estimar alguns comportamentos do sistema em estudo. Para os casos onde temos influências circulares, ou seja, caminhos fechados onde uma variável pode ser considerada ao

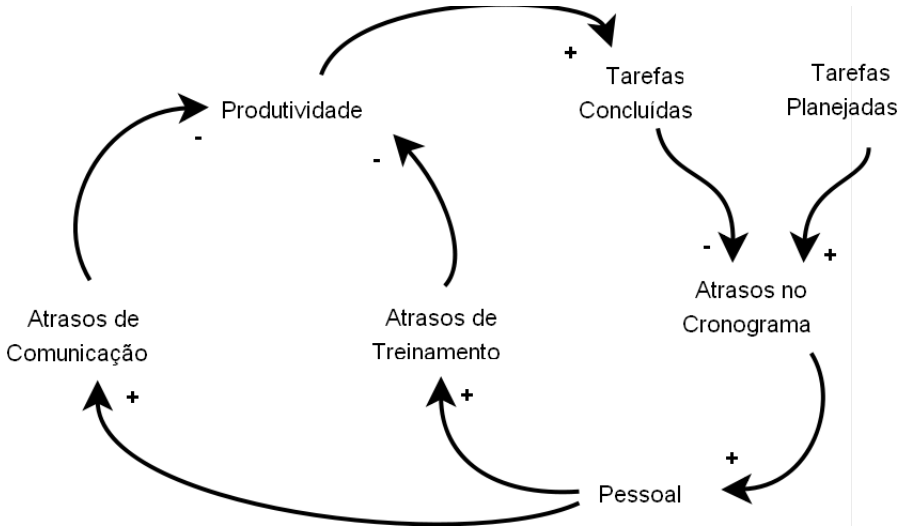

Figura 3.3: Diagrama Causal: Inclusão de tomada de decisão gerencial para aumentar a equipe quando se prevê um atraso no cronograma. Adaptado de Madachy (2008, p.81).

mesmo tempo um ponto de partida e de chegada dizemos que há um laço de realimentação. Os laços fechados indicam que uma alteração em uma variável vai afetar, novamente, a própria variável no futuro.

Os laços de realimentação podem ser considerados como positivos (laços de reforço) ou negativos (laços de atenuação). Em um laço de realimentação positiva, uma pequena mudança em uma variável tende a reforçar ainda mais essa variação. Já em um laço de realimentação negativa, uma mudança em uma variável tende a contrabalancear esta própria mudança, levando o sistema de volta ao seu estado anterior à variação.

Para cada laço fechado podemos fazer uma simples análise para definirmos se o laço em questão é considerado um laço de reforço ou atenuação: para um caminho fechado, contamos quantos sinais "-" existem. Uma quantidade par indica que é uma realimentação positiva, caso contrário, é uma realimentação negativa. A Figura 3.4 ilustra os laços de realimentação em nosso diagrama anterior. Podemos observar que neste modelo aumentar a equipe com base no atraso do cronograma, pode contribuir ainda mais para o atraso do projeto.

Segundo Madachy (2008, p.80), com o advento das ferramentas visuais para desenvolvimento e simulação de Dinâmica de Sistemas, os diagramas causais têm sido mais usados para explicar conceitos baseando-se em um modelo já desenvolvido em diagramas de estoque e fluxo. Isto é devido à crescente facilidade de uso que os ambientes de modelagem e simulação agregam e acabam por agilizar o processo de entendimento do sistema. Entretanto, os diagramas causais ainda superam, em matéria de agilidade, os diagramas de estoque e fluxo para expressar de forma qualitativa os efeitos dinâmicos encontrados.

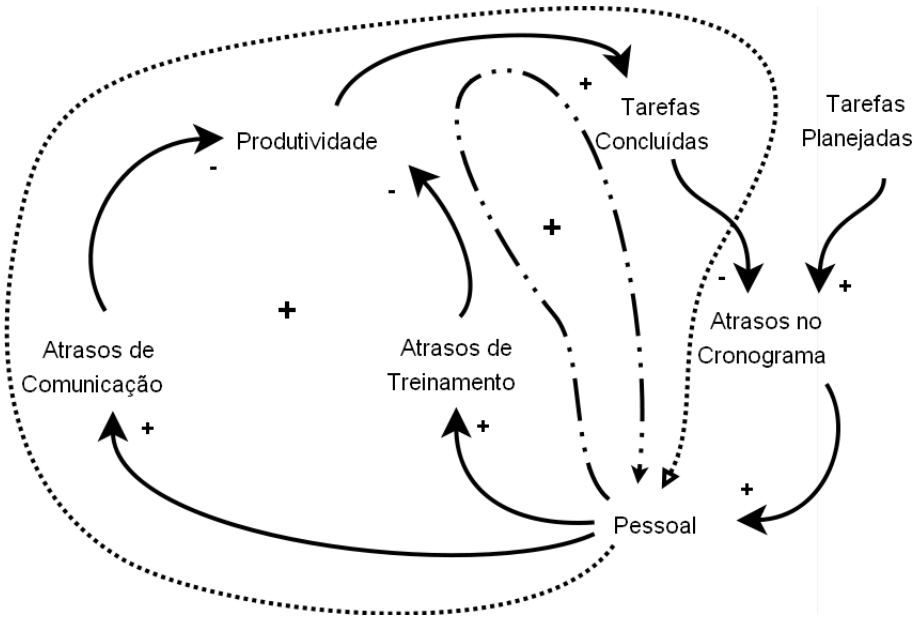

Figura 3.4: Dois laços de realimentação positiva onde uma mudança reforça a própria mudança inicial. Aumentar a equipe com base no atraso do cronograma pode contribuir ainda mais para seu atraso.

Alternativas no sentido de oferecer técnicas de modelagem que unem a rapidez da construção qualitativa, mais conceitual, à simulação imediata para busca de padrões de comportamento, motivaram a criação de ferramentas com suporte aos chamados modelos semiquantitativos. Os ambientes de modelagem e simulação como o *WLinkIT* (SAMPAIO, 1996 apud PEDRO, 2006) e *JLinkIT* (PEDRO, 2006) permitem a construção de modelos semiquantitativos onde os conceitos, mesmo sem um valor aparente, podem ser alterados visualmente na forma de controles deslizantes. As influências entre os conceitos são indicadas como “forte”, “normal” e “fraca” e os comportamentos dinâmicos são observados por gráficos adimensionais e sem indicação de valores. Por motivos de espaço, não incluímos um estudo detalhado dos modelos semiquantitativos neste trabalho. Entretanto, os consideramos uma alternativa viável e útil para captura dos comportamentos de processos.

3.2.2 Diagramas de Estoque e Fluxo

Os diagramas de estoque e fluxo apresentam, detalhadamente, quais são as variáveis, parâmetros e estrutura de um modelo de sistema. Sendo um modelo quantitativo, é necessário registrar as relações numéricas e algébricas por meio de equações através de uma linguagem matemática formal (PEDRO, 2006, p.30).

Uma simulação de um modelo permite acompanhar os valores de seus elementos quantificáveis durante um certo período do tempo. O comportamento do sistema surge devido às

características e interrelações de seus elementos componentes.

3.2.3 Elementos Básicos

O diagramas de estoque e fluxo são compostos por cinco elementos básicos: *estoques finitos*, *estoques infinitos*, *taxas*, *auxiliares* e *informações*. Estes elementos podem ter representações gráficas diferentes ou apresentar variações para alguma especialização dependendo da ferramenta utilizada. Neste trabalho nos concentramos em lidar apenas com esses cinco elementos comuns em todas as ferramentas e utilizaremos a representação gráfica da Figura 3.5.

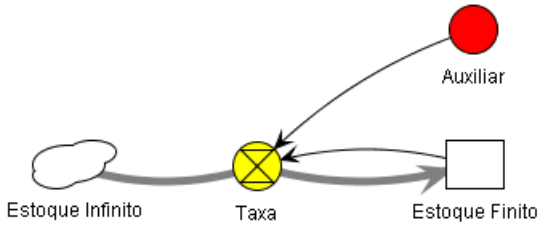

Figura 3.5: Elementos básicos de diagramas de estoque e fluxo: estoque finito, estoque infinito, taxa, auxiliar e informação.

Estoques finitos ou níveis são entendidos como acumulações de quantidades dentro do sistema, são os pontos de onde vêm ou para onde vão as quantidades quando o modelo é simulado e dão a capacidade de "memória" ao sistema. Os estoques finitos são representados graficamente por um retângulo e um nome descritivo. Eles também são responsáveis por trazer as condições iniciais do sistema antes da simulação, portanto, esta informação deve vir anexada ao diagrama junto à lista de equações.

Os estoques finitos têm interpretação intuitiva, quase imediata, quando representam quantidades como pessoas, número de defeitos, tarefas, dias para a entrega de um projeto, etc. Porém, também podem representar acumulações de medidas não físicas como estresse, conhecimento, experiência, felicidade, etc.

Um segundo tipo de estoque são os estoques infinitos, também conhecidos como fontes ou sorvedouros. Representam estoques fora dos limites de interesse do modelo. São considerados, ao mesmo tempo, sempre "cheios", capazes de prover elementos e sempre "vazios", capazes de receber fluxos de entidades. Por estarem fora do escopo do modelo, não possuem valor quantificável ou qualquer equação associada. São representados nos diagramas como um desenho de uma nuvem e um nome descrito e como exemplos podemos citar possíveis usuários de um software, o conjunto de defeitos possíveis que um artefato pode ter, desenvolvedores que saíram

do projeto, etc.

As taxas (ou fluxos) são os elementos que movem as quantidades de um estoque para outro. O valor de um estoque só pode ser alterado quando as entidades quantificáveis se movem para "dentro" ou para "fora" dele, através de uma taxa. Portanto, este elemento representa a taxa de variação de um estoque em relação a um estado de tempo (BARROS, 2001). Como são elas que mudam o estado do sistema durante a simulação, são consideradas as responsáveis pelo comportamento dinâmico do sistema. Podem ser interpretadas, intuitivamente, como válvulas que permitem o fluxo de entidades de um estoque para outro, justificando, inclusive, a sua representação gráfica, lembrando uma válvula, na maioria das ferramentas (como na figura 3.5). Portanto, só podem existir conectando dois estoques por setas (geralmente mais grossas) que indicam o fluxo das entidades.

Uma taxa possui uma expressão associada que irá calcular o seu valor. Essa expressão pode depender dos valores dos estoques, auxiliares ou mesmo outras taxas. Exemplos diretos de taxas relacionadas ao estoques citados acima são: taxa de contratação de pessoal; taxas de desligamento ou transferência de pessoal entre projetos; taxa de produção de defeitos; taxa com que os requisitos funcionais são alterados ou implementados. Há também analogias para as taxas para elementos não físicos como cansaço, descanso, treinamento, etc. O importante é ter em mente que as taxas têm as unidades compatíveis com os estoques que elas alteram por unidade de tempo.

Auxiliares são variáveis ou constantes utilizadas como parâmetros ou para cálculos indiretos (como avaliadores) a partir dos outros elementos do sistema. Um modelo pode ser descrito apenas pelos valores diretos de estoques e taxas, mas o uso de auxiliares o torna mais legível, destacando o real significado de uma expressão específica. Auxiliares geralmente são representadas graficamente por círculos com um nome descritivo. Precisam de uma expressão matemática em função dos outros elementos do diagrama para cálculo de seu valor, da mesma forma que as taxas. Algumas ferramentas destacam constantes como um tipo especial de variável que têm seu valor especificado diretamente na expressão e servem exclusivamente como parâmetros para a configuração do sistema. Como exemplo de auxiliares, podemos representar data limite para entrega de um projeto (em dias ou meses), densidade de erros, valor médio de produtividade, metas de qualidade quantificáveis, etc.

Informações são elementos que indicam, visualmente, o fluxo de dados levados de um elemento a outro, em um modelo de estoque e fluxo. É um artifício gráfico utilizado para mostrar quais elementos fazem parte das equações que descrevem o comportamento do sistema. Esse

fluxo de dados age de forma não conservativa, não é como um fluxo de quantidades representado por taxas. Através das informações, é possível ter uma idéia das relações de causa e efeito e laços de realimentação existentes no modelo. Uma informação é representada por uma seta (geralmente bem fina) que parte de um elemento (que chamaremos de fonte de informação) do diagrama para um outro (consumidor de informação).

Um diagrama de estoque e fluxo é baseado, implicitamente, em um modelo matemático de um sistema de equações diferenciais. Uma simulação de um diagrama de estoque e fluxo é uma resolução numérica de um sistema de equações diferenciais [5] com condições iniciais[6]. Os estoques representam as variáveis do sistema e as taxas suas equações diferenciais. O valor de um estoque, em um determinado momento no tempo, é determinado pela integração das taxas a partir do conhecimento de seu valor inicial, dado pelas condições de contorno.

Podemos escrever a Equação 3.1 que retorna o valor de um estoque finito em qualquer período de tempo `T`.

$$\text{Estoque}(T) = \text{Estoque}(0) + \int_0^T \left(\sum Taxas_{chegando} - \sum Taxas_{saindo}\right) \mathrm{d}t \tag{3.1}$$

3.3 Aplicações de Modelos de Dinâmica de Sistemas em Engenharia de Software

Nas próximas seções apresentamos adaptações dos modelos de processos de software encontrados em Madachy (2008) que servem como exemplos para os diagramas de estoque e fluxo. Os comportamentos dinâmicos simples, apresentados por estes modelos, são tidos como canônicos e são utilizados como base para a construção de modelos complexos.

3.3.1 Crescimento de Usuários

Como exemplo inicial, considere o diagrama causal da Figura 3.6 que descreve o crescimento de usuários de um software inovador, sem concorrentes e sem estagnação de mercado. O conjunto de usuários, por meio da chamada propaganda "boca à boca", contribui para a divulgação do software, aumentando a adesão de novos usuários. Estes novos usuários, por sua

[5] Pelo fato do tempo ser discretizado nos métodos numéricos, as equações passam a ser denominadas como "à diferenças".

[6] Hirsch, Smale e Devaney (1974) conduzem um estudo aprofundado sobre análise das características de sistemas dinâmicos.

vez, vão contribuir ainda mais para a adesão. Um fator de adesão é definido para expressar o quanto as condições de divulgação são favoráveis. A idéia central é que uma população reforça seu próprio crescimento.

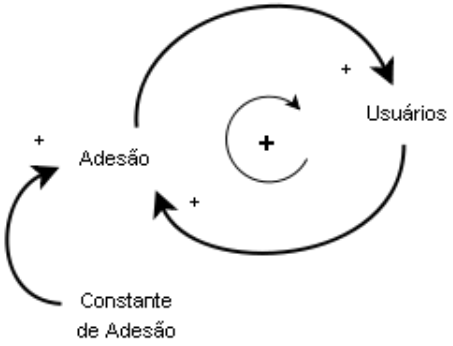

Figura 3.6: Diagrama Causal do crescimento exponencial de usuários de um novo sistema sem competição.

Em diagramas de estoque e fluxo, esse modelo pode ser visto na Figura 3.7 junto com o conjunto de Equações 3.2. Como queremos acompanhar a quantidade de usuários do sistema, definimos um estoque finito *Usuários* com valor inicial (Equação 3.2c). A nuvem *Possíveis Usuários* simbolizando um estoque infinito (não quantificável) representa toda a população que não é usuária do software. O auxiliar *Fator de Adesão* e sua equação associada (3.2b) representam a eficiência com que os usuários divulgam as vantagens entre seus colegas via Internet, análises em revistas ou mesmo no contato individual com possíveis usuários.

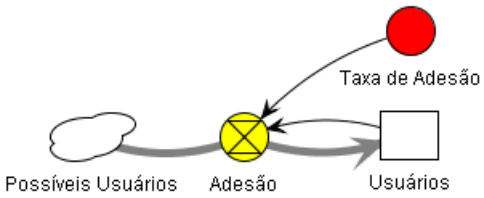

Figura 3.7: Modelo de estoque e fluxo que modela o crescimento exponencial de usuários de um novo sistema sem competição.

$$\text{Adesão} = \text{Fator de Adesão} * \text{Usuários(t)} \tag{3.2a}$$

$$\text{Fator de Adesão} = 0.05 \tag{3.2b}$$

$$\text{Usuários}(0) = 10.0 \tag{3.2c}$$

O símbolo de taxa e as setas largas representam a variação do estoque Usuários. Seu valor

depende da expressão de taxa dada pela Equação 3.2a e o sentido indica a movimentação das quantidades, neste caso, do estoque infinito para o estoque finito.

Os laços de informação, representados pelas setas com linhas finas, são traçados para mostrar como as informações (e não quantidades) são passadas entre os elementos do diagrama.

Nesse diagrama, podemos observar o laço de realimentação formado pelo fluxo de usuários vindo da taxa e pela informação que parte do estoque. Neste caso o efeito é um reforço dependente do sinal do *Fator de Adesão*. Um reforço positivo (dada uma constante positiva) faz a quantidade de usuários aumentar. Um reforço negativo, por sua vez, a faz diminuir.

Devido à simplicidade desse modelo, ele pode ser descrito e resolvido através de sistema de equações diferenciais[7]. O sistema de equações formado pelas Equações 3.3a, 3.3b, 3.3c fazem esta relação com o diagrama de estoque e fluxo. O resultado deste sistema é dado pela Equação 3.3d e descreve o comportamento dinâmico do modelo.

$$\frac{\partial \text{Usuários}}{\partial t} = \text{Constante de Adesão} * \text{Usuários} \tag{3.3a}$$

$$\text{Fator de Adesão} = 0.05 \tag{3.3b}$$

$$\text{Usuários}(0) = 10 \tag{3.3c}$$

$$\text{Usuários}(t) = \text{Usuários}(0) * e^{(\text{Fator de Adesão} * t)} \tag{3.3d}$$

A solução analítica 3.3d é obtida pela integração das taxas, onde cada estoque finito pode ter seu valor encontrado em um determinado momento no tempo. Esta associação com uma solução analítica nem sempre é possível, então, a solução numérica é frequentemente dada como suficiente para a descrição dos modelos. Portanto, não mais apresentaremos a solução analítica dos sistemas de equações diferenciais neste trabalho.

A Figura 3.8 apresenta o resultado da simulação deste modelo. Vemos o comportamento do número de usuários ao longo de um período de tempo de 36 meses. Observamos que o número de usuários cresce, de forma ilimitada, se mantivermos a configuração do sistema.

[7]Uma revisão completa da obtenção de soluções analíticas de equações diferenciais é encontrada em Kreider et al. (1980).

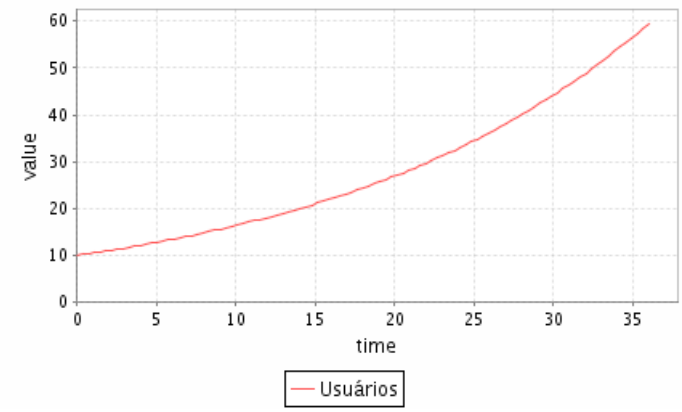

Figura 3.8: Resultado da Simulação do crescimento exponencial de usuários de um novo sistema sem competição. Obtido com o JynacoreSim.

3.3.2 Aprendizado em Equipe

Um segundo bloco básico para construção de modelos mais complexos, envolve dispor estoques em fila, conectados por taxas, de forma que haja movimentação das quantidades de um estoque para outro. As taxas realizam a conversão de unidades de uma quantidade para outra, quando necessário.

As filas podem ser encontradas em diversas partes de um processo de software. Inclusive, todo o processo de software pode ser considerado uma fila que converte requisitos em software. Entretanto, inicialmente vamos lidar com o aspecto de pessoal.

Separamos os membros de uma equipe de desenvolvimento em dois estoques finitos: *Pessoal Novato* e *Pessoal Experiente*. E, da mesma forma que no exemplo anterior, realizamos a conexão dos estoques por uma taxa *Assimilação* e um auxiliar com uma constante *Fator de Assimilação*. O diagrama da Figura 3.9 e as Equações 3.4 descrevem este modelo.

A quantidade de membros novatos irá diminuir, gradativamente, com sua permanência no projeto, assim que assimilarem suas características. Este pessoal é transferido para o grupo de experientes. O valor desta taxa de conversão fica por conta da Equação 3.4a de Assimilação. O auxiliar *Fator de Assimilação* corresponde a 20 dias como o tempo necessário para uma pessoa assimilar o suficiente para ser considerado experiente.

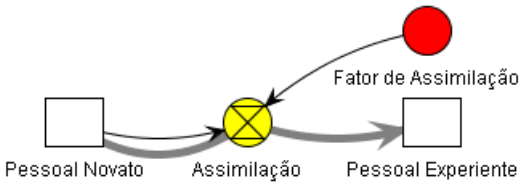

Figura 3.9: Diagrama de Estoque e Fluxo que exemplifica a assimilação do conhecimento em um projeto em equipe.

$$\text{Assimilação} = (\text{Pessoal Novato}) / \text{Fator de Assimilação} \tag{3.4a}$$

$$\text{Fator de Assimilação} = 20.0 \tag{3.4b}$$

$$\text{Pessoal com o Conhecimento}(0) = 1 \tag{3.4c}$$

$$\text{Pessoal sem o Conhecimento}(0) = 24 \tag{3.4d}$$

Definimos uma equipe composta apenas por 1 membro experiente e 24 novatos. O gráfico da Figura 3.10 ilustra o resultado da simulação. Podemos observar a mudança das quantidades dos estoques com o passar dos dias, até que todos os membros da equipe sejam considerados experientes.

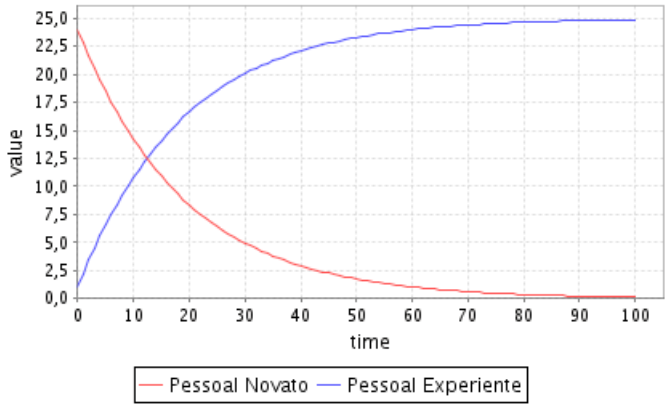

Figura 3.10: Resultado da Simulação da Assimilação de conhecimento pela Equipe. Obtido com o JynacoreSim.

3.3.3 Produção de Software

Um processo de desenvolvimento de software, em sua forma mais básica, consiste em converter *Requisitos de Software*[8] em *Software Desenvolvido*[9]. O diagrama da Figura 3.11, junto com as Equações 3.5, modelam esse cenário assumindo uma equipe com 20 membros fixos com uma produtividade individual constante de 0.1 pontos de função por dia.

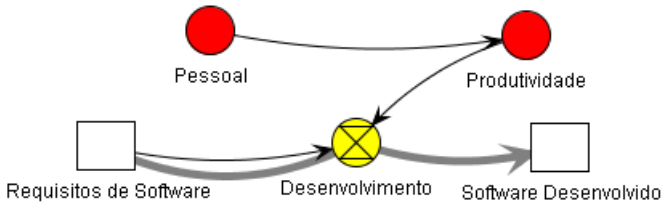

Figura 3.11: Diagrama de Estoque e Fluxo que modela a conversão de Requisitos em Software considerando uma equipe homogênea

$$\text{Desenvolvimento} = \min(\text{Requisitos de Software}, \text{Produtividade}) \tag{3.5a}$$

$$\text{Produtividade} = 0.1 * \text{Pessoal} \tag{3.5b}$$

$$\text{Pessoal} = 10 \tag{3.5c}$$

$$\text{Requisitos de Software}(0) = 150 \tag{3.5d}$$

$$\text{Software Desenvolvido}(0) = 0 \tag{3.5e}$$

Este modelo, quando simulado, apresenta um comportamento linear com o Desenvolvimento a 1 ponto de função por dia e, consequentemente, 150 dias para sua conclusão. A Figura 3.12 apresenta o comportamento do modelo.

Podemos ampliá-lo ao relacionar a experiência do desenvolvedor aos dias que este está ligado ao projeto e levar em conta que desenvolvedores com experiências diferentes possuem produtividades diferentes. Este comportamento pode ser modelado ao associarmos nosso modelo atual com os elementos que usamos na Seção 3.3.2. Com a introdução desses elementos na Figura 3.13 e nas Equações 3.6, o nosso modelo passa a considerar dois níveis de produtividade dos membros da equipe e a sua evolução desde o início do projeto.

[8] As necessidades do usuário final, das metas de qualidade ou padrões formais.
[9] Por software desenvolvido entendemos como os pontos funcionais do sistema final.

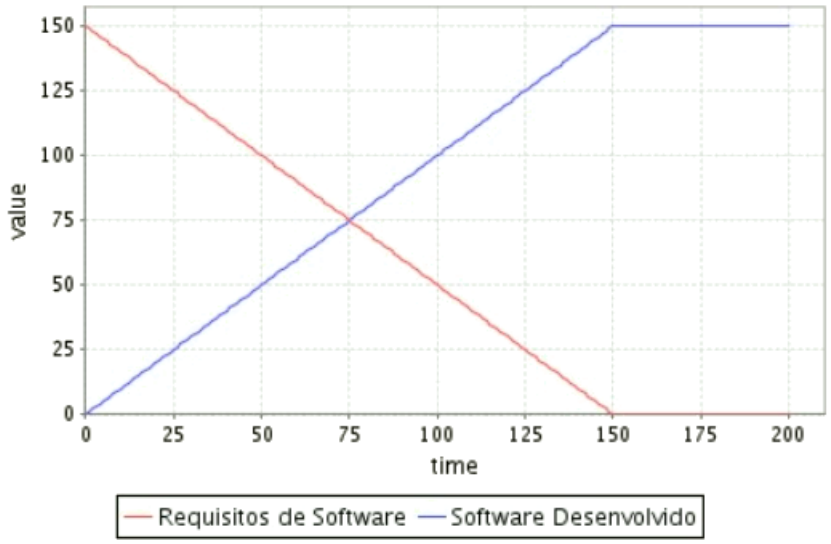

Figura 3.12: Resultado da simulação sem contratação durante o projeto.

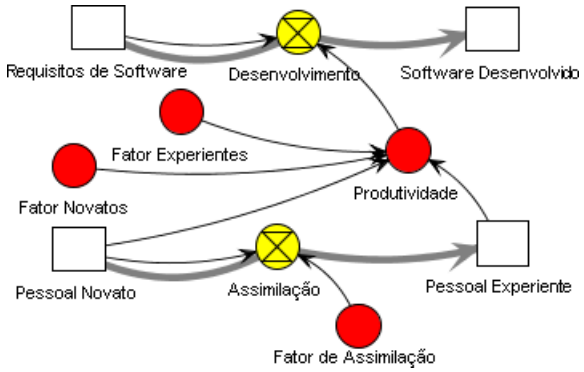

Figura 3.13: Diagrama de estoque e fluxo que modela a diferença de produção entre Desenvolvedores Novatos e Experientes.

$$\text{Assimilação} = \text{Pessoal Novato}/\text{Fator de Assimilação} \tag{3.6a}$$

$$\text{Fator de Assimilação} = 20.0 \tag{3.6b}$$

$$\text{Pessoal Novato}(0) = 5 \tag{3.6c}$$

$$\text{Pessoal Experiente}(0) = 5 \tag{3.6d}$$

$$\text{Fator Novatos} = 0.8 \tag{3.6e}$$

$$\text{Fator Experientes} = 1.2 \tag{3.6f}$$

$$\begin{aligned}\text{Produtividade} = 0.1 * (&\text{Fator Novatos} * \text{Pessoal Novato} \\ &+ \text{Fator Experientes} * \text{Pessoal Experiente})\end{aligned} \tag{3.6g}$$

Podemos continuar a incrementar o modelo para incluir os efeitos da chegada de novos

integrantes a um projeto já em andamento. O diagrama da Figura 3.14 e as Equações 3.7 incluem elementos extras para este fim.

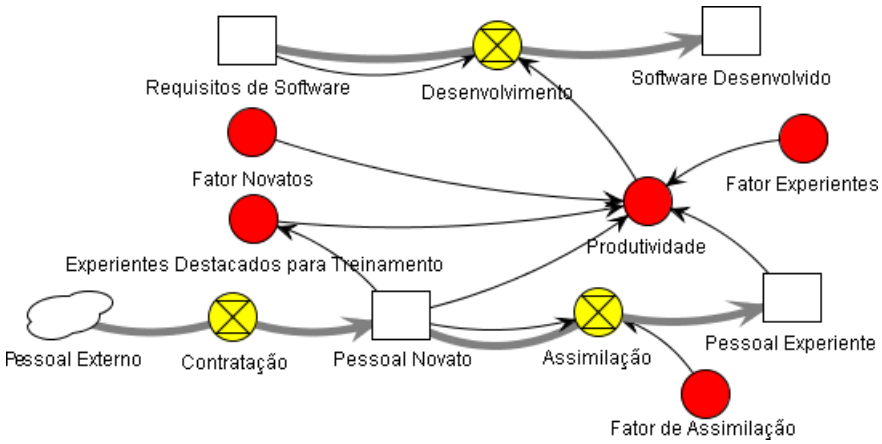

Figura 3.14: Diagrama de estoque e fluxo que modela a conversão de requisitos em software.

$$\text{Contratação} = \text{IF}(\text{\_TIME\_} = 50, 5.0/\text{\_TIME\_STEP\_}, 0.0) \tag{3.7a}$$

$$\text{Experientes Destacados para Treinamento} = \text{Pessoal Novato}/3 \tag{3.7b}$$

$$\begin{aligned}\text{Produtividade} = 0.1 * (&\text{Fator Novatos} * \text{Pessoal Novato} \\ &+ \text{Fator Experientes} * (\text{Pessoal Experiente} \\ &- \text{Experientes Destacados para Treinamento}))\end{aligned} \tag{3.7c}$$

Se quisermos modelar o caso em que um grupo de desenvolvedores integra a equipe quando o projeto já teve início, adicionamos mais uma taxa *Contratação* que insere novas pessoas apenas no tempo indicado com a Equação 3.7a. Conforme vemos na Figura 3.15 esta equação insere um 5 desenvolvedores em um determinado momemto do projeto na forma de um impulso.

Outro efeito que inserimos ao modelo é que um membro da equipe experiente passa a dedicar parte do seu tempo para treinamento de até três novatos. Para tanto, alteramos a Produtividade como mostrado na Equação 3.7c. Assim que os desenvolvedores vão ganhando experiência, a atenção destes experientes vai sendo liberada para trabalhar novamente no projeto. A figura 3.16 apresenta a simulação dos experientes destacados para desenvolvimento.

A Figura 3.17 apresenta o resultado da simulação dos efeitos da chegada de novos desenvolvedores sobre o desenvolvimento do projeto. Há uma queda no desenvolvimento que só é recuperada com o ganho de experiência e diminuição da necessidade de treinamento. Portanto, os ganhos por acrescentar pessoal só são obitidos após um período de tempo.

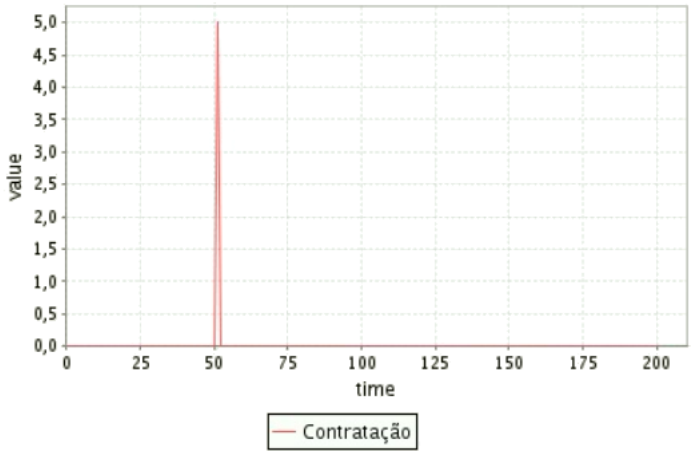

Figura 3.15: Simulação da contratação de novos desenvolvedores na forma de um impulso.

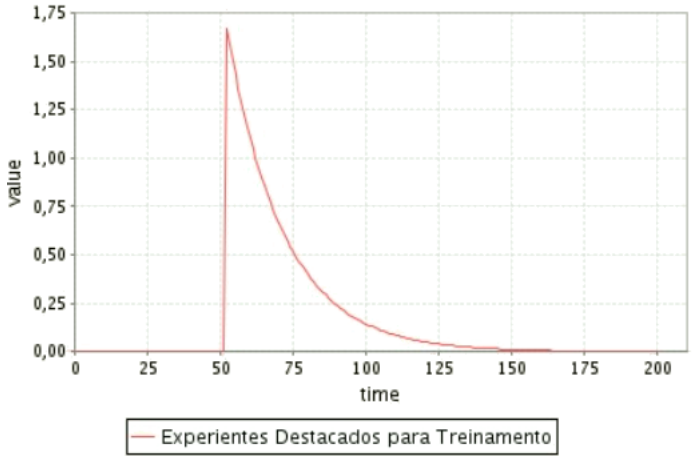

Figura 3.16: Simulação de Experientes Destacados para Treinamento.

A decisão de quando adicionar novos membros a um projeto deve ser estudada cuidadosamente por causa deste efeito. A Figura 3.18 apresenta a simulação para 5 desenvolvedores que são adicionados ao projeto no 50º dia e temos uma redução considerável no tempo do projeto comparado ao sem acrécimo de pessoal da Figura 3.14.

Já na Figura 3.19 apresenta o caso onde os mesmos 5 desenvolvedores são adicionados ao projeto no 135º dia. O resultado da simulação da Figura 3.20 Não há tempo hábil para recuperar a perda de tempo com o treinamento e o projeto acaba por ter a mesma duração que teria sem acréscimo de pessoal.

3.3.4 Flutuação de Pessoal

Os modelos Dinâmicos que envolvem mais de um estoque finito e laços de realimentação podem apresentar um comportamento oscilatório dependendo de sua estrutura. O seguinte

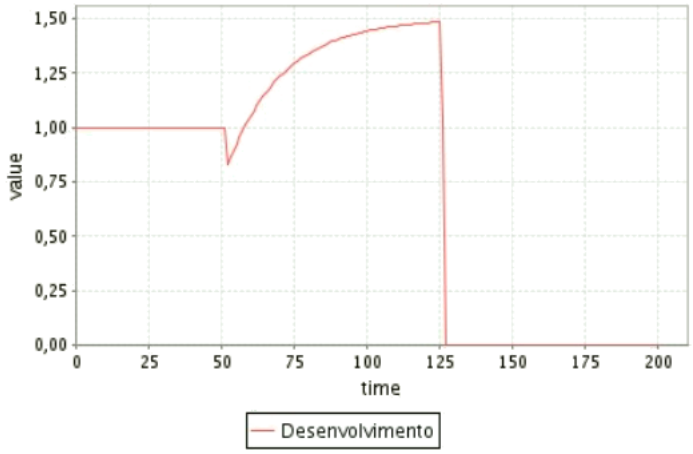

Figura 3.17: Diagrama de estoque e fluxo que modela a conversão de requisitos em software.

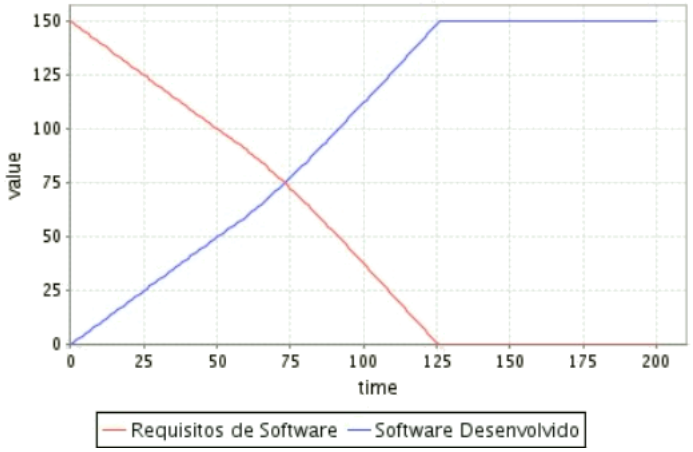

Figura 3.18: Simulação dos Requisitos e Software Desenvolvido com contratação logo no início do projeto.

modelo representa uma organização que possui um conjunto de softwares em seu catálogo formando seu *portfolio*. Assumindo que os softwares possuem uma vida útil, estes são descartados a uma determinada taxa. Com o portfolio abaixo de um certo limiar, a gerência da empresa decide desenvolver software novo, para substituir o que foi descartado. Para tanto, contratam novos funcionários que irão participar do desenvolvimento dos novos softwares. Esse modelo apresenta o comportamento oscilatório devido à demanda e consumo estarem fora de fase.

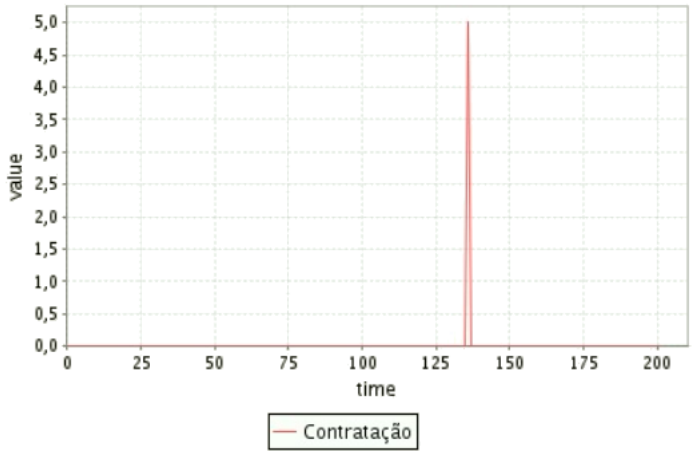

Figura 3.19: Simulação da Contratação próxima ao fim do projeto.

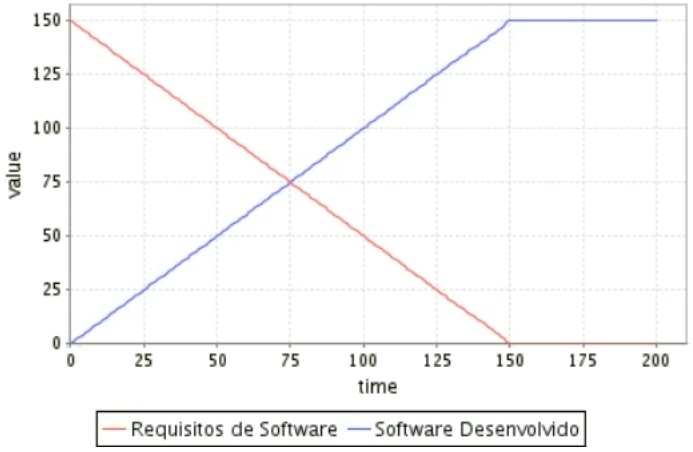

Figura 3.20: Simulação dos Requisitos e Software Desenvolvido com contratação próximo ao fim do projeto.

$$\text{Produção de Software} = (\text{Produtividade} * \text{Pessoal}) \tag{3.8a}$$

$$\text{Contratação} = \text{Pessoal necessário}/\text{Atraso de Contratação} \tag{3.8b}$$

$$\text{Produção necessária} = \text{Diferença}/\text{Tempo para Eliminar a Diferença} \tag{3.8c}$$

$$\text{Pessoal a Contratar} = \text{Produtividade necessária}/\text{Produtividade} \tag{3.8d}$$

$$\text{Eliminação de Software} = 100.0 \tag{3.8e}$$

$$\text{Atraso de Contratação} = 1.0 \tag{3.8f}$$

$$\text{Tempo para Eliminar a Diferença} = 0.5 \tag{3.8g}$$

$$\text{Portfolio Desejado} = 2000 \tag{3.8h}$$

$$\text{Pessoal}(0) = 400 \tag{3.8i}$$

$$\text{Portfolio}(0) = 2500 \tag{3.8j}$$

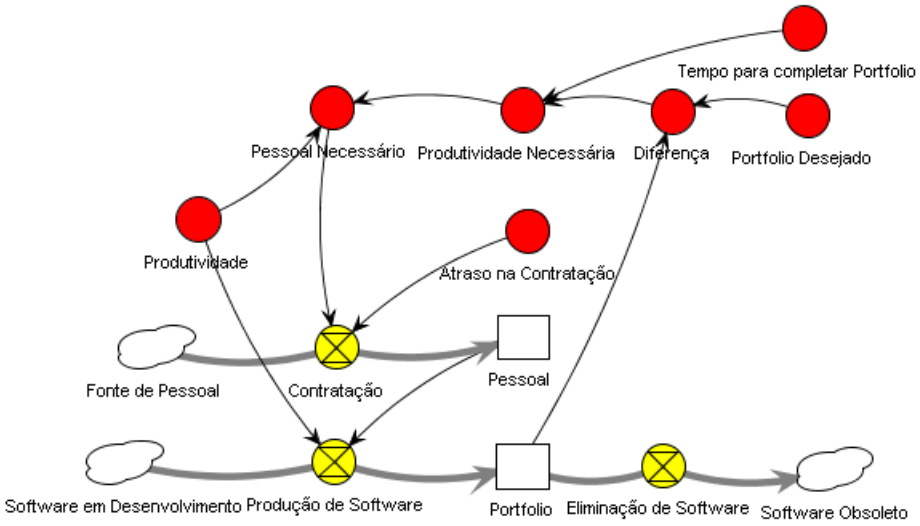

Figura 3.21: Diagrama de estoque e fluxo que exemplifica a oscilação de uma equipe em função da demanda.

Na Figura 3.22 observamos o resultado da simulação. Observamos que o tamanho da equipe aumenta quando o portfolio começa cair abaixo do limite desejado. O oposto também ocorre, quando há mais software no portfolio do que o desejado, os desenvolvedores são dispensados e há a queda por descarte. Como a demanda por desenvolvedores e a quantidade de software no portfolio não estão sincronizadas, há a oscilação.

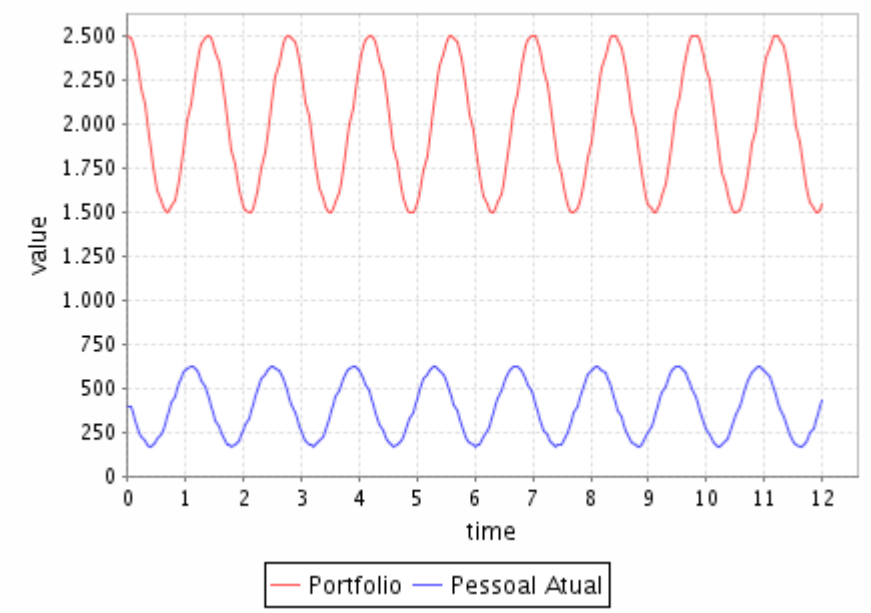

Figura 3.22: Resultado da simulação do diagrama com oscilação de uma equipe em função da demanda de manter um portfolio de software.

3.4 Conclusões Parciais

Os diagramas da Dinâmica de Sistemas permitem capturar o comportamento dinâmico de um sistema trocando uma abordagem relacional de entrada e saída por uma visão de causa e efeito. Assim, o modelo possui um comportamento endógeno que emerge de sua estrutura.

Esta abordagem é particularmente útil para criação de modelos onde há um conhecimento ou teoria sobre os comportamentos das partes componentes do sistema: cada parte pode ser modelada individualmente e depois, as partes são combinadas na esperança de que o conjunto de elementos vá apresentar o comportamento geral do sistema.

A Dinâmica de Sistemas possui dois diagramas para construção de modelos: os diagramas causais e diagramas de estoque e fluxo. Os diagramas causais capturam de forma qualitativa as relações existentes entre os conceitos do modelo e servem para uma estimativa rápida de comportamento. Os diagramas de estoque e fluxo capturam os comportamentos de forma quantitativa, através da descrição detalhada da natureza de cada conceito por meio de equações matemáticas. Os diagramas de estoque e fluxo de Dinâmica de Sistemas podem ser simulados numericamente com a ajuda de um computador.

Os modelos de Dinâmica de Sistemas tendem a crescer em complexidade quando um grande número de detalhes é modelado. Cada ferramenta de modelagem trata este problema de forma independente. O uso de linguagens baseadas em Dinâmica de Sistemas se torna uma alternativa para evitar vincular os modelos desenvolvidos às soluções propostas por ferramentas específicas. No Capítulo 4 abordamos, em detalhes, os fundamentos dos metamodelos de Dinâmica de Sistemas como linguagem para construção de modelos de processos de desenvolvimento de software.

4 *Introdução à Modelagem de Processos em Metamodelos de Dinâmica de Sistemas*

Um modelo, por sua própria definição, não pode representar todos os detalhes de um sistema real. Ele foca nos aspectos mais essenciais para responder às questões propostas, em uma linguagem que seu usuário compreenda. Entretanto, um mesmo modelo pode ser usado por um público composto por pessoas com vários interesses no modelo. Então, fica complicado definir o que é realmente importante e, muitas vezes, vários modelos simplificados são construídos a partir de um modelo mais geral.

Neste capítulo, apresentamos os fundamentos de uma extensão da linguagem de Dinâmica de Sistemas, chamada de Metamodelos de Dinâmica de Sistemas, proposta por Barros (2001, p.71). O uso da linguagem estendida permite que modelos sejam construídos em dois níveis de abstração: um menos abstrato, com poucos elementos extras e bem próximo dos diagramas de estoque e fluxo e um mais abstrato, com elementos mais próximos do domínio modelado.

O uso da linguagem estendida de Dinâmica de Sistemas é uma alternativa elegante para se lidar com o crescimento da complexidade dos modelos de Dinâmica de Sistema quando usados para modelar estruturas que possuem o mesmo comportamento. Outro ponto de interesse é que a linguagem permite capturar formalmente alterações estruturais nestes elementos de forma sistemática, permitindo explorar cenários que podem influenciar no comportamento do modelo.

Este capítulo é composto por três seções: a Seção 4.1 expõe os problemas dos modelos tradicionais e os motivos para se utilizar uma linguagem baseada Dinâmica de Sistemas; a Seção 4.2 apresenta os conceitos do metamodelos de Dinâmica e Sistemas e a Seção 4.3 apresenta nossas conclusões. A Seção 4.2 é dividida nas Subseções 4.2.1,4.2.2 e 4.2.3, que lidam, respectivamente, com os modelos de domínio, instância e cenário da linguagem estendida.

4.1 Linguagens baseadas em Dinâmica de Sistemas

O conjunto limitado de elementos dos diagramas de estoque e fluxo de Dinâmica de Sistemas torna simples a sua aprendizagem, permitindo que sistemas dinâmicos sejam modelados rapidamente. Estes modelos, em conjunto com um computador, permitem simular a média dos elementos de um sistema, como: o esforço de uma equipe, custos de atividades, quantidade de erros gerados ou o tempo em que o projeto é concluído. Apesar da representação destas quantidades levarem em conta valores médios ou a soma total, nada impede que representemos elementos complexos em detalhes. Por exemplo, o modelo da Figura 4.1, tenta representar a exaustão de um único desenvolvedor durante um período de trabalho com hora extra.

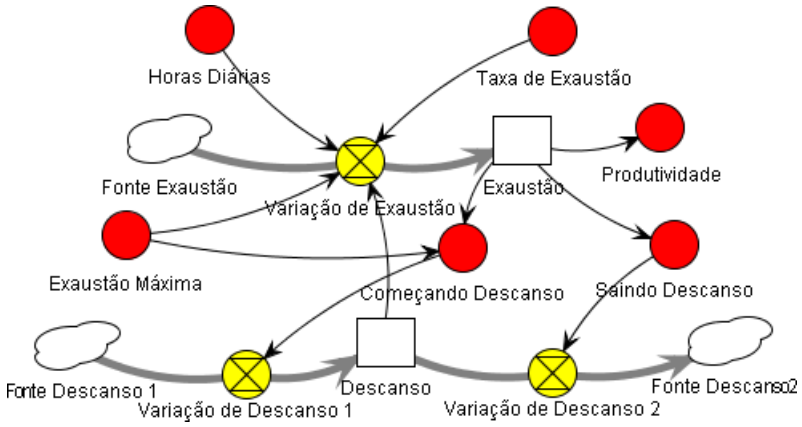

Figura 4.1: Diagrama de estoque e fluxo que modela a Exaustão de um desenvolvedor. Adaptado de Barros (2001, p.181).

Se quisermos modelar um segundo desenvolvedor da equipe, sob as mesmas condições, será necessário utilizar a mesma estrutura, geralmente incluindo um prefixo para simbolizar a qual indivíduo os elementos estão associados. No diagrama da Figura 4.2 consideramos os elementos do primeiro desenvolvedor com o prefixo “D1.” e os elementos do segundo com “D2.”.

A complexidade aumenta para cada desenvolvedor que for adicionado ao modelo, pois esta estrutura deve ser repetida. E, além desse aumento da complexidade visual, a replicação de estruturas torna-se particularmente inconveniente quando o modelo é modificado, onde cada conjunto de elementos deve ser alterado para reproduzir a nova configuração.

Recursos como agrupamento dos elementos de Dinâmica de Sistemas em blocos funcionais reutilizáveis ficam restritos às implementações individuais das ferramentas mais populares. Portanto, faz-se necessário inserir mais um nível de abstração durante o processo de construção, apresentação e uso dos modelos. A equipe responsável pela ferramenta PowerSim (POWER-

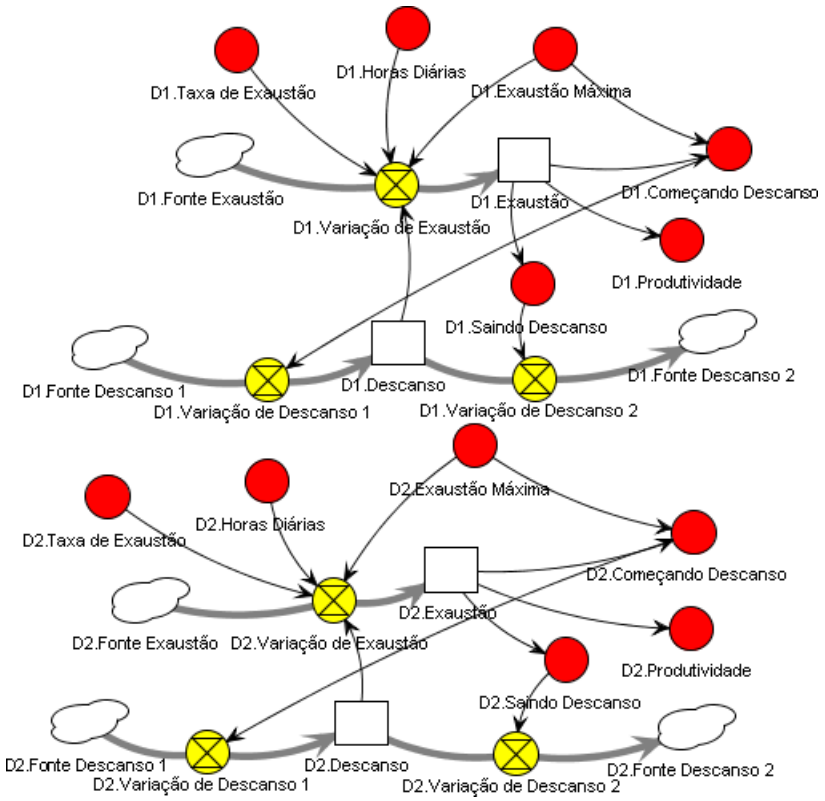

Figura 4.2: Modelo de estoque e fluxo que representa a produtividade de uma equipe de desenvolvedores trabalhando por regime de hora extra.

SIM SOFTWARE, 2009), propôs uma extensão orientada a objetos buscando resolver esses problemas dos diagramas de estoque e fluxo (MYRTVEIT, 2000 apud BARROS, 2001, p.89). Porém, segundo Barros (2001, p.90), a solução agrega muitos elementos extras que dificultam o aprendizado da linguagem.

Os *metamodelos de Dinâmica de Sistemas* compõem a linguagem estendida proposta por Barros (2001, p.71), incluem poucos elementos extras aos diagramas de estoque e fluxo da linguagem de Dinâmica de Sistemas. Estes metamodelos são utilizados para gerar modelos em diagramas de estoque e fluxo tradicionais, através de um compilador de metamodelos.

Esta linguagem permite controlar a complexidade crescente de um modelo, aumentando o nível de abstração dos seus construtores. Um maior nível de abstração é atingido quando são criados construtores que vão representar conceitos mais próximos do mundo real, domínio do problema, sem ter contato com as equações que descrevem seu comportamento dinâmico.

O processo de modelagem em questão possui duas etapas distintas: a primeira, chamada de *modelagem de domínio*, consiste na definição das classes que compõem o domínio, seu comportamento dinâmico e as possíveis relações entre as instâncias destas classes; a segunda, chamada de *modelagem de instância*, consiste na criação de instâncias das classes definidas no

modelo de domínio, definição dos valores de suas propriedades individuais e relacionamentos entre as instâncias. A Figura 4.3 apresenta essa separação de modelos.

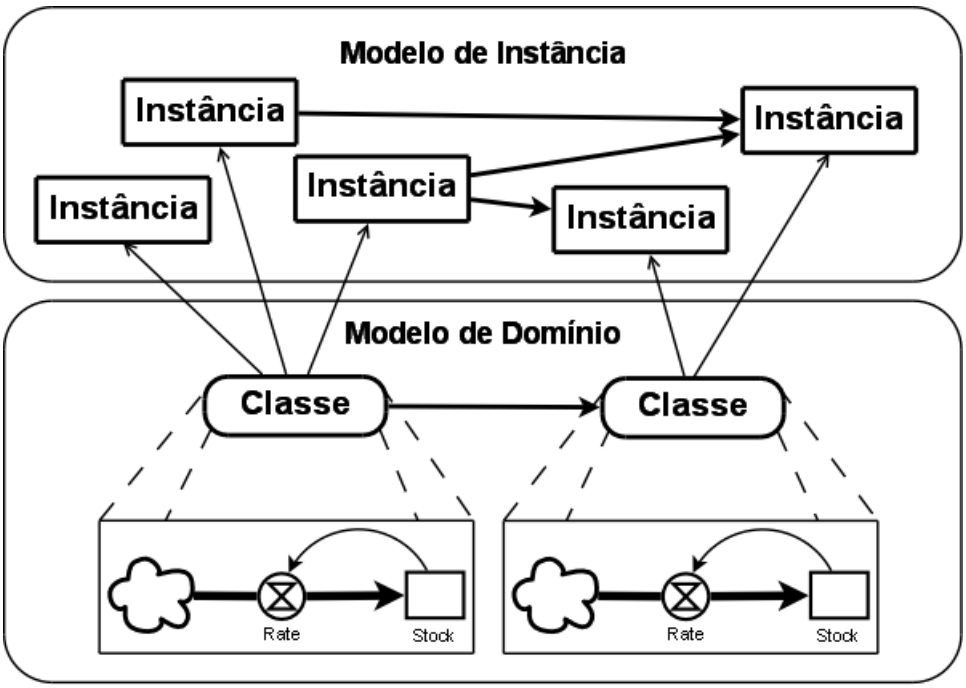

Figura 4.3: Arquitetura da linguagem estendida de Dinâmica de Sistemas: modelo de domínio com classes descritas em Dinâmica de Sistema e modelo de instância descrito com estas classes.

Com essas duas etapas bem definidas e separadas, a modelagem separa os conhecimentos necessários para se interagir com os modelos em dois níveis de abstração.

4.2 Metamodelos de Dinâmica de Sistemas

O metamodelo é uma representação em alto nível de abstração dos construtores da Dinâmica de Sistemas, que permite a definição de linguagens de modelagem específicas para determinados domínios (BARROS, 2001). A definição do domínio se dá pela construção de um *modelo de domínio* e *modelos de cenários*. A partir dos construtores definidos nesses dois modelos, um sistema é representado por um *modelo de instância*. A seguir, expomos a construção de cada tipo de modelo.

4.2.1 Modelo de Domínio

Um modelo de domínio tenta capturar as propriedades, comportamentos e relações de um conjunto de elementos que descrevem um domínio de conhecimento. É uma descrição genérica que pode ser especializada para um problema específico. Por ser apenas uma descrição, não

podemos realizar simulações com modelos de domínio. A construção deste modelo exige um maior conhecimento do sistema e das técnicas de modelagem com diagramas de estoque e fluxo.

O conceito central do modelo de domínio é a *classe*. Uma classe é um conjunto de elementos que podem ser descritos pelas mesmas propriedades e exibem comportamento similar. A classe pode ser entendida como um projeto, uma descrição detalhada com a qual pode-se criar uma *instância de classe* (visto em detalhe na Seção 4.2.2). Podemos fazer uma analogia do funcionamento de uma classe com um projeto estrutural de uma casa: este descreve detalhadamente como construir a edificação, mas não é uma construção real e, portanto, não podemos interagir com ela.

A classe pode conter elementos conhecidos como propriedades. Uma *propriedade* guarda um dado relevante que as instâncias de classe vão ter. Ela possui um valor padrão definido, mas este pode ser alterado individualmente em cada instância para refletir suas características próprias.

Os elementos de classe, chamados comportamentos, são responsáveis pela descrição da dinâmica de suas instâncias. Um *comportamento* é especializado por um dos construtores dos diagramas de estoque e fluxo, definidos por estoque, taxa, auxiliar e tabela.

O *estoque* é equivalente ao estoque finito dos diagramas de estoque e fluxo. É um acumulador de quantidades que necessita de uma taxa para mudar seu valor inicial. De forma diferente dos modelos de estoque e fluxo tradicionais, na linguagem estendida não há a necessidade representar os estoques infinitos. Uma taxa que não possua uma das duas conexões assume que o lado ausente age como um estoque infinito.

A *taxa* é o deslocamento das quantidades presentes nos estoques. Portanto, ela possui um sentido para o deslocamento de quantidades e pode conectar até dois estoques.

Um *auxiliar* é um cálculo indireto a partir das propriedades, estoques, taxas ou outros auxiliares. É utilizado para isolar indicadores ou informações importantes. Uma *tabela* guarda uma lista de valores e os associa a um intervalo contínuo, agindo como um auxiliar que retorna valores discretos.

Para tornar mais claro o uso da linguagem estendida, vamos criar um modelo simplificado de um processo de desenvolvimento de software, adaptando os exemplos descritos em Barros (2001) para capturar a interdependência de atividades de um processo. Vamos criar classes para os elementos tidos como básicos de um processo de desenvolvimento de software: *Desenvolvedor*, *Atividade* e *Artefato* e iremos relacioná-las umas às outras para descrição da estrutura geral

de um processo em particular.

Um desenvolvedor será modelado como uma classe contendo apenas uma propriedade *experiência* e um auxiliar *Produtividade*. Inicialmente, vamos assumir que a sua Produtividade não depende de sua experiência e é constante, como indicam as Equações associadas 4.1.

Figura 4.4: Classe Desenvolvedor de um Modelo de Domínio

$$\text{experiência} = 1.0 \tag{4.1a}$$

$$\text{Produtividade} = 1.0 \tag{4.1b}$$

A Figura 4.5 e as Equações 4.2 apresentam o modelo de uma classe Atividade. Criamos uma propriedade *duração*, para indicar a duração da atividade; um estoque finito *Tempo para Concluir*, para registrar o tempo restante para a conclusão; um auxiliar *Produção* que é o valor com o qual a atividade é realizada e uma taxa *Trabalho* que, com base na Produção, altera o valor de Tempo para Concluir.

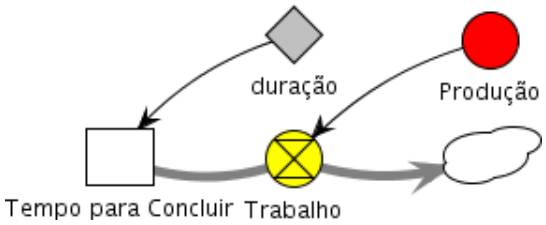

Figura 4.5: Classe Atividade de um modelo de domínio.

$$\text{Produção} = 1.0 \tag{4.2a}$$

$$\text{Trabalho} = MIN(\text{Tempo para Concluir}, \text{Produção})) \tag{4.2b}$$

$$\text{duração} = 25.0 \tag{4.2c}$$

$$\text{Tempo para Concluir}(0) = \text{duração} \tag{4.2d}$$

Já a classe Artefato será modelada apenas contendo um estoque finito *Erros* para registrar a quantidade de erros que possui.

Figura 4.6: Classe Artefato de um modelo de domínio.

$$\text{Erros}(0) = 0.0 \tag{4.3a}$$

Relacionamentos

Um relacionamento define a possibilidade que as instâncias das classes têm de criar uma ligação estrutural com outras instâncias (de mesma classe[1] ou não). Os relacionamentos são orientados: a instância de classe alvo assume um *papel* para a instância de classe relacionada. O relacionamento passa a ser bidirecional se também for definido um papel para a instância de origem. A multiplicidade dever ser especificada de forma que o relacionamento será *simples* se puder existir apenas uma ligação para a instância alvo (um para um) ou será *múltiplo* caso possam haver ligações para várias instâncias alvo (um para muitos).

Neste nosso modelo a classe *Atividade* agirá como origem principal dos quatro relacionamentos existentes. O primeiro, chamado *Equipe*, relaciona-se com Desenvolvedor, de forma que a Atividade tenha conhecimento de quem está responsável por ela. Dois relacionamentos com *Artefato*, um chamado *Entrada* e o outro *Saída*, representam os artefatos consumidos e gerados durante a execução da atividade. Um autorelacionamento múltiplo chamado *Precedente* representa a interdependência das atividades, permitindo relacioná-las da mesma forma que nos diagramas PERT (WIEST; LEVY, 1977 apud BARROS, 2001). A Figura 4.7 mostra o modelo de domínio com as classes e relacionamentos.

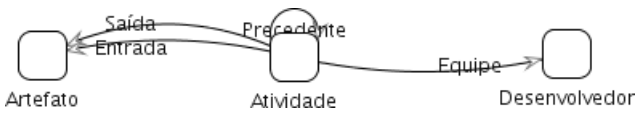

Figura 4.7: Modelo de domínio a nível de classes e relacionamentos.

Assim que as classes e relacionamentos são definidos no modelo de domínio, podemos partir para a criação de instâncias das classes e definir relacionamentos entre elas em um modelo de instância.

[1] Um relacionamento entre instâncias de mesma classes é conhecido como *autorelacionamento*

4.2.2 Modelo de Instância

Sistemas reais são modelados a partir de modelos de instância. Um modelo de instância é composto pela construção de instâncias a partir da descrição de classes em um modelo de domínio.

Cada instância de classe possui um identificador único e suas propriedades que diferem do padrão da classe. As instâncias incluem também os relacionamentos que compõem a estrutura do modelo.

A modelagem de instância possui um maior grau de abstração, não exibindo os elementos de Dinâmica de Sistemas e suas equações durante a criação dos modelos. Isto permite que um usuário com conhecimento restrito ao domínio consiga interagir com o modelo e simulações deste.

Voltando ao exemplo que construímos como modelo de domínio na Seção 4.2.1, podemos criar um modelo de instância onde um projeto é descrito por duas atividades, *Projeto* e *Codificação*, e dois desenvolvedores, *D1* e *D2*, respectivamente responsáveis por cada atividade. Três instâncias de Artefato serão criadas para representar os artefatos consumidos e gerados durante as atividades: *Modelo de Análise*, *Modelo de Projeto* e *Código Fonte*.

A atividade Projeto usa como entrada o artefato Modelo de Análise e produz como saída Modelo de Projeto. O artefato Modelo de Projeto, por sua vez serve como entrada para a atividade Codificação, que gera um artefato *Código Fonte* como saída.

A relação entre atividades é feita de forma que a atividade Projeto é descrita como precedente de Codificação, assumindo que ela só terá início quando o projeto estiver completo. O diagrama do modelo de instância está exposto na Figura 4.8.

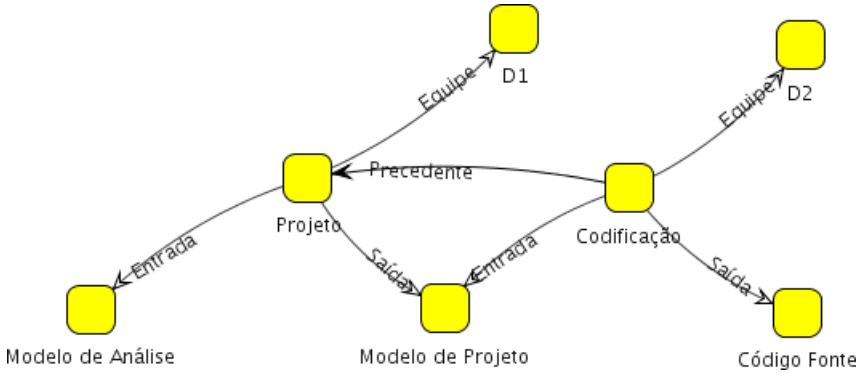

Figura 4.8: Modelo de instância a nível de classes e relacionamentos

Vale a pena frisar que uma instância é o resultado da criação dos elementos contidos em uma

classe. Após construída (ou instanciada) a instância de classe passa a ter seus próprios estados e não possui quaisquer relações com os das outras instâncias. Podemos recorrer novamente à analogia com projeto estrutural: duas casas podem ser construídas (instanciadas) utilizando o mesmo projeto, porém, se você faz alterações estruturais em uma casa, não tem efeito nenhum na casa ao lado. Entretanto, se você fizer alterações no projeto, as próximas casas a serem construídas terão as alterações, mas as que já foram construídas permanecerão iguais.

Os nossos modelos de domínio e instância criados como exemplos são descritivos quanto à estrutura do projeto, mas não são completamente funcionais. Podemos simulá-los da forma como estão descritos e seu comportamento irá mostrar que as atividades são executadas paralelamente, independente da estrutura de precedência e equipe. Isto ocorre porque as equações não implementam a dependência de atividades, a influência da experiência na produtividade dos desenvolvedores e nem a geração de erros.

Estas características serão implementadas separadamente dos modelos básicos para ilustrar uma outra construção presente na linguagem estendida, os *modelos de cenários* descritos mais detalhadamente na Seção 4.2.3, a seguir.

4.2.3 Modelos de Cenários

O terceiro tipo de modelo da linguagem estendida é conhecido como *modelo de cenário*. Definiremos um cenário como um elemento reutilizável que descreve modificações estruturais, que alteram o comportamento das instâncias de classes as quais forem conectadas.

No âmbito de gerenciamento de processos de software, Barros (2001, p.91) define um cenário como uma representação de eventos, políticas, procedimentos, ações e estratégias gerenciais que não podem ser considerados parte de um projeto de desenvolvimento de software, mas práticas impostas ou aplicadas sobre o projeto ou situações excepcionais que o gerente pode encontrar ao longo do projeto.

O autor destaca ainda que os cenários podem ser organizados em uma base de conhecimento centralizada da organização, permitindo a documentação das assertivas e informações conhecidas sobre os elementos de projeto. No âmbito educacional, por exemplo, participantes da atividade podem experimentar a combinação de diversas políticas de gerenciamento, teorias e eventos. Assim, é possível analisar os impactos de estratégias de gerenciamento através do uso de modelos simulados.

Um modelo de cenário é criado para um modelo de domínio específico. Ele é descrito no

mesmo nível de abstração que as classes, mas é usado junto aos modelos de instâncias. Os modelos de cenários listam um conjunto de conexões que os relacionam com as classes de domínio. Uma conexão de cenário, por sua vez, é descrita por propriedades, comportamentos e ajustes. As propriedades e comportamentos são descritos da mesma forma que no modelo de classe e são inseridos na instância durante sua ligação. Os ajustes afetam as expressões previamente definidas nos elementos. São usadas para adicionar referências aos novos elementos ou para alterar completamente a expressão.

Vamos voltar aos modelos que começamos a construir nas Seções 4.2.1 e 4.2.2 para, via cenários, adicionar os comportamentos que faltaram. Primeiro vamos definir um cenário *Produtividade Baseada na Experiência* que consiste em alterar nossa classe Desenvolvedor, de forma que sua propriedade experiência passe a influenciar na sua Produtividade. Este cenário exige apenas um ajuste na Equação 4.1b da Produtividade. Vamos utilizar a Equação 4.4 proposta em Barros (2001, Apêndice C, p.202) que inclui o efeito. O diagrama de uma instância de classe Desenvolvedor, com o cenário conectado, fica como mostrado na Figura 4.9.

$$\text{Produtividade} = 0.667 + 0.666 * \text{experiência} \tag{4.4}$$

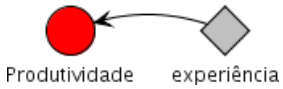

Figura 4.9: Instância de classe com cenário “Produtividade por Experiência” aplicado.

Vamos definir um outro cenário, *Produção Baseada na Equipe* que ao ser conectado à Atividade altera a equação de Produção. A idéia é levar em conta o efeito Produtividade do Desenvolvedor, através do relacionamento Equipe. Como é um elemento externo à classe, ele é identificado pelo nome do relacionamento, ou seja, *Equipe.Produção*. A conexão do cenário altera a Produção para ficar na forma da Equação 4.5. Este elemento referenciado via relacionamento aparece como um Auxiliar com linha tracejada no diagrama da Figura 4.10.

$$\text{Produção} = \text{Equipe.Produtividade} \tag{4.5}$$

As conexões de cenários podem incluir outros elementos além de apenas alterar as equações. Por exemplo, um cenário *Geração de Erros Baseada na Produção* tenta modelar a produção de erros com base no Trabalho de uma Atividade. Ele acrescenta uma taxa *Geração de Erros* à Atividade. Essa taxa cria erros no artefato de saída ligado à instância pelo relacionamento Saída. A Equação 4.6 e o diagrama da Figura 4.11 apresentam a estrutura de uma

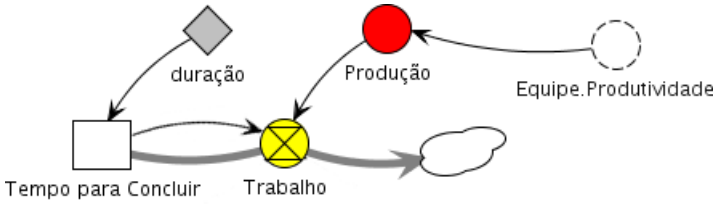

Figura 4.10: Instância de classe Atividade com Cenário “Produtividade por Experiência” aplicado

instância de Atividade com o cenário conectado.

$$\text{Geração de Erros} = \text{Trabalho} \tag{4.6}$$

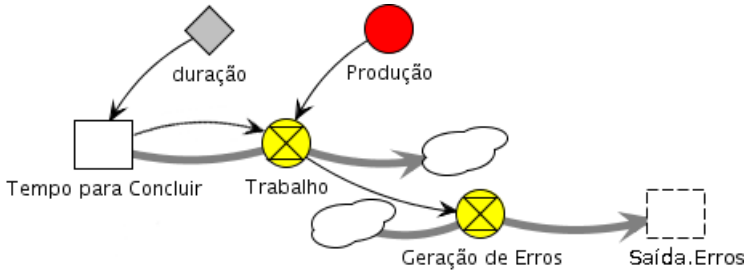

Figura 4.11: Instância de classe Atividade com cenário “Geração de Erros” aplicado.

Consideramos que a geração de erros é diretamente dependente do Trabalho. De forma análoga aos auxiliares externos, o estoque afetado, por ser externo à instância de classe, é representado por um retângulo tracejado.

Como uma instância pode ter mais de um cenário aplicado simultaneamente, a ordem de ligação de cenário é importante. Isto porque as conexões de cenários alteram as expressões contidas nas instâncias de modelo e, na maioria dos casos, não é comutativa. A Figura 4.12 apresenta uma instância de classe Atividade com um conjunto de cenários aplicados simultaneamente.

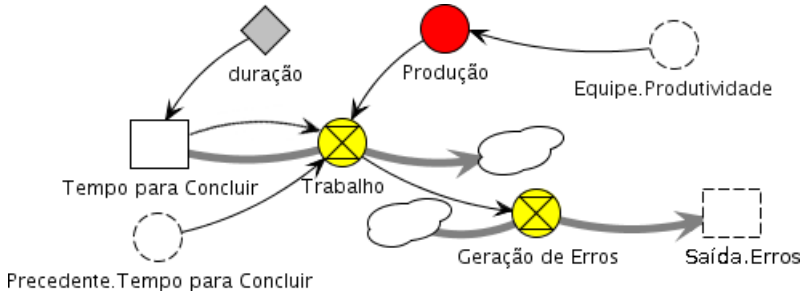

Figura 4.12: Instância de classe Atividade com vários cenários aplicados.

Estes modelos usados como exemplos são uma adaptação simplificada dos modelos encontrados na literatura. Podemos aumentar o nível de detalhamento e a complexidade dos modelos de domínio e cenários sem nos preocupar-mos com os modelos de instância. Esta característica torna os metamodelos de Dinâmica de Sistemas muito elegantes para lidar com situações onde há mais de um tipo de interessado nos modelos e simulações.

4.3 Conclusões Parciais

Os metamodelos de Dinâmica de Sistemas ou linguagem estendida de Dinâmica de Sistemas foram propostos como uma alternativa para lidar com o crescimento da complexidade dos modelos dos diagramas de estoque e fluxo quando há a necessidade de criar estruturas distintas que representam um mesmo conceito.

A abordagem permite criar modelos de domínio, instância e cenários. Os modelos de domínio descrevem os conceitos envolvidos por meio de classes e relacionamentos. Um modelo de instância usa instâncias de classe criadas com base nas descrições do modelo de domínio para representar um sistema real. Modelos de cenário são usados para capturar, de forma reutilizável, alterações estruturais nas instâncias de classes que modificam os seus comportamentos.

Esta divisão em modelos distintos permite que pessoas com diferentes níveis de conhecimento das técnicas de modelagem possam criar e discutir sobre um mesmo problema em duas camadas de abstração. Um especialista define modelos de domínio e cenário e, baseando-se nesses modelos, um conhecedor do domínio cria os modelos de intância para representar um sistema real.

Os metamodelos de Dinâmica de Sistemas foram definidos no trabalho original como uma linguagem textual. Modelos descritos nessa linguagem são compilados para os construtores de diagramas de estoque e fluxo de Dinâmica de Sistemas para então serem simulados. Acreditamos que esta abordagem insere um fator complicador quando há a necessidade de se criar simuladores onde a estrutura do modelo muda constantemente durante a simulação. Isto cria a necessidade de uma infraestrutura que permita manipular diretamente os modelos e simuladores durante o desenvolvimento de novas aplicações.

Esta necessidade é encontrada, particularmente, em ambientes de pesquisa onde os modelos e simuladores disponíveis na literatura não podem ser considerados como definitivos. Com isso em mente, faz-se necessário oferecer alternativas para construção de aplicações que vão utilizar modelos e simuladores de Dinâmica de Sistemas em sua versão tradicional ou estendida. No

Capítulo 5 desenvolvemos uma infraestrutura que permite a construção de aplicações que usam recursos de metamodelos e diagramas de estoque e fluxo onde o desenvolvedor tem uma maior liberdade para interagir como processo de modelagem e simulação.

5 *Arquitetura da Biblioteca de Simulação*

Neste capítulo apresentamos a arquitetura que construímos para a biblioteca *JynaCore API*, desenvolvida de forma a permitir a modelagem e simulação de modelos descritos na linguagem de estoque e fluxo de Dinâmica de Sistemas (FORRESTER, 1961) e na linguagem estendida proposta por Barros (2001). O uso dessa biblioteca é exemplificado pela construção de uma aplicação protótipo chamada *JynacoreSim* que consiste em um ambiente de simulação de modelos dinâmicos.

A organização deste capítulo foi feita em cinco seções: a Seção 5.1 apresenta os motivos que levaram a escolha da estrutura da biblioteca; a Seção 5.2 expõe a arquitetura que descreve os componentes de um processo de simulação; na Seção 5.3.1, apresentamos as estruturas das duas linguagens atualmente suportadas para construção de modelos; já a Seção 5.4 detalha o protótipo de ambiente JynacoreSim, construído utilizando a JynaCore API. Por fim, a Seção 5.6 expõe nossas conclusões do capítulo.

5.1 Justificativa da Abordagem de Implementação

O uso de ferramentas para modelagem e simulação baseados na linguagem de Dinâmica de Sistemas, em ciências econômicas e sociais, alavancou o desenvolvimento de uma grande gama de produtos profissionais no mercado. Ambientes comerciais como *PowerSim* (POWERSIM SOFTWARE, 2009), *VenSim* (VENTANA SYSTEMS, INC., 2009), *Stella* (ISEE SYSTEMS, 2009) apresentam o estado da arte em matéria de ambientes de modelagem. São dotados de editores gráficos de diagramas, realizam crítica de modelos durante sua construção e alguns possuem editores para interfaces com o usuário, de forma a facilitar a construção de simuladores mais amigáveis. Porém, além de cada ambiente acrescentar construtores próprios à linguagem de Dinâmica de Sistemas, são muito restritos a criar adaptações para integrá-los a outros ambientes. Quanto à facilidade para adicionar outros tipos de modelos, como o suporte à linguagem

estendida, faz-se necessário o uso de compiladores de maneira a realizar a tradução dos modelos. Estes ambientes também impõem restrições quanto ao uso de sistemas operacionais, sendo necessário a utilização de emuladores fora da plataforma para qual foram desenvolvidos. Por estarem vinculados à compra de licenças, versões de avaliação ou uso estritamente educacional, a utilização em instituições de ensino ou pequenas empresas deve ser avaliada caso a caso com base nos recursos disponíveis.

Existem opções não comerciais, como o *Illium* (BARROS, 2000) desenvolvido como parte do estudo do uso de simulação como ferramenta de auxílio à tomada de decisão em processos de software. Essa ferramenta substitui a representação gráfica dos modelos por uma linguagem textual e permite que os dados de uma simulação possam ser observados na forma de gráficos e tabelas. Projetos de código livre como *SystemDynamics* (MELCHER, 2009) e *Sphinxes* (KUZMENKO; ZAJTSEV; MIVERTFT, 2009) foram desenvolvidos utilizando a linguagem Java, portanto, são independentes de sistemas operacionais. Entretanto, estes projetos seguem a tendência das aplicações comerciais, focando em criar um ambiente gráfico de modelagem, não se preocupando em tornar o processo de confecção e simulação de modelos acessível externamente a outras aplicações.

Neste trabalho, optamos por capturar o processo básico de simulação representado na Figura 5.1, que consiste em utilizar um método numérico e a descrição de um modelo para gerar um conjunto de dados como resultado de simulação.

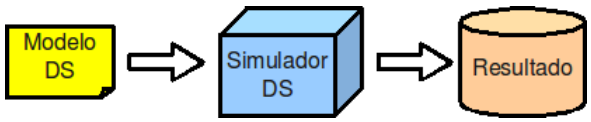

Figura 5.1: Processo básico de simulação de um modelo de Dinâmica de Sistemas.

Este processo básico é o núcleo mínimo para desenvolvedores que queiram acrescentar simulação computacional em suas aplicações. Tornando-o acessível na forma de uma biblioteca, permitimos que outras pessoas possam integrar ou adaptar a construção, modificação e simulação de modelos aos problemas que estiverem lidando.

Todos os ambientes citados não permitem a simulação direta dos modelos da linguagem estendida de Dinâmica de Sistemas, sendo necessário o uso de um compilador que faz a tradução dos modelos de domínio, instância e cenários, para os arquivos de um ambiente que trabalha com modelos de estoque e fluxo tradicionais. O *Hector* (BARROS, 2001) é um compilador de modelos descritos em linguagem estendida que faz a compilação para modelos de estoque e fluxo. A Figura 5.2 ilustra seu uso em conjunto com o Illium para modelar e simular sistemas

descritos na linguagem estendida.

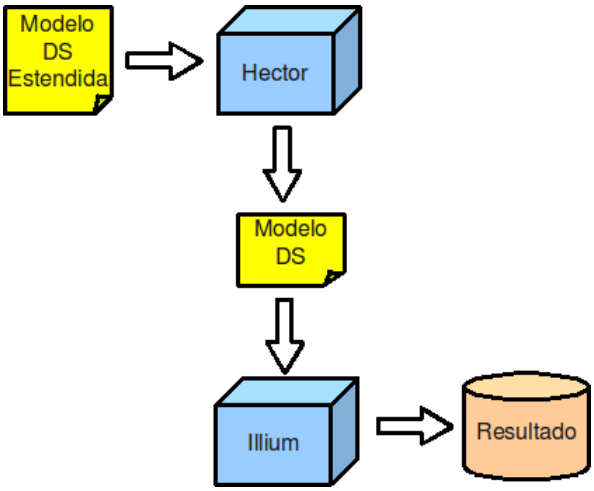

Figura 5.2: Processo de simulação de um modelo em linguagem estendida de Dinâmica de Sistemas usando a ferramenta *Hector*.

O processo de compilação implementado no Hector inclui técnicas de otimização de forma que os modelos gerados sejam eficientes, aproveitando melhor o uso de recursos de armazenagem e processamento. Acreditamos que a utilização de um compilador insere uma complexidade extra no processo de simulação. Isto dificulta a criação de aplicações que lidam com a estrutura do modelo, pois, durante uma simulação, é necessário associar os resultados de elementos compilados a elementos não compilados.

Neste trabalho retiramos o processo de compilação de metamodelos, colocando a linguagem estendida no mesmo nível dos modelos de estoque e fluxo. Com esta abordagem perdemos o passo de otimização, porém, ganhamos uma maior flexibilidade na manipulação dos modelos e simuladores. Entretanto, a nossa estrutura não exclui a possibilidade de uso de compiladores de modelos e, inclusive, pode ser usada para encapsular essa opção.

A nossa solução apresenta a JynaCore API, uma biblioteca de código livre, desenvolvida inteiramente em Java. Esta biblioteca captura o processo básico de simulação de modelos dinâmicos: descrição de modelos, simulação através de um método numérico e registro dos resultados. O diagrama da Figura 5.3 apresenta a estrutura básica do processo. Usuários de nossa biblioteca, cujo único interesse seja o resultado da simulação, podem simular os modelos em um nível mais elevado de abstração, não vinculando sua aplicação a uma linguagem específica.

Para os casos onde a aplicação precise lidar diretamente com a estrutura dos modelos, a JynaCore API torna disponível todos os construtores das linguagens de estoque e fluxo e da linguagem estendida de Dinâmica de Sistemas. Com a finalidade de manter a estrutura deste processo básico idêntico para ambas as linguagens suportadas, construímos um simulador que

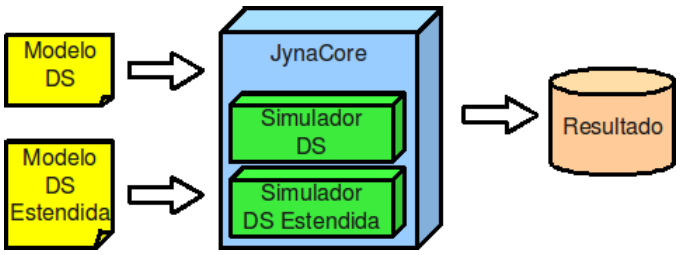

Figura 5.3: Processo de simulação de um modelos via JynaCore API.

opera diretamente sobre os modelos de instância da linguagem estendida.

O protótipo de ambiente para simulação de modelos, JynacoreSim, foi construído como prova de conceito do uso da JynaCore API. Este ambiente provê uma interface em *Java Swing* que, em conjunto com outras bibliotecas de código livre, tenta reproduzir as funções básicas dos ambientes tradicionais de modelagem e simulação. Nele é possível carregar os modelos, apresentar os diagramas que representam suas características estruturais e exibir os resultados da simulação de forma tabular ou em gráficos de linhas.

Os modelos expostos nos Capítulos 3 e 4 são usados como exemplos que representam um conjunto de modelos de domínio, instância e cenários voltados para treinamento de gestão de um projeto de software. O protótipo de simulador e a biblioteca permitem que estes modelos possam ser criados ou modificados e tenham seu comportamento estudado por meio de simulações. Modelos de outros domínios de conhecimento podem ser criados e simulados pelo ambiente, porém, estudos nessa linha não fazem parte do escopo deste trabalho.

5.2 Arquitetura e Projeto

Em todo projeto da JynaCore API nos preocupamos em descrever cada módulo funcional por interfaces[1], desacoplando a arquitetura de nossa implementação padrão que acompanha a biblioteca. Futuras extensões e melhorias visando desepenho ou compatibilidade com outros projetos podem ser facilmente criadas desta forma. Os pacotes das interfaces possuem subpacotes $*$.impl com as implementações que são prefixados com a palavra Default e seguidos pelo nome da interface que implementam. Por exemplo, a interface MetaModelClass do pacote br. ufjf . mmc.jynacore.metamodel possui uma implementação padrão DefaultMetaModelClass dentro do pacote br. ufjf .mmc.jynacore.metamodel.impl.

A JynaCore API é composta basicamente por um conjunto de interfaces que descreve mo-

[1]Padrão de projeto *Prototype* segundo (GAMMA et al., 1995).

delos, simuladores e o conjunto de dados resultante da simulação de um modelo dinâmico contendo seu comportamento em um período de tempo. Este conjunto básico é especializado em módulos específicos para a linguagem de modelagem desejada. Atualmente, apenas a linguagem de diagramas de estoque e fluxo e a linguagem estendida de Dinâmica de Sistemas estão disponíveis. Entretanto, isso não impede que outros tipos de modelos sejam criados.

5.2.1 Módulo Básico de Simulação

A arquitetura básica de uma simulação dentro da JynaCore API é descrita no pacote br.ufjf.mmc.jynacore e envolve o processo de gerar um conjunto de dados a partir de um modelo e seus parâmetros. Neste pacote se encontram as interfaces que descrevem o processo básico de simulação em alto nível de abstração. Os outros módulos estendem essas interfaces para implementar linguagens específicas, como a dos diagramas de estoque e fluxo e modelos em linguagem estendida de Dinâmica de Sistemas.

A funcionalidade central do processo de simulação é gerido pela interface JynaSimulation. Ela atua como um adaptador entre as demais interfaces envolvidas, atuando como mediadora conforme indicado na Figura 5.4.

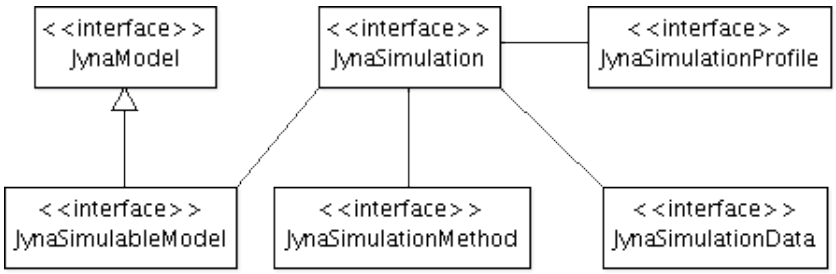

Figura 5.4: Diagrama de classes dos elementos de um processo básico de simulação em JynaCore API.

O JynaModel é a interface que descreve um modelo qualquer. Se este modelo puder ser simulado, ele também será um JynaSimulableModel. Atualmente, apenas modelos de estoque e fluxo de Dinâmica de Sistemas ou modelos de instância em metamodelos estendem JynaSimulableModel.

O JynaSimulation orquestra todos os passos de uma simulação a fim de que um JynaSimulableModel possa gerar um conjunto de dados. Este conjunto de dados é armazenado em JynaSimulationData, observa os elementos quantificáveis dos modelos e registra seus valores durante a simulação.

O JynaSimulatonProfile é um conjunto de parâmetros de configuração de uma simulação. Estes parâmetros são encontrados, comumente, como intervalos de tempo em que a simulação ocorre, o tamanho e quantidade de passos, etc. O JynaSimulationMethod encapsula o método

numérico que resolve o sistema de equações diferenciais com condições iniciais, contido implicitamente nos modelos de estoque e fluxo e linguagem estendida de Dinâmica de Sistemas.

5.2.2 Outras interfaces

A JynaCore API necessita de algumas interfaces auxiliares para operar e manter um comportamento comum entre elementos de modelos diferentes. As interfaces expostas na Figura 5.5 procuram garantir a compatibilidade entre as linguagens implementadas durante uma simulação.

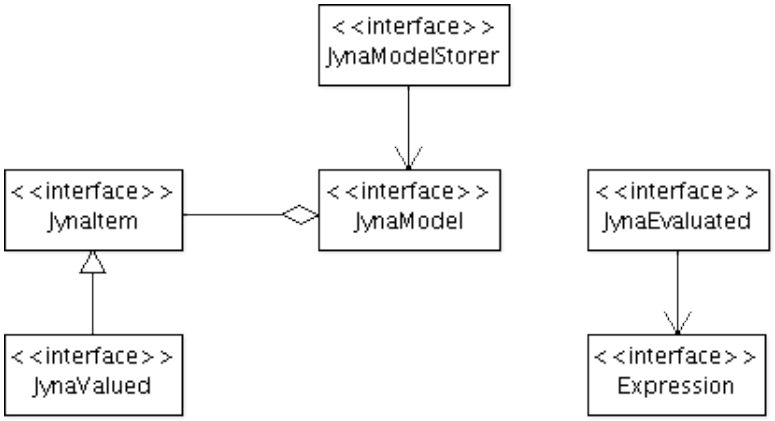

Figura 5.5: Diagrama de classes dos elementos auxiliares ao processo básico.

Um JynaModelStorer é responsável por obter e guardar um modelo a partir de uma *URI* arbitrária, desacoplando padrões para os tipos de arquivos e meios de armazenamento dos modelos. Dependendo de sua implementação, ele pode obter um modelo a partir de um arquivo texto, XML, banco de dados ou criar modelos a partir de dados provenientes de um sistema legado.

O JynaValued define o comportamento dos elementos internos dos modelos que possuam um valor numérico real, quantificável. Estes elementos são passíveis de ter seu valor registrado por um JynaSimulationData. A interface JynaEvaluated define o comportamento de objetos que possuam uma expressão matemática associada e poderão ter seus valores calculados a partir da resolução dessa expressão. Estas expressões por sua vez, são representadas pela interface Expression, do pacote br.ufjf.mmc.jynacore.expressions.

5.3 Linguagens disponíveis na Jynacore API

Atualmente, duas linguagens para construção de modelos dinâmicos são descritas e implementadas pela JynaCore API: os diagramas de estoque e fluxo, localizados no pacote br.

ufjf.mmc.jynacore.systemdynamics e os modelos da linguagem estendida de Dinâmica de Sistemas, disponíveis em br.ufjf.mmc.jynacore.metamodel. Ambas as linguagens também são descritas por interfaces e possuem uma implementação padrão associada.

5.3.1 Dinâmica de Sistemas

No Capítulo 3, apresentamos os construtores da Dinâmica de Sistemas e, em especial, os diagramas de estoque e fluxo. Esses diagramas são integrados à JynaCore API como uma das linguagens para descrição de modelos dinâmicos. As interfaces responsáveis por sua representação estão contidas no pacote br.ufjf.mmc.jynacore.systemdynamics e são apresentadas na Figura 5.6.

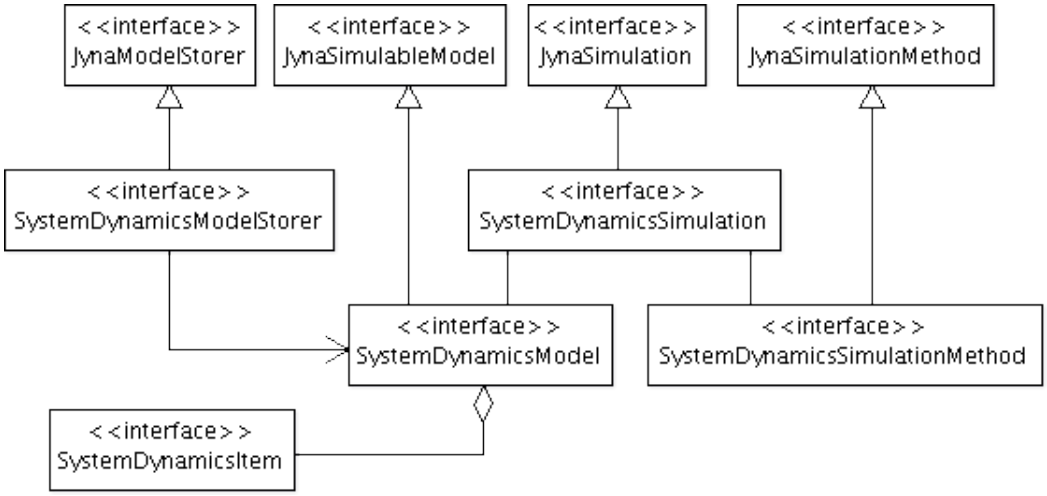

Figura 5.6: Diagrama de classes que apresenta a definição do modelo da linguagem de diagramas de estoque e fluxo na JynaCore API.

A interface SystemDynamicsModel se comporta como um conjunto de elementos descritos pela interface SystemDynamicsItem. Já as interfaces SystemDynamicsSimulation e SystemDynamicsMethod encapsulam, respectivamente, o simulador e os métodos de integração para modelos específicos de Dinâmica de Sistemas .

Os elementos básicos de Dinâmica de Sistemas são os *estoques finitos* (ou níveis), *estoques infinitos* (fontes ou sorvedouros), *taxas* (ou fluxos), *auxiliares* (variáveis ou constantes) e *informações*. Estes elementos são descritos pelo diagrama da Figura 5.7, onde vemos todas as classes que a JynaCore API suporta como elementos de um modelo de estoque e fluxo. Todos eles implementam a interface SystemDynamicsItem que, por sua vez, é um JynaItem a fim de manter a compatibilidade com as interfaces de simulação.

Os estoques são representados na JynaCore API pela interface Stock. Esta, por sua vez, é

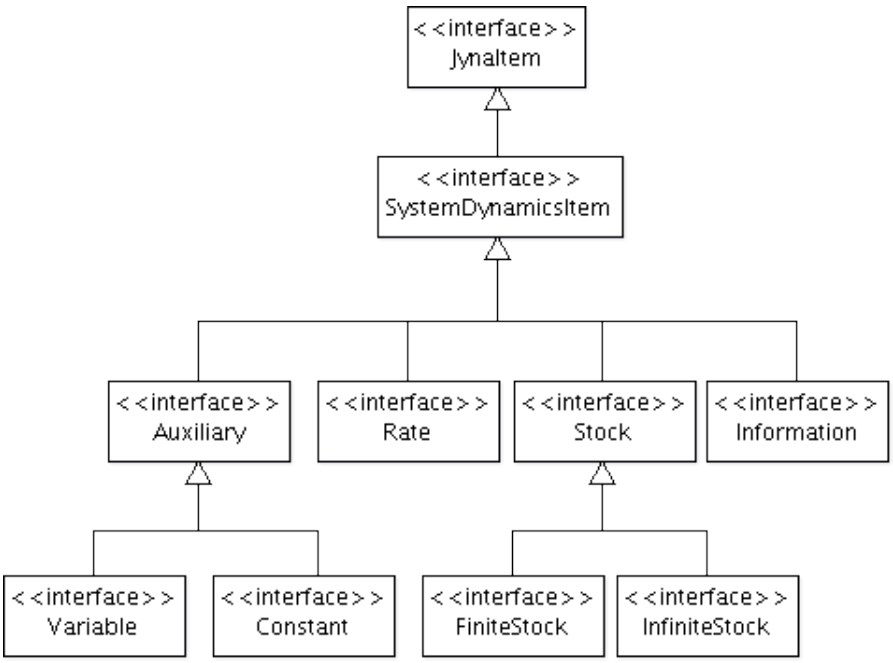

Figura 5.7: Diagrama de classes que apresenta a definição dos elementos de diagramas de estoque e fluxo na JynaCore API.

especializada pelas interfaces FiniteStock e InfiniteStock e representam, respectivamente, estoques finitos e estoques infinitos. Estoques finitos são estoques onde o acúmulo é quantificável e de interesse para o modelo do sistema. Descrevem o estado do modelo, pois são os únicos parâmetros necessários para obter os demais valores em um dado instante. Possuem valores iniciais, antes do início da simulação, e estes variam durante a sua execução de forma conservativa dentro dos limites do modelo. Já os estoques infinitos são conhecidos como fontes ou sorvedouros, de onde as quantidades podem vir ou ir indefinidamente. Representam estoques fora dos limites de interesse do modelo. Podemos entendê-los como estoques não quantificáveis, que consideramos infinitamente cheios e sempre capazes de acumular mais quantidades.

Uma taxa (ou fluxo) é modelada em JynaCore API pela interface Rate. As taxas ligam dois Stock para representar o fluxo conservativo das quantidades no modelo. Elas também realizam as conversões de unidades necessárias entre as quantidade envolvidas. Porém, na implementação atual, a JynaCore API não tem suporte a unidades e, portanto, uma limitação do modelo é que todos os valores são adimensionais. Como valor de uma taxa é calculado a partir de uma expressão matemática definida por Expression, ela também é uma JynaEvaluated.

A interface Auxiliary representa os elementos utilizados para cálculos indiretos como variáveis e constantes. As constantes são descritas pela interface Constant, podem guardar apenas um valor numérico. A interface Variable também é uma JynaEvaluated, pois seu valor depende de uma Expression.

O último elemento dos diagramas de estoque e fluxo são as informações, descritas pela interface Information e representam um fluxo de dados passado entre os elementos. Os elementos que podem participar deste tipo de ligação implementam a interface InformationSource e InformationConsumer, que representam, respectivamente, uma fonte e um consumidor de informação. As informações servem para validar as referências feitas nas expressões dos JynaEvaluated.

Um exemplo de construção de modelos de estoque e fluxo de Dinâmica de Sistemas pode ser visto no Apêndice B. O exemplo é construído programaticamente através da JynaCore API e, alternativamente utilizando uma implementação do JynaModelStorer que usa arquivos XML em um padrão próprio. Apesar de haver trabalhos em andamento (DIKER; ALLEN, 2005; CROWE, 2009), ainda está em aberto a discussão para definição de padrões, reconhecidos e internacionalmente, para arquivos e modelos de Dinâmica de Sistemas. Acreditamos que nossa implementação provê meios para a discussão e testes de padrões existentes e emergentes.

5.3.2 Linguagem estendida de Dinâmica de Sistemas

No Capítulo 4, apresentamos as bases da extensão da linguagem de Dinâmica de Sistemas criada por Barros (2001). Esta linguagem representa sistemas através de três modelos diferentes: modelos de domínio, modelos de instância e modelos de cenários. No trabalho original, estes modelos são conhecidos como *metamodelos de Dinâmica de Sistemas* por serem compilados para modelos de estoque e fluxo, para então realizar a sua simulação. A JynaCore API elimina a compilação dos metamodelos e coloca a linguagem estendida no mesmo patamar dos modelos de estoque e fluxo de Dinâmica de Sistemas. Torna-se necessário, então, descrever os elementos dos modelos, simuladores e métodos numéricos.

As interfaces responsáveis pela descrição dos modelos de domínio e cenários estão localizadas no pacote br.ufjf.mmc.jynacore.metamodel. Já as responsáveis pelos modelos de instância se encontram no pacote br.ufjf.mmc.jynacore.metamodel.instance.

Os modelos de domínio definem os conceitos que vão ser usados para representar elementos de um sistema cujo comportamento dinâmico está sob estudo. Portanto, não descrevem um modelo de um sistema, mas como construir um. São descritos pela interface MetaModel que é uma JynaModel.

Já os modelos de instância, utilizam as estruturas e relações definidas em um modelo de domínio para criar um modelo de um sistema real e portanto, simulável. São representados pela interface MetaModelInstance que é a única JynaModelSimulable da linguagem estendida.

Os modelos de cenários são mais próximos aos modelos de domínio. Eles descrevem alterações estruturais que as instâncias podem sofrer. São descritos pela interface MetaModelScenario que também é uma JynaModel.

A Figura 5.8 apresenta a ligação destes três modelos com as interfaces da camada mais abstrata da JynaCore API.

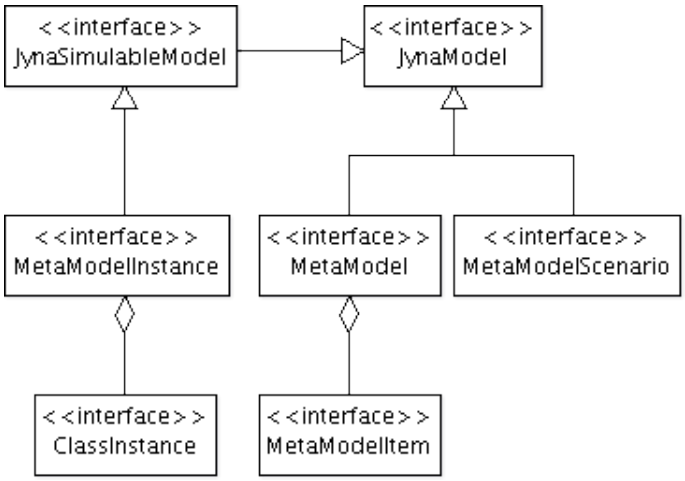

Figura 5.8: Diagrama de classes dos modelos de domínio, instância e cenários. O modelo de instância é o único que pode ser simulado.

O diagrama da figura 5.9 mostra que, apesar da estrutura de modelos da linguagem estendida ser mais complexa, o modelo de instância se comporta como um modelo simulável comum do ponto de vista das interfaces básicas da JynaCore API. Uma aplicação que tenha interesse apenas nos resultados da simulação pode interagir, apenas, com as interfaces mais simples, mesmo que na realidade interaja com três tipos de modelos diferentes da linguagem implementada.

Modelo de Domínio

Um modelo de domínio descreve os conceitos estruturais de um sistema com base em classes e nas interrelações que a instâncias dessas podem realizar. Para a descrição das classes são usados construtores dos diagramas de estoque e fluxo de Dinâmica de Sistemas, tabelas e propriedades. Como o diagrama da Figura 5.8 adiantou, a interface MetaModel é uma agregação de MetaModelItem. A interface MetaModelItem estende uma JynaItem com métodos de acesso a MetaModel a qual pertencem. Os descritores do modelo de domínio são especializados a partir da interface MetaModelItem conforme a Figura 5.10.

Os objetos que implementam a interface MetaModelClass se comportam como descritores

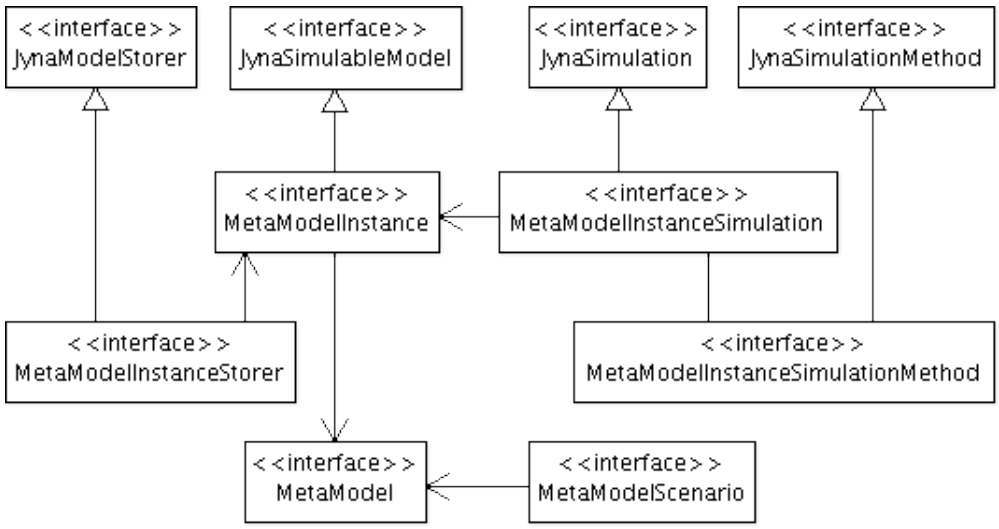

Figura 5.9: Diagrama de classes do simulador de modelo de instância: Estrutura complexa é vista como um processo simples JynaCore API.

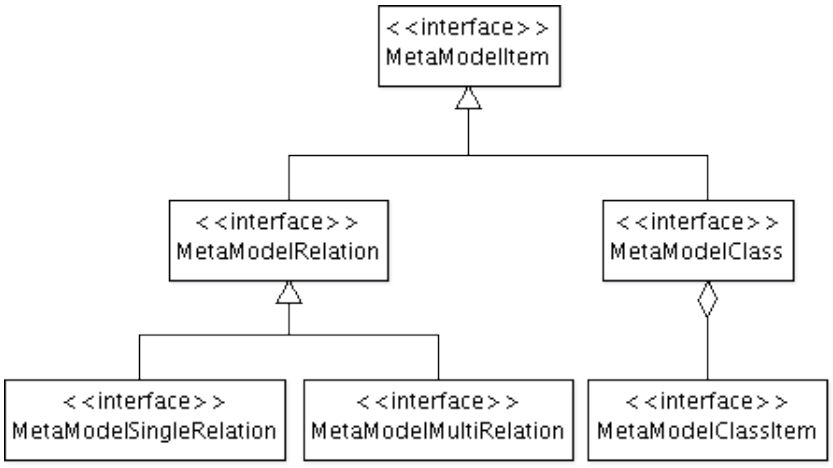

Figura 5.10: Diagrama de classes dos elementos de um modelo de domínio.

para a construção de instâncias de classe, em um modelo de instância. A esta descrição atribuímos o termo classe. Os relacionamentos entre estas as instâncias destas classes ficam a cargo de uma das duas interfaces que estendem MetaModelRelation: a MetaModelSingleRelation e MetaModelMultiRelation. Estas representam, respectivamente, os relacionamentos que só permitem uma única ou várias ligações simultâneas entre as instâncias de classes envolvidas.

Os objetos que agregam uma classe são implementações da interface MetaModelClassItem que são especializadas em MetaModelClassProperty ou um subtipo de MetaModelClassBehavior. Uma MetaModelClassProperty define um único valor real constante. É usada como parâmetro de configuração da instância durante a criação de uma instância de modelo. A Figura 5.11 mostra a relação das interfaces que descrevem os elementos de uma classe em um modelo de domínio. Já as interfaces que especializam MetaModelClassBehavior representam as regras para construção

dos elementos básicos de Dinâmica de Sistemas.

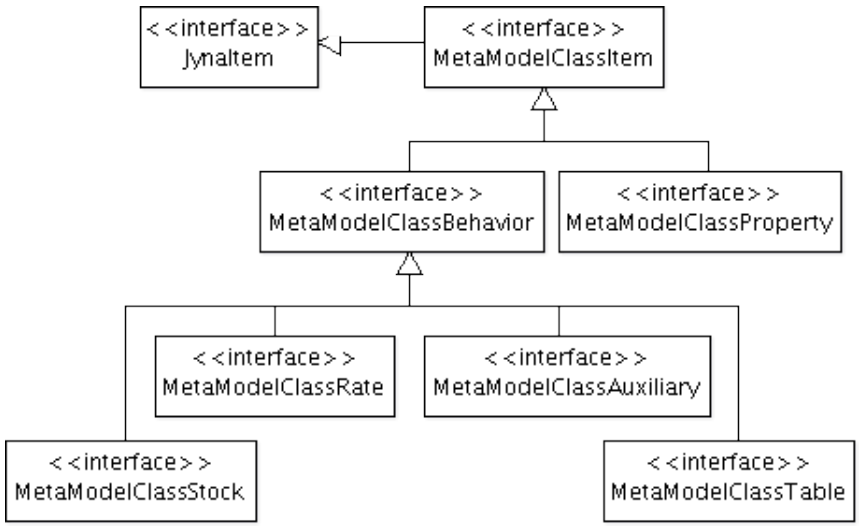

Figura 5.11: Diagrama de classes detalhado dos elementos de um modelo de domínio.

A interface MetaModelClassAuxiliary representa um auxiliar variável com sua equação associada e, portanto, estende JynaEvaluated. Os estoques finitos são representados pela interface MetaModelClassStock e, como nos diagramas de estoque e fluxo, representam o acúmulo de quantidades e o estado do subsistema que a instância de classe representa. A MetaModelClassRate é uma representação de uma taxa, ou seja, a variação das quantidades contidas nos estoques para reproduzir o comportamento dinâmico da classe. Elas afetam um estoque de origem e um de destino de acordo com uma expressão matemática (são uma extensão JynaEvaluated). Os estoques infinitos não possuem um elemento na linguagem estendida, mas podem ser representados se um do estoques afetados por uma taxa for deixado sem conexão. Já uma MetaModelClassTable define uma lista de valores constantes indexadas por um valor inteiro que é obtido a partir de uma função especial[2].

Modelo de Instância

A descrição dos modelos de instância na linguagem estendida é mantida pelas interfaces do pacote br. ufjf .mmc.jynacore.metamodel.instance. Os objetos que implementam a MetaModelInstance, tal como foi indicado no diagrama da Figura 5.8, são modelos completos e, portanto, podem ser simulados para estudo do comportamento do sistema.

Um modelo de instância é composto por instâncias de classes, representadas pela interface ClassInstance. Estas instâncias são construídas seguindo a estrutura da classe relacionada no

[2]Esta função *LOOKUP*, da mesma forma que no trabalho original (BARROS, 2001), realiza uma interpolação linear entre os valores da tabela, convertendo um intervalo e parâmetro reais em uma função discreta.

modelo de domínio. São compostas de objetos que implementam a interface ClassInstanceItem. O diagrama da figura 5.12 apresenta as interfaces dos elementos que são construídos junto com a instância de classe do modelo de domínio .

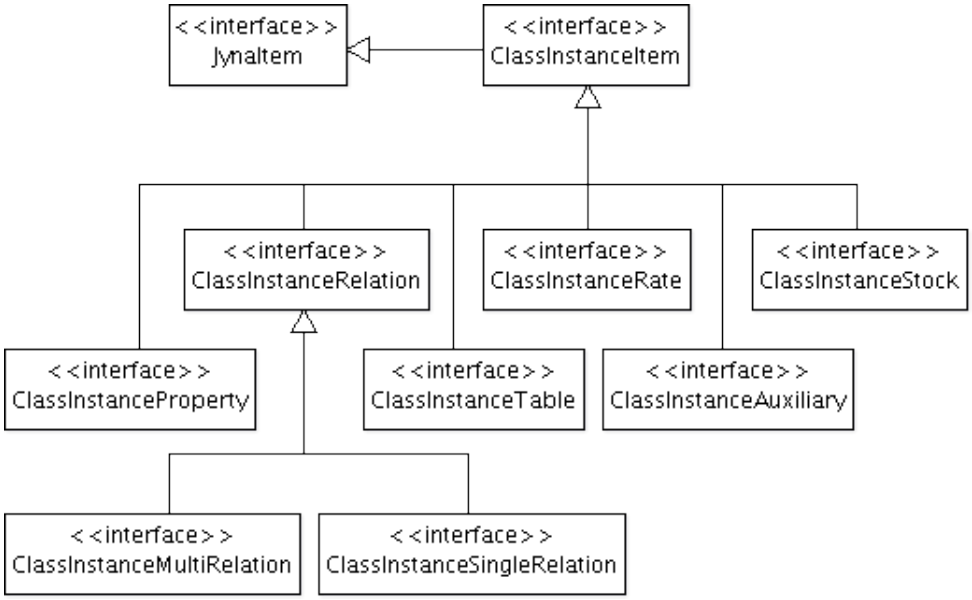

Figura 5.12: Diagrama de classes dos elementos de um modelo de instância.

Os objetos que implementam as interfaces ClassInstanceAuxiliary , ClassInstanceStock , ClassInstanceRate , ClassInstanceProperty e ClassInstanceTable representam as construções realizadas a partir dos elementos de classes nos modelos de domínio e passam a fazer parte de uma instância de classe. Seu comportamento é o mesmo dos elementos de um modelos de estoque e fluxo tradicional vistos na Seção 3. As interfaces ClassInstanceSingleRelation e ClassInstanceMultiRelation , descendentes de ClassInstanceRelation se referem aos relacionamentos que as instâncias de classes podem ter entre si.

Modelo de Cenário

Cenários são elementos reutilizáveis que descrevem alterações estruturais nas classes de um modelo de domínio específico. Sendo conectados a uma instância de classe em um modelo de instância, alteram seu comportamento. A partir dessa conexão, a instância de classe passa a ter estrutura diferente da sua classe padrão.

A interface MetaModelScenario é um JynaModel que descreve o cenário como um conjunto de MetaModelScenarioConnection. Os objetos que implementam MetaModelScenarioConnection descrevem como uma classe do modelo de domínio é alterada. Cenários são compostos por conjuntos de MetaModelClassItem, MetaModelScenarioConstraint e MetaModelScenarioAffect conforme diagrama da

figura 5.13.

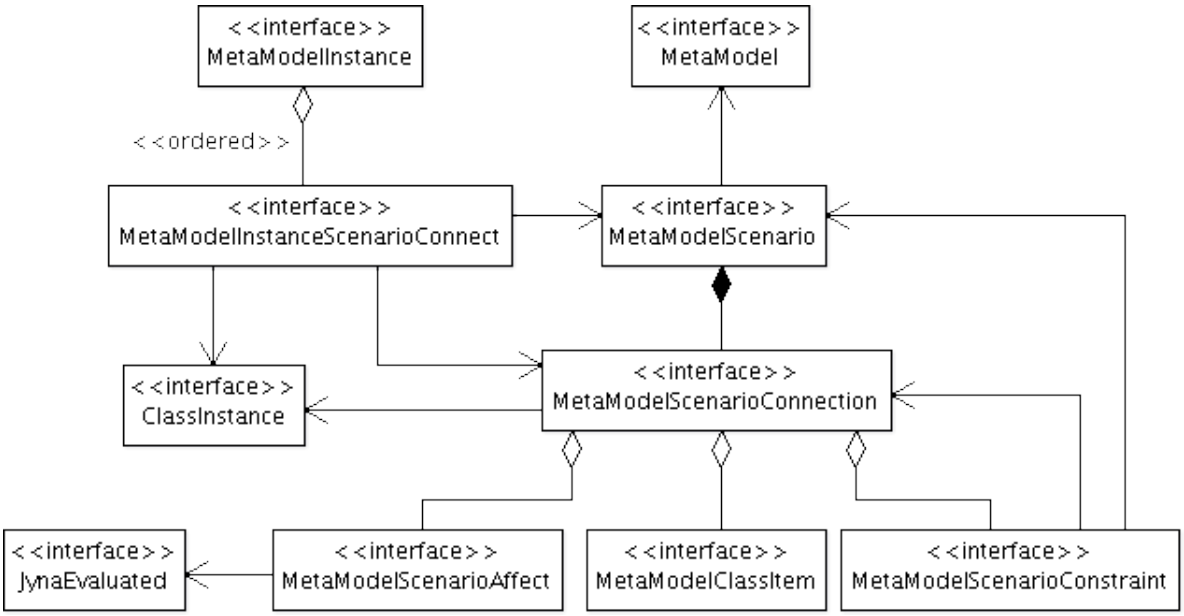

Figura 5.13: Diagrama de classes de modelos de cenários e seus componentes.

Durante a conexão, o conjunto de MetaModelClassItem é adicionado à estrutura da instância de classe ClassInstance, como se fizesse parte da classe no modelo de domínio original.

O MetaModelScenarioConstraint define quais outros cenários são pré-requisitos para a aplicação do cenário em questão. Um exemplo é o caso onde um cenário que modela a variação da produtividade de um desenvolvedor, em regime de horas extras, só fazer sentido se um cenário de horas extras estiver ativo previamente no desenvolvedor. Assim, um cenário com um pré-requisito irá realizar as conexões necessárias a outros cenários antes de ser aplicado.

O conjunto de MetaModelScenarioAffect contido no MetaModelScenarioConnection define as modificações nas equações dos elementos aos quais estão sendo conectados. Isto é necessário porque, com a inserção de novos elementos, o comportamento da instância será alterado.

A lista de objetos que implementam a interface MetaModelInstanceScenarioConnect descreve a ordem que os cenários devem ser conectados. Isto é importante devido ao fato que uma equação final é resultado da ordem das alterações das equações originais.

Um exemplo de um sistema contendo modelos de domínio, instância e cenários em linguagem estendida pode ser visto no Apêndice C. Ele pode ser construído programaticamente através da JynaCore API ou utilizando uma implementação do JynaModelStorer que usa um padrão próprio de arquivos XML.

5.4 Ambiente de Simulação

Com a JynaCore API e sua implementação padrão é possível construir, rapidamente, um simulador de modelos[3]. Entretanto, nem sempre é interessante explicitar a construção do simulador, principalmente, nos casos onde os modelos dos sistemas ainda estão sendo construídos.

Quando o objeto de estudo é o próprio modelo e seu comportamento, há a necessidade de ambientes mais elaborados a fim de simplificar a execução de simulações. Estas simulações podem ser resultado de um conjunto complexo de configurações, frutos de experimentações por parte do usuário final, exigindo uma interface amigável com usuário, e incluindo uma saída de dados visualmente mais rica, como gráficos que mostrem a variação do estado do modelo no tempo e seus diagramas estruturais.

Assim, construímos o ambiente *JynacoreSim* com o intuito de prover uma aplicação que apresente as características básicas das ferramentas de experimentação virtual, mas utilizando a JynaCore API como motor de descrição e simulação. O JynacoreSim define uma interface com o usuário em torno do nosso processo básico de simulação. Ele permite carregar modelos a partir de um arquivo no disco, expõe a estrutura dos seus diagramas, realiza as configurações de parâmetros e métodos numéricos necessárias para simulação e exibição do resultado. A Figura 5.14 apresenta uma captura de tela desse ambiente, onde podemos observar um modelo de Dinâmica de Sistemas com seu diagrama de estoque e fluxo.

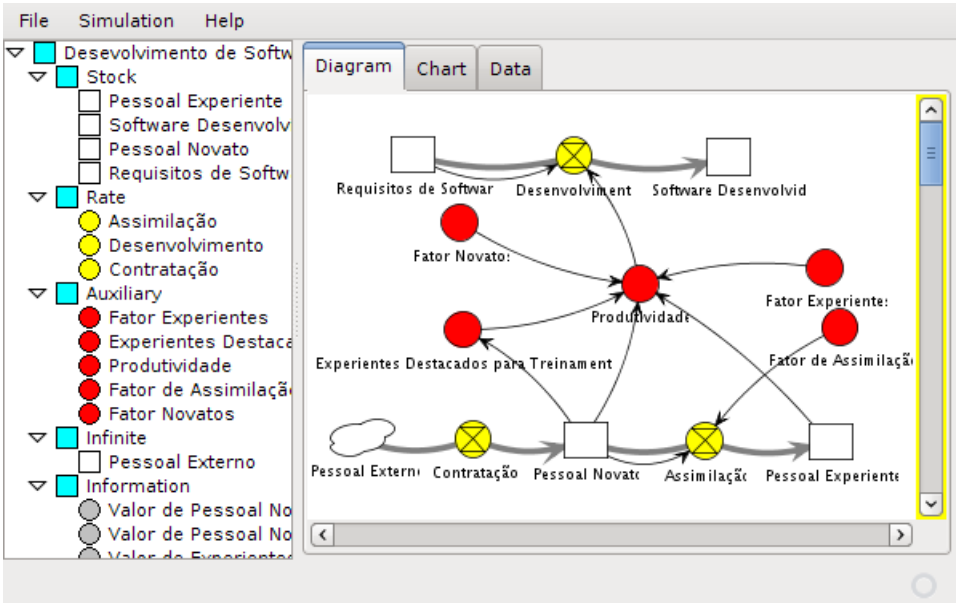

Figura 5.14: Diagrama de Dinâmica de Sistemas no JynacoreSim.

[3]Um exemplo de simulador que carrega um modelo e apresenta o resultado na saída padrão é construído, com poucas linhas de código, no Apêndice A.

Os modelos, quando simulados, apresentam o resultado em uma tabela para fácil visualização, conforme podemos observar na Figura 5.15. Os dados do resultado apresentam os valores observados pela interface JynaSimulationData em todos os intervalos de tempo.

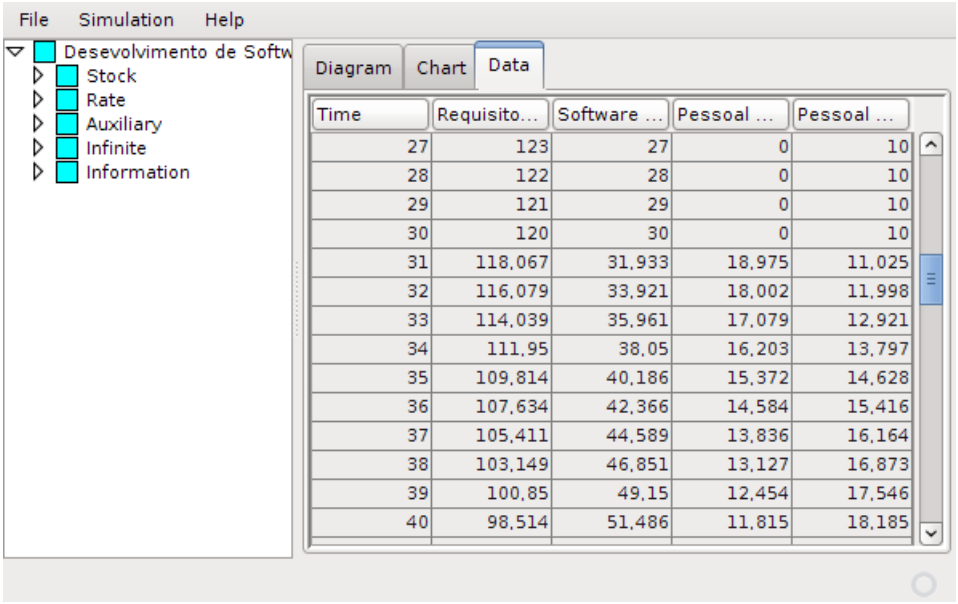

Figura 5.15: Resultado de uma simulação na forma tabular no JynacoreSim.

A Figura 5.16 mostra os dados oriundos da simulação do modelo. Estes dados são apresentados na forma de gráficos de linha, uma para cada elemento observado, e com os instantes de tempo no eixo das abscissas.

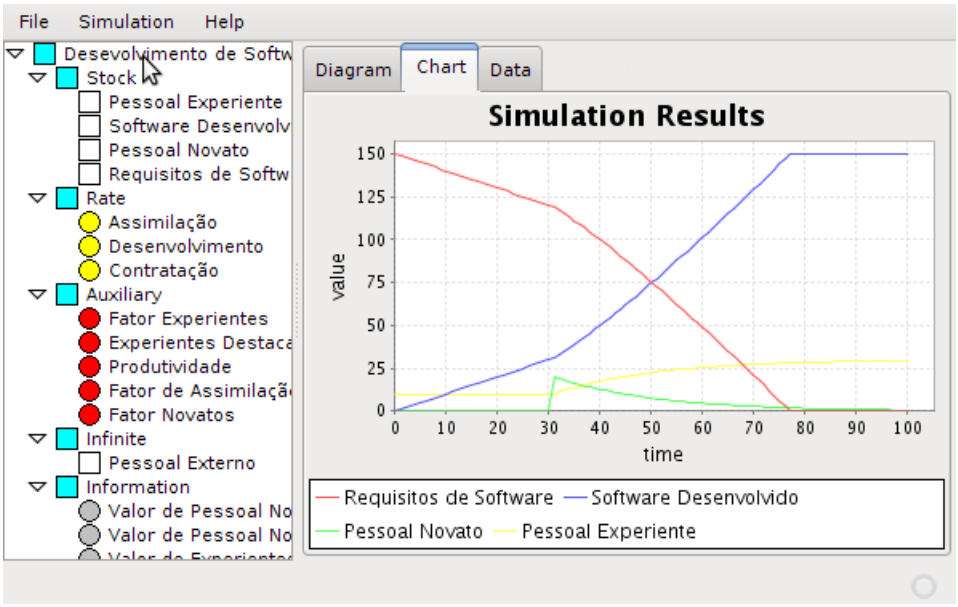

Figura 5.16: Resultado de uma simulação na forma de gráfico no JynacoreSim.

O ambiente JynacoreSim usa recursos simples para estabelecer, de forma transparente para o usuário, a configuração dos tipos de modelos (modelos de estoque e fluxo tradicionais ou de

instância da linguagem estendida), métodos numéricos de integração, visibilidade dos elementos e intervalo da simulação.

Cada linguagem de modelo possui seu próprio simulador e métodos de simulação, porém, em um nível de abstração mais elevado todos são vistos como um bloco báscio de simulação. A simulação é realizada diretamente sobre a estrutura dos modelos, e no caso dos metamodelos, o passo de compilação para diagramas de estoque e fluxo é eliminado. Ambas as linguagens, simuladores e métodos de integração foram reimplementados a partir das definições encontradas na literatura.

5.5 Tecnologias empregadas

A biblioteca JynaCore API e sua implementação básica foram desenvolvidas utilizando os recursos básicos do Java, disponíveis a partir da versão 5. As classes da implementação padrão seguem o padrão *Java Beans*[4] e, em sua maioria, também são POJOs[5].

O protótipo de ambiente de simulação JynacoreSim foi desenvolvido dentro do ambiente de desenvolvimento integrado *Netbeans* (SUN MICROSYSTEMS INC., 2009b), utilizando o conjunto de componentes para construção de interface com usuário Java Swing (SUN MICROSYSTEMS INC., 2005) junto do *Swing Application Framework* (SUN MICROSYSTEMS INC., 2006). Outras três bibliotecas externas são utilizadas, uma para apresentação de gráficos dos resultados (JFREE CHART PROJECT, 2000), uma para desenho dos diagramas dos modelos (JUNG PROJECT, 2003) e a JynaCore API como motor de modelagem e simulação.

A *JFreeChart* é uma biblioteca de software livre utilizada para construção de gráficos em Java. Ela é utilizada no protótipo de simulador para apresentar os resultados da simulação na forma de gráficos de linha. Construímos uma segunda implementação de JynaSimulationData, a DefaultSimulationDataXY que facilita a integração dos dados de saída com o tipo de dados usado pela biblioteca.

O JynacoreSim carrega a estrutura dos modelos suportados a partir de arquivos XML e a apresenta na forma de diagramas. Estes tentam seguir o padrão dos diagramas tradicionais de Dinâmica de Sistemas incluindo poucos elementos extras devido aos construtores da linguagem estendida. Este recurso é implementado pela biblioteca *JUNG*, desenvolvida para construção, análise e visualização de grafos.

[4]Classes Java com métodos de acesso padronizados e pelo menos um construtor sem parâmetros.
[5]Do inglês "*Plain Old Java Object*" ou objetos Java puros, sem herança de classes e interfaces externas.

A implementação padrão dos simuladores de Dinâmica de Sistemas e metamodelos disponibilizam dois métodos de integração numérica: o Euler explícito e Runge-Kutta ordem 4. Esses métodos foram adaptados para trabalhar diretamente a partir da estrutura dos modelos evitando conversões para a simulação. Apesar de ambos os métodos implementados apresentarem problemas de convergência e dependência do passo de integração, estes são considerados os mais encontrados na maioria das ferramentas de Dinâmica de Sistemas populares. Entretanto, nada impede que outros métodos mais robustos sejam implementados e usados de forma transparente pelas aplicações.

Espera-se construir uma comunidade de desenvolvimento em torno da nossa infraestrutura. Para tanto, todo o código fonte e documentação serão disponibilizados como um projeto de código livre e poderão ser acessados em `http://code.google.com/p/jynacore/`.

5.6 Conclusões Parciais

A construção de aplicações para modelagem e simulação de processos de desenvolvimento de software apresenta uma maior necessidade de flexibilidade devido ser comum na indústria de software a constante quebra de paradigmas e adaptação ou surgimento de novos processos.

O uso de ambientes de simulação de uso geral permite que modelos sejam criados e explorados rapidamente. Entretanto, durante a criação de novos modelos, as linguagens e seus ambientes podem oferecer limitações para explorar novas técnicas ou abordagens híbridas. Isso exige dos desenvolvedores um esforço extra para adaptar os recursos disponíveis em soluções que, por fim, são fortemente vinculadas aos ambientes.

Nosso trabalho contribuiu com o desenvolvimento de uma infraestrutura que permite ao desenvolvedor criar novos modelos, simuladores e aplicações com alto grau de flexibilidade. A responsável por estas funcionalidades é a nossa biblioteca livre JynaCore API, que foi desenvolvida em Java para implementar, a partir da definição, as linguagens de diagramas de estoque e fluxo e metamodelos de Dinâmica de Sistemas.

Como prova de conceito, desenvolvemos um simulador de uso geral, o JynacoreSim, onde é possível simular os modelos nas duas linguagens atualmente disponíveis na biblioteca. Adicionalmente, este simulador apresenta a estrutura dos modelos na forma de diagramas e os resultados na forma de tabelas e gráficos de linha.

A nossa abordagem permite que aplicações possam simular e interagir com modelos de estoque e fluxo e metamodelos de Dinâmica de Sistemas com um baixo acoplamento de im-

plementação. Novas linguagens e simuladores podem ser construídos de forma a expandir a biblioteca sem perder a compatibilidade com aplicações existentes.

Por fim, os simuladores atualmente implementados permitem que os modelos de processos de software da literatura, construídos com Dinâmica de Sistemas em diagramas de estoque e fluxo ou com metamodelos, possam ser estudados e expandidos.

6 *Considerações finais*

O estudo de modelos que reproduzem o comportamento de um sistema real é amplamente utilizado nas Engenharias como alternativa mais economicamente viável à implantação de um sistema real para testes. Nas últimas duas décadas procurou-se aplicar modelagem a sistemas que representam processos de desenvolvimento de software. Entretanto, os modelos tradicionais se baseiam em conhecimentos físicos bem estabelecidos e documentados, enquanto que os novos modelos de processos de software tentam capturar comportamentos dinâmicos observados apenas empiricamente. Então, ainda há um grande esforço na busca por novos modelos e ferramentas que comprovem que os comportamentos reais de um processo estão sendo capturados.

Os trabalhos de pesquisa podem ser divididos em dois grandes grupos: os que procuram aplicar as técnicas, modelos e simulações estabelecidos a processos reais como ferramentas de apoio à tomada de decisão e na busca de novos e melhores modelos de processos para aproximar ainda mais os comportamentos simulados dos reais. Em ambos os casos, há uma necessidade de desenvolvimento de aplicações novas, de caráter experimental.

Esta dissertação apresentou a implementação de uma infraestrutura aberta que permite a modelagem e simulação de modelos dinâmicos. Ela consiste em uma biblioteca com duas linguagens de modelos dinâmicos conhecidas da literatura e desenvolveu seus respectivos simuladores. Esta infraestrutura captura e torna acessível os constrututores mais fundamentais de um processo de modelagem e simulação. Com sua estrutura básica podem ser criadas novas aplicações ou modelos para estudos de sistemas dinâmicos como um processo de desenvolvimento de software.

Este capítulo está dividido em quatro seções. A Seção 6.1 apresenta as principais contribuições desta dissertação. Na Seção 6.2, indicamos algumas possibilidades de uso da biblioteca e ambiente de simulação com base em seu estado atual. A Seção 6.3.1 descreve algumas limitações das soluções propostas. E, por fim, a Seção 6.3.2 sugere algumas direções para melhorias e possíveis trabalhos futuros.

6.1 Contribuições

Este trabalho contribuiu com o desenvolvimento de uma infraestrutura aberta para ser utilizada na construção de aplicações que utilizão modelagem e simulação de sistemas dinâmicos. O principal elemento desta infraestrutura é biblioteca *JynaCore API*, de código livre e desenvolvida em Java. Os construtores desta biblioteca são especializados de forma a implementar duas linguagens de modelagem, baseadas nos conceitos de Dinâmica de Sistemas, e estruturas auxiliares que permitem a sua simulação. Os modelos construídos usando a JynaCore API podem ser descritos por diagramas de estoque e fluxo ou por metamodelos de Dinâmica de Sistemas.

A JynaCore API é utilizada para descrever as três partes componentes de um processo básico de simulação: os modelos de sistemas dinâmicos, os simuladores e os dados de saída. Os modelos contêm as estruturas e dados quantificados, que representam o estado do sistema em um determinado momento do tempo. O processo de simulação consiste em modificar estes valores segundo um método de integração numérica. Os dados registrados descrevem o comportamento dinâmico do sistema em intervalos de tempo regulares. A nossa solução procura encapsular esse fluxo de trabalho básico de simulação de modelos em um nível maior de abstração. Desta forma, é possível esconder os detalhes de implementação e facilitar a inserção das funcionalidades em outras aplicações que vão fazer uso de simulação de forma transparente da linguagem dos modelos.

Um protótipo de simulador, *JynaCoreSim* foi construído a fim de servir de prova de conceito para a JynaCore API. Este simulador, desenvolvido usando como interface gráfica uma aplicação Java Swing, lê arquivos XML contendo a descrição dos modelos na linguagem de estoque e fluxo ou metamodelos na linguagem estendida de Dinâmica de Sistemas. Ele executa a integração com base em um dos dois métodos numéricos implementados: Euler explícito e Runge-Kutta ordem 4. Estes podem ser escolhidos em tempo de execução no simulador e apresentam os resultados na forma de dados tabulares e gráficos de linha. O simulador também exibe os diagramas dos modelos de estoque e fluxo, domínio, instância e cenários usando uma identidade visual próxima a encontrada nos ambientes de Dinâmica de Sistemas tradicional.

Contribuímos, por fim, com um panorama histórico das técnicas e tendências relacionadas à construção de modelos e ferramentas para modelagem e simulação de processos de desenvolvimento de software. Incluímos uma revisão teórica das definições das linguagens atualmente suportadas que definem as bases deste trabalho ilustrada por um conjunto de exemplos construídos passo a passo com base nos modelos presentes na literatura.

6.2 Possibilidades de Uso

As possibilidades de uso para as contribuições desenvolvidas neste trabalho podem ser divididas em quatro grandes áreas: inserir modelagem e simulação em aplicações existentes; facilitar a construção de simuladores de processos; auxiliar a construção de novos modelos e ferramentas; auxiliar no ensino de processos de software e modelagem computacional.

A estrutura da JynaCore API permite que ela seja embutida facilmente em outras aplicações Java. Aplicativos de acompanhamento de projetos geram dados que servem para alimentar parâmetros dos modelos. A adaptação pode ser feita a partir da interface responsável por obter os modelos de um arquivo externo. Desta forma, é possível encapsular a conversão dos dados de seu formato padrão para os construtores das linguagens de modelos suportadas. A simulação deste modelo gerado pode ser usada para estimar o comportamento futuro do projeto de acordo com um dos modelos disponíveis na literatura. De forma análoga, as interfaces de saída podem ser estendidas para integrar o resultado das simulações aos indicadores de estimativas de custo, esforço ou cronograma de um processo real.

Aplicativos construídos para servirem de simuladores de projetos, frequentemente, usam recursos gráficos sofisticados e animações no lugar de gráficos e tabelas para apresentar resultados ao usuário. Tentam, assim, passar a sensação de imersão em um ambiente real e são conhecidos como "simuladores de vôo". Este tipo de simulador pode utilizar a JynaCore API como motor para agregar o comportamento dinâmico aos seus elementos durante uma simulação contínua, em um espaço de tempo ilimitado. Nos simuladores que utilizam as linguagens de metamodelos de Dinâmica de Sistemas a simulação é realizada por um ciclo iterativo de colher as ações do usuário, gerar um conjunto de metamodelos refletindo as ações em modificações estruturais, compilar os metamodelos, simular o novo modelo gerado e associar os resultados da simulação com elementos na interface com o usuário. A nossa abordagem permite eliminar o processo de compilação de metamodelos, permitindo simulá-los diretamente. Isso facilita a criação de aplicações onde a simulação deve ser feita levando em conta alterações constantes na estrutura dos modelos.

A própria formulação de modelos de processos ainda é uma questão em aberto e muito se tem feito para aproximar o comportamento destes modelos ao comportamento levantado pelo acompanhamento de processos reais. O uso da JynaCore API para criação de novos modelos é facilitada por sua estrutura aberta. Dada esta estrutura, cada componente do processo básico de simulação pode ser adaptado ou reescrito sem gerar grande esforço de adaptação nas aplicações desenvolvidas com a biblioteca. A própria estrutura do processo básico permite que se experi-

mente outras abordagens para construção de simulações mais complexas ou que combine outras abordagens como Eventos Discretos e Agentes aos Simuladores de Dinâmica de Sistemas.

Na literatura há um grande conjunto de modelos baseados em Dinâmica de Sistemas para capturar os comportamentos dinâmicos de um processo de software real. Professores e alunos interessados no estudo dos comportamentos que emergem das interrelações dos elementos de tais modelos podem utilizar o protótipo de ambiente de simulação JynacoreSim para experimentação mais imediata. O protótipo permite visualizar graficamente as estruturas dos modelos e o comportamento é exibido em gráficos de linhas, da mesma forma que nos simuladores tradicionais. Os professores podem inserir progressivamente o uso de modelos durante as aulas teóricas, combinando o estudo tradicional com modelos formais. Os alunos podem observar os efeitos de políticas e eventos comuns em modelo de processo de software, analisando cenários construídos previamente para experimentação. Exercícios podem ser propostos na forma de modelos para que os alunos tomem decisões de forma a contornar contingências como modificações na equipe, limitação de recursos e inserção de novos requisitos. Com os problemas descritos em um modelo formal, métricas podem ser propostas e as soluções podem ser avaliadas e comparadas quantitativamente, de forma bem diferente da abordagem tradicional de exposição de métodos que só são experimentados na prática.

6.3 Perspectivas Futuras

Esta seção expõe as limitações das soluções propostas, possíveis melhorias e sugestões para trabalhos futuros. A Figura 6.1 ilustra a organização destes comentários.

Na região central nos concentramos em listar as principais limitações das soluções propostas no âmbito do desenvolvimento da biblioteca JynaCore API e do protótipo de simulador JynacoreSim. Esta região apresenta uma característica mais interdisciplinar para compor a biblioteca.

Saindo da região central, destacamos algumas propostas de trabalhos com maior complexidade, porém julgamos ser aplicações relacionadas a este trabalho. Nestas propostas, cresce a exigência de conhecimentos mais específicos que devem ser explorados em maior profundidade.

6.3.1 Limitações e Melhorias

Uma limitação estrutural da JynaCore API é a falta de suporte as unidades nas descrições dos modelos. Um modelo quantitativo real contém as unidades das quantidades envolvidas,

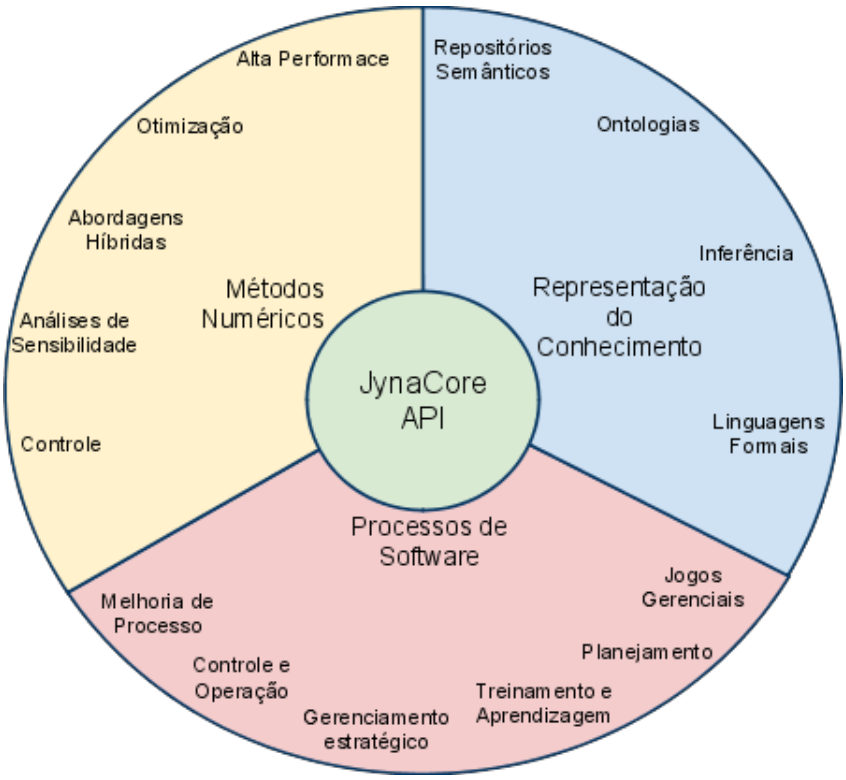

Figura 6.1: Perspectivas futuras por área disciplinar: ao centro do círculo as limitações e melhorias das soluções propostas; em direção aos bordos do círculo as possibilidades de aplicações e trabalhos futuros por área disciplinar.

estas unidades servem como um primeiro teste de corretude das taxas e estoques modeladas. Os modelos descritos atualmente pela JynaCore API são quantitativos, porém, o teste de coerência de unidades fica por conta do desenvolvedor do modelo. Portanto, é necessário uma maior atenção durante a confecção dos modelos para não inserir uma inconsistência.

A implementação padrão que acompanha a JynaCore API foi desenvolvida para minimizar o uso de bibliotecas externas, utilizando as estruturas de dados e tipos nativos do Java. A preocupação foi prover uma implementação semanticamente rica do ponto de vista das linguagens de modelagem implementadas. Esta opção foi tomada para tornar fácil a integração da estrutura dos modelos as ferramentas externas e não estruturas otimizadas para simulação de alto desempenho. Situações onde a escala dos modelos se torne um problema durante o processo de simulação podem ser resolvidas com o uso de bibliotecas otimizadas para cálculos numéricos como a COLT (HOSCHEK, 1999), que podem contribuir para a melhoria de desempenho dos métodos de integração.

A JynaCore API reimplementa as estruturas que definem analisadores semânticos das expressões matemáticas utilizadas nos modelos. Entretanto, uma limitação atual é que estas são usadas para ambas as linguagens. Uma possível melhoria na arquitetura é permitir que cada

linguagem de modelagem defina as funções e operadores próprios. Outra contribuição seria adicionar uma alternativa ao atual método de definição de expressões, na forma estruturada, para inserir um analisador léxico, permitindo definí-las na forma textual em modelos criados programaticamente.

A aplicação construída como protótipo, o JynacoreSim, apresenta uma interface para o uso básico da API: carregar modelos; configurar os parâmetros e métodos da simulação e exibir o resultado. Um ambiente completo para uso educacional inclui outros requisitos como: uma melhor interação visual com os modelos; mais recursos de apresentação dos resultados; interfaces ricas para análise de sensibilidade de variáveis e simulações de Monte Carlo. Essas funcionalidades ainda não estão presentes no ambiente. Entretanto, a arquitetura atual permite que sejam implementadas em uma próxima versão.

A implementação padrão da biblioteca e o protótipo de ambiente não possuem conversores para os tipos de arquivos utilizados em outras ferramentas de simulação disponíveis. Para os casos onde seja necessária a conversão, é preciso escrever uma implementação para a interface de persistência para que os modelos sejam carregados a partir de tais tipos de arquivos. A descrição dos modelos pelo bloco de persistência usa padrões próprios de arquivo, no formato XML. Entretanto, uma preocupação atual é o uso de padrões abertos para troca de informações e integração de ferramentas, mas apenas as expressões matemáticas embutidas são descritas em um padrão aberto Content MathML (WORLD WIDE WEB CONSORTIUM, 2001). Porém, acreditamos que a estrutura da biblioteca permite propor e experimentar o uso de padrões externos para auxiliar na descrição dos modelos suportados.

6.3.2 Trabalhos Futuros

Finalizamos esta dissertação com um conjunto de sugestões de trabalhos futuros que julgamos ser caminhos possíveis para continuação deste estudo. Lembramos, que esta lista não esgota, de forma alguma, as possibilidades na área.

Um desses caminhos é a adaptação da JynaCore API para conformá-la como bloco básico de aplicações de e-Science (LUDÄSCHER et al., 2006; BONIFACIO, 2008). Acreditamos que o estudo para se disponibilizar os simuladores da *JynaCore API* na forma de serviços, contribuirá para a integração de diversas ferramentas utilizadas durante o trabalho científico. Sendo interessante testar seu funcionamento através de ferramentas de orquestração de serviços como os ambientes Kepler (KEPLER PROJECT, 2004), Vistrails (VISTRAILS PROJECT, 2005) e Taverna (TAVERNA PROJECT, 2007) junto com uma aplicação externa que gere parâmetros e

estruturas para os modelos.

Outro possível trabalho pode incluir aos modelos de cenários, anotações semânticas de acordo com ontologias para estabelecer uma relação mais rica que, apenas, uma relação direta para com o modelo de domínio específico. A busca em um repositório pode retornar modelos de cenários completa ou parcialmente compatíveis, com base em inferências. Um trabalho realizado para incluir anotações semânticas na descrição de modelos dinâmicos, no domínio da Eletrofisiologia Computacional, foi descrito em Matos (2008) e o uso de repositórios de elementos matemáticos, também anotados semanticamente, em Matos et al. (2007). Em Liao, Qu e Leung (2005), os autores construíram uma ontologia para descrição de processos de software chamada *Software Process Ontology* - SPO. A partir da SPO, eles conseguem criar diferentes modelos de processos ou representar os já existentes como *Capability Maturity Model Integration* - CMMI, ISO/IEC 15504. Acreditamos que ao associarmos comportamentos dinâmicos aos componentes do SPO, ou a qualquer outra ontologia que descreva elementos de processos de software de uma maneira geral, podemos obter modelos dinâmicos genéricos para processos mapeados por ela. Acreditamos, ainda, que isto pode acelerar o processo de criação ou adaptação de modelos para processos de software.

O JynaCoreSim é um simulador de uso geral que foi construído como prova de conceito para a JynaCore API e sua independência das linguagem dos modelos e métodos de integração. Esta abordagem não procurou inserir representações gráficas específicas para domínios como processos de software. Para ambientes do tipo "simuladores de vôo" é necessária a construção de plataformas com mais recursos gráficos que reproduzam, com um maior grau de imersão, os conceitos do domínio em estudo. Esses ambientes podem ser criados de forma a desenvolver jogos gerenciais para uso educacional de uma maneira mais lúdica.

Uma estrutura de tratadores de eventos pode ser construída em torno do bloco básico do JynaCore a fim de se ter um simulador híbrido entre Dinâmica de Sistemas e Eventos Discretos. Esta estrutura pode definir descritores de eventos de forma que cenários e relações possam ser conectados ou desconectados durante a execução de uma simulação. A captura, armazenamento e reprodução desses eventos gerados pelo usuário, pode servir de base para estudos sobre as decisões tomadas por gerentes de projetos durante situações programadas.

6.4 Encerramento

Por limitação de tempo, este trabalho não consegue explorar, em profundidade, todos os problemas relacionados à área de modelagem de processos de software. Entretanto, serve como ponto de partida comum para envolver, de forma interdisciplinar, diversas áreas de estudo como: linguagens de programação; modelos formais; engenharia de software; gerenciamento de processos; representação do conhecimento e métodos numéricos.

Este trabalho partiu da necessidade de ferramentas abertas para construção de soluções que usam modelagem e simulação de Dinâmica de Sistemas. Ele provê uma biblioteca que é uma alternativa ao uso dos ambientes de modelagem convencionais, para construção de aplicações que utilizam modelos dinâmicos.

Lembramos que esta dissertação foi desenvolvida a partir de trabalhos que nortearam os estudos em modelagem de processos de software com Dinâmica de Sistemas nas últimas décadas. Nos preocupamos, desta forma, em manter a estrutura mais flexível possível para não limitar o campo de aplicação ou tecnologias empregadas. Acreditamos que a arquitetura aberta contribui para a criação de novas abordagens, principalmente nos campos de ensino e pesquisa.

Referências Bibliográficas

ABDEL-HAMID, T.; MADNICK, S. E. *Software Project Dynamics: An Integrated Approach*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991. ISBN 0-13-822040-9.

ARONSON, D. *Overview of Systems Thinking*. 1998. 3 p. `http://www.thinking.net/Systems_Thinking/OverviewSTarticle.pdf`. Cited: 09 Sep 2008.

BARROS, M. d. O. *Gerenciamento de Projetos Baseado em Cenários: Uma Abordagem de Modelagem Dinâmica e Simulação*. Tese (Doutorado) — COPPE/UFRJ, Rio de Janeiro, 2001.

BARROS, M. de O. Illium: Uma ferramenta de simulação de modelos dinâmicos de projetos de software. *Anais da Seção de Ferramentas do XIV Simpósio Brasileiro de Engenharia de Software*, 2000.

BOEHM, B. Foreword. In: ____. *Software Process Dynamics*. Hoboken, New Jersey: IEEE Press Wiley-InterScience, 2008. cap. Foreword. ISBN 978-0471274551.

BOEHM, B. et al. *Software Cost Estimation with COCOMO II*. Upper Saddle river, New Jersey: NJ:Prentice-Hall, 2000.

BONIFACIO, A. L. *Análise de Ferramentas Computadorizadas para suporte à Modelagem Computacional - Estudo de caso no domínio de dinâmica dos corpos deformáveis*. Dissertação (Mestrado) — Mestrado em Modelagem Coputacional/UFJF, Juiz de Fora, MG, 2008.

BROOKS. *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1975. 336 p.

CROWE, B. *System Dynamics Information Model - Project Page*. 2009. `http://sourceforge.net/projects/sdinfomodel/`. Cited: 10 Jun 2009.

CURTIS, B.; KELLNER, M. I.; OVER, J. Process modeling. *Commun. ACM*, ACM, New York, NY, USA, v. 35, n. 9, p. 75–90, 1992. ISSN 0001-0782.

DANTAS, A. R. *Jogos de simulação no treinamento de gerentes de projetos de software*. Rio de Janeiro: [s.n.], 2003. 98 p.

DIKER, V. G.; ALLEN, R. B. Xmile: towards an xml interchange language for system dynamics models. In: *System Dynamics Review*. [S.l.]: John Wiley Sons, Ltd., 2005. v. 21, n. 4, p. 351–359.

DONZELLI, P.; IAZEOLLA, G. *Hybrid Simulation Modelling of the Software Process*. Roma, Italy: [s.n.], 2000. `http://www.prosim.pdx.edu/prosim2000/paper/ProSimEA23.pdf`. Cited: 18 May 2008.

FORRESTER, J. W. *Industrial dynamics*. [S.l.]: The M.I.T. Press, 1961. ISBN 0-262-56001-1.

GALORATH, D. D.; EVANS, M. W. *Software Sizing, Estimation, and Risk Management: When Performance is Measured Performance Improves*. [S.l.]: CRC Press, 2006. 541 p.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Massachusetts: Addison-Wesley Publishing, 1995. 416 p.

HIRSCH, M. W.; SMALE, S.; DEVANEY, R. L. *Differential Equations, Dynamical Systems, and an Introduction to Chaos*. 2 ed. 2003. ed. San Diego, California: Elsevier Academic Press, 1974.

HOSCHEK, W. *Open Source Libraries for High Performance Scientific and Technical Computing in Java*. 1999. `http://acs.lbl.gov/~hoschek/colt`. Cited: 24 Mar 2009.

ISEE SYSTEMS. *STELLA Systems Thinking for Education and Research*. 2009. `http://www.iseesystems.com/`. Cited:.

JFREE CHART PROJECT. *JFreeChart a free Java chart library*. 2000. `http://www.jfree.org/jfreechart/`. Cited: 24 Mar 2009.

JUNG PROJECT. *JUNG: Java Universal Network/Graph Framework*. 2003. `http://jung.sf.net`. Cited: 24 Mar 2009.

KELLNER, M. Software process modeling and support for management planning and control. In: *Proceedings of the First International Conference on the Software Process*. Washington DC: IEEE Computer Society, 1991. p. 8–28.

KELLNER, M.; R, M.; D, R. Software process simulation modeling: Why? what? how? *Journal of Systems and Software*, 1999.

KEPLER PROJECT. *KEPLER - Scientific Workflows and Process Networks*. 2004. `http://kepler-project.org`. Cited: 24 Mar 2009.

KREIDER, D. L. et al. *An Introduction to Linear Analysis*. [S.l.]: Addison-Wesley, 1980. 773 p.

KUZMENKO, P.; ZAJTSEV, R. G.; MIVERTFT. *Sphinxes SD Tools*. 2009. `http://sourceforge.net/projects/sphinxes/`. Cited: 26 Fev 2009.

LAKEY, P. B. A hybrid software process simulation model for project management. *ProSim 2003*, 2003.

LIAO, L.; QU, Y.; LEUNG, H. K. N. *A Software Process Ontology and Its Application*. 2005. 10 p. `http://mel.nist.gov/msid/conferences/SWESE/repository/4proc_ont.pdf`. Cited: 12 Nov 2008.

LUDÄSCHER, B. et al. Scientific workflows: More e-science mileage from cyberinfrastructure. *Workshop on Scientific Workflows and Business Workflow Standards in E-Science at ESCIENCE'06*, Amsterdam, Netherlands, 2006.

MADACHY, R. *Software Process Dynamics*. [S.l.]: IEEE Press Wiley-InterScience, 2008. ISBN 978-0471274551.

MARTIN, R. H.; RAFFO, D. M. A model of the software development process using both continuous and discrete models. *International Journal of Software Process Improvement/Practice*, 2000.

MATOS, E. E. *CelOWS: Um Framework Baseado em Ontologias com Serviços Web para Modelagem Conceitual em Biologia Sistêmica*. Dissertação (Mestrado) — Mestrado em Modelagem Computacional,Universidade Federal de Juiz de Fora, UFJF, Juiz de Fora, Minas Gerais, 2008.

MATOS, E. E. et al. Mathws: Broker de serviços web para e-science. *2007 e-Science Workshop*, João Pessoa, Paraíba, 2007.

MELCHER, J. *SystemDynamics*. 2009. `http://sourceforge.net/projects/system-dynamics/`. Cited: 26 Fev 2009.

MI, P.; SCACCHI, W. *A Knowledge-based Environment for Modeling and Simulating Software Engineering Processes*. 1993. `http://www.usc.edu/dept/ATRIUM/Papers/Articulator.ps`. Cited: 12 Jun 2009.

MYRTVEIT, M. Object oriented extensions to system dynamics. *The Proceedings of the 18th International Conference of System Dynamics Society*, Bergen, Norway, Agosto 2000.

NAVARRO, E. *SimSE Homepage*. 2008. `http://www.ics.uci.edu/~emilyo/SimSE/`. Cited: 23 Oct 2008.

NAVARRO, E. O.; van der Hoek, A. *Software Process Modeling for an Interactive, Graphical, Educational Software Engineering Simulation Game*. 2004. 311-325 p. `http://www.ics.uci.edu/~emilyo/papers/SPIP2004.pdf`. Cited: 23 Apr 2009.

OGATA, K. *Engenharia de Controle Moderno*. [S.l.]: Prentice / Hall do Brasil, 1982.

OSTERWEIL, L. Software processes are software too. In: *ICSE '87: Proceedings of the 9th international conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987. p. 2–13. ISBN 0-89791-216-0.

PEDRO, M. V. *JLinkIt: Desenho e implementação de um ambiente de modelagem computacional para o ensino*. Dissertação (Mestrado) — Núcleo de Computação Eletrônica, Instituto de Matemática, Universidade Federal do Rio de Janeiro, 2006.

POWERSIM SOFTWARE. *Powersim Studio*. 2009. `http://www.powersim.com/`. Cited: 26 Fev 2009.

PRICE SYSTEMS. *TRUE-S User Manual*. 2005. `http://www.pricesystems.com`.

RAFFO, M. D. *Modeling Software Processses Quantitatively and Assessing the Impact of Potential Process Chnges on Process Performance*. Tese (Doutorado) — Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1995.

RICHMOND, B. System dynamics, systems thinking: Let's just get on with it! *International Systems Dynamics Conference in Sterling*, Sterling,Scotland, 1994.

SAMPAIO, F. F. *LINKIT: Design Development and Testing of a Semi-Quantitative Computer Modeling Tool*. Tese (Doutorado) — Departamento de Ciência e Tecnologia, Instituto de Educação da Universidade de Londres, London, England, 1996.

SENGE, P. *The Fifth Discipline*. New York, USA: Doubleday Business, 1990. 432 p. ISBN 0385260946.

SENGE, P. Leading learning organizations. In: *Training Development*. [S.l.: s.n.], 1996. v. 50, n. 12, p. 36.

SUN MICROSYSTEMS INC. *Swing (Java(TM) Foundation Classes)*. 2005. `http://java.sun.com/javase/6/docs/technotes/guides/swing`. Cited: 24 Mar 2009.

SUN MICROSYSTEMS INC. *Swing Application Framework*. 2006. `https://appframework.dev.java.net/`. Cited: 24 Mar 2009.

SUN MICROSYSTEMS INC. *Developer Resources for Java Developers*. 2009. `http://java.sun.com`. Cited: 24 Mar 2009.

SUN MICROSYSTEMS INC. *Netbeans IDE*. 2009. `http://www.netbeans.org/`. Cited: 24 Mar 2009.

TAVERNA PROJECT. *Taverna: Workflow Authoring Environment*. 2007. `http://taverna.sourceforge.net`. Cited: 10 May 2009.

VENTANA SYSTEMS, INC. *Vensim Simulation Software*. 2009. `http://www.vensim.com/`. Cited: 26 Fev 2009.

VILLELA, P. R. de C. *Ciência Viva*. Juiz de Fora, Minas Gerais, 2005.

VISTRAILS PROJECT. *VisTrails: A new scientific workflow and provenance management system*. 2005. `http://www.vistrails.org/`. Cited: 10 May 2009.

WIEST, J.; LEVY, F. *A Management Guide to PERT/CPM*. Englewood Cliffs, New Jersey: Prentice Hall, 1977.

WISE, A. *Little-JIL*. 2000. `http://laser.cs.umass.edu/tools/littlejil.shtml`. Cited: 24 Sep 2008.

WISE, A. *Little-JIL 1.5 Language Report*. Amherst, Ohio, 2006.

WISE, A. et al. *Using Little-JIL to Coordinate Agents in Software Engineering*. [S.l.], 2000.

WORLD WIDE WEB CONSORTIUM. *W3C Math*. 2001. `http://www.w3.org/Math/`. Cited: 24 Mar 2009.

ZHAO, X.; CHAN, K.; LI, M. Applying agent technology to software process modeling and process-centered software engineering environment. In: *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. New York, NY: ACM, 2005. p. 1529–1533. ISBN 1-58113-964-0.

APÊNDICE A – Exemplo de Aplicação usando a JynaCore API

Neste apêndice, utilizando a implementação padrão da JynaCore API, apresentamos duas formas de se criar uma simulação simples: a primeira via objetos Java e a segunda, recebe um modelo em arquivos XML e apresenta o resultado na saída padrão.

A.1 Uso da JynaCore API através da implementação padrão

Para usar a JynaCore API é necessário criar instâncias de: modelo; simulador; configurações de simulação e repositório da dados. Então, através da instância do simulador realiza-se toda a injeção de dependências.

O fluxo básico de trabalho pode ser construído diretamente a partir das implementações de acordo com o tipo de modelo a ser utilizado. A listagem A.1 mostra como podemos criar uma aplicação que lê um modelo em Dinâmica de Sistemas e expõe o resultado da simulação na saída padrão.

```
1  import (...)
2  public class SysoutSimulation {
3     public static void main(String[] args) {
4        try {
5           URI uri = new URI(args[0]);
6
7           JynaModelStorer storer = new DefaultSystemDynamicsModelStorerJDOM
               ();
8           JynaSimulation simulation = new DefaultSystemDynamicsSimulation();
9           JynaSimulationMethod method = new DefaultSystemDynamicsEulerMethod
               ();
10          JynaSimulationProfile profile = new DefaultSimulationProfile();
11          JynaSimulationData data = new DefaultSimulationData();
12
```

```
13            JynaModel model = storer.load(uri);
14
15            data.addAll(((JynaSimulableModel)model).getAllJynaValued());
16
17            simulation.setProfile(profile);
18            simulation.setMethod(method);
19            simulation.setModel((JynaSimulableModel) model);
20            simulation.setSimulationData(data);
21
22            simulation.reset();
23            simulation.run();
24
25            System.out.println(data);
26        } catch (Exception e) {
27            System.err.println(e);
28        }
29    }
30 }
```

Listagem A.1: Simulação de um modelo de Dinâmica de Sistemas para a saída padrão

Podemos destacar, ainda na listagem A.1, as linhas 7 a 9 criam as instâncias que são específicas dos modelos de Dinâmica de Sistemas e as linhas 10 e 11 criam as que são independentes de modelo. A linha 12 carrega o modelo a partir de uma URI criada a partir de um parâmetro da linha de comando na linha 5. A linha 15 coloca todos os elementos que possuem valores para registro no repositório de dados da simulação. As linhas 17 a 20 realizam a injeção de dependências a partir da instância de simulação. Os comandos das linhas 22 e 23 realizam uma simulação completa, usando a configuração padrão e o resultado é registrado na instância de dados. O conteúdo é impresso na saída padrão na linha 25.

O mesmo procedimento é feito na listagem A.2, onde apresenta o mesmo programa para uma aplicação que lê um modelo de instâncias em linguagem estendida de Dinâmica de Sistemas e expõe o resultado da simulação na saída padrão.

```
1 import (...)
2 public class SysoutMMSimulation {
3   public static void main(String[] args) {
4     try {
5       URI uri = new URI(args[0]);
6
7       JynaModelStorer storer = new DefaultMetaModelInstanceStorerJDOM();
8       JynaSimulation simulation = new DefaultMetaModelSimulation();
```

```
9         JynaSimulationMethod method = new DefaultMetaModelRK4Method();
10        JynaSimulationProfile profile = new DefaultSimulationProfile();
11        JynaSimulationData data = new DefaultSimulationData();
12
13        JynaModel model = storer.load(uri);
14
15        data.addAll(((JynaSimulableModel)model).getAllJynaValued());
16
17        simulation.setProfile(profile);
18        simulation.setMethod(method);
19        simulation.setModel((JynaSimulableModel) model);
20        simulation.setSimulationData(data);
21
22        simulation.reset();
23        simulation.run();
24
25        System.out.println(data);
26      } catch (Exception e) {
27        System.err.println(e);
28      }
29    }
30  }
```

Listagem A.2: Simulação de um modelo de instância de metamodelos de Dinâmica de Sistemas para a saída padrão

Podemos observar que a única mudança em relação ao código da lista A.1 ocorre na instanciação dos elementos nas linhas 7 a 9. Portanto, todo o código seguinte que realiza a simulação é independente da implementação e modelo.

A.2 Uso da JynaCore API através de uma Factory

Nos exemplos da seção A.1, o código que cria as instâncias dos elementos de uma simulação JynaCore API acopla a aplicação a um tipo de modelo específico. O código da listagem A.1 se prende ao modelo de estoque e fluxo da Dinâmica de Sistemas e o da código da A.2 aos metamodelos da linguagem estendida.

Quando for de interesse do projeto evitar o relacionamento com uma linguagem específica, pode-se diminuir o acoplamento pelo uso de uma *Factory* (GAMMA et al., 1995). A listagem A.3 apresenta o uso de uma interface e classe criadas para isolar as linguagens de modelagem:

```
1  import (...)
2  public class SysoutFactorySimulation {
3    public static void main(String[] args) {
4      try {
5        URI uri = new URI(args[1]);
6
7        JynaFactory factory = new DefaultJynaFactory();
8        factory.setTypeByURI(uri);
9        JynaModelStorer storer = factory.newModelStorer();
10       JynaSimulation simulation = factory.newModelSimulation();
11       JynaSimulationMethod method = factory.newModelMethod();
12       JynaSimulationProfile profile = factory.newSimulationProfile();
13       JynaSimulationData data = factory.newSimulationData();
14
15       JynaModel model = storer.load(uri);
16
17       data.addAll(((JynaSimulableModel)model).getAllJynaValued());
18
19       simulation.setProfile(profile);
20       simulation.setMethod(method);
21       simulation.setModel((JynaSimulableModel) model);
22       simulation.setSimulationData(data);
23
24       simulation.reset();
25       simulation.run();
26
27       System.out.println(data);
28     } catch (Exception e) {
29       System.err.println(e);
30     }
31   }
32 }
```

Listagem A.3: Simulação de um modelo para a saída padrão usando Factory para isolar o tipo de modelo.

O código apresenta a mesma estrutura dos anteriores, porém, ele usa um menor número de importações a implementações. Construímos a classe *DefaultJynaFactory* apenas para dar uma idéia de como isolar os as linguagens de modelos implementadas pela distribuição padrão.

APÊNDICE B – Exemplo de um modelo de dinâmica de sistemas

Este Apêndice mostra como podemos criar programaticamente um modelo em Dinâmica de Sistemas em Java, usando a implementação padrão fornecida junto da JynaCore API. Alternativamente, apresentamos um arquivo XML contendo o mesmo modelo para ser usado em conjunto com a implementação do `JynaModelStorer`.

A listagem B.1 apresenta o código em java para a criação do modelo de estoque e fluxo. Desta forma podemos interagir com o modelo sem usar um JynaModelStorer.

```
1  SystemDynamicsModel modelSD = new
2  DefaultSystemDynamicsModel(); modelSD.setName("Crescimento de Usuários");
3
4  FiniteStock users = new DefaultFiniteStock();
5  users.setName("Usuários");
6  users.setInitialValue(10.0);
7  modelSD.put(users);
8
9  InfiniteStock userPool = new DefaultInfiniteStock();
10 userPool.setName("Possíveis Usuários");
11 modelSD.put(userPool);
12
13 Variable incFactor = new DefaultVariable();
14 incFactor.setName("Taxa de Adesão");
15 incFactor.setExpression(new DefaultNumberConstantExpression(0.05));
16 modelSD.put(incFactor);
17
18 Rate userInc = new DefaultRate();
19 userInc.setName("Adesão");
20 userInc.setSource(userPool);
21 userInc.setTarget(users);
22 userInc.setSourceAndTarget(userPool, users);
```

```
23
24 Expression exp = new DefaultExpression();
25 exp.setOperator(NumberOperator.TIMES);
26 exp.setLeftOperand(new DefaultReferenceExpression(incFactor));
27 exp.setRightOperand(new DefaultReferenceExpression(users));
28 userInc.setExpression(exp);
29 modelSD.put(userInc);
30
31 Information info1 = new DefaultInformation(users,userInc);
32 modelSD.put(info1);
33 Information info2 = new DefaultInformation(incFactor,userInc);
34 modelSD.put(info2);
```

Listagem B.1: Modelo de estoque e fluxo construído programaticamente.

Alternativamente, o modelo ode ser descrito em um padrão próprio de arquivo XML, com extensão ".jyna". Este padrão foi desenvolvido para ser legível e de fácil entendimento. Ele inclui na sua descrição apenas os dados fundamentais para a descrição do modelo e não dados de apresentação de diagramas. A listagem B.2 expõe o conteúdo de um arquivo que descreve o modelo anterior.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <systemDynamicsModel  xmlns:m="http://www.w3.org/1998/Math/MathML">
3    <name>Crescimento dos usuários de um Software sem Concorrência</name>
4    <stocks>
5      <finiteStock>
6        <name>Usuários</name>
7        <initialValue>
8          <m:math>
9            <m:cn>10.0</m:cn>
10         </m:math>
11       </initialValue>
12     </finiteStock>
13   </stocks>
14   <infiniteStocks>
15     <infiniteStock>
16       <name>Possíveis Usuários</name>
17     </infiniteStock>
18   </infiniteStocks>
19   <rates>
20     <rate>
21       <name>Adesão</name>
```

```
22        <expression>
23          <m:math>
24            <m:apply>
25              <m:times />
26              <m:ci>Taxa de Adesão</m:ci>
27              <m:ci>Usuários</m:ci>
28            </m:apply>
29          </m:math>
30        </expression>
31        <source>Possíveis Usuários</source>
32        <target>Usuários</target>
33      </rate>
34    </rates>
35    <auxiliaries>
36      <auxiliary>
37        <name>Taxa de Adesão</name>
38        <expression>
39          <m:math>
40            <m:cn>0.05</m:cn>
41          </m:math>
42        </expression>
43      </auxiliary>
44    </auxiliaries>
45    <informations>
46      <information>
47        <name>Taxa atual de Adesão</name>
48        <source>Taxa de Adesão</source>
49        <consumer>Adesão</consumer>
50      </information>
51      <information>
52        <name>Quantidade atual de Usuários</name>
53        <source>Usuários</source>
54        <consumer>Adesão</consumer>
55      </information>
56    </informations>
57  </systemDynamicsModel>
```

Listagem B.2: Modelo de estoque e fluxo em XML.

Estes modelos podem ser usados em simulações como nas aplicações de exemplo do Apêndice A.

APÊNDICE C – Exemplo de um modelo de domínio, cenário e instância em JynaCore API

Este Apêndice mostra como podemos criar programaticamente um modelo de domínio, instância e cenário usando a linguagem estendida de Dinâmica de Sistemas. Para tanto, são usadas as interfaces e classes da implementação padrão da JynaCore API. Alternativamente, apresentamos arquivos XML que contêm os mesmos modelos para serem usados em conjunto com uma implementação do JynaModelStorer.

A listagem C.1 apresenta um fragmento de código que cria o modelo de domínio contendo duas classes e duas relações.

```
1  // Modelo de Domínio
2  MetaModel domainModel = new DefaultMetaModel();
3  domainModel.setName("Projeto de Software Simples");
4
5  // Classe Desenvolvedor
6  MetaModelClass developer = new DefaultMetaModelClass();
7  developer.setName("Desenvolvedor");
8
9  MetaModelClassProperty experience = new DefaultMetaModelClassProperty();
10 experience.setName("experiência");
11 experience.setDefaultValue(1.0);
12 developer.put(experience);
13
14 MetaModelClassAuxiliary productivity = new DefaultMetaModelClassAuxiliary()
       ;
15 productivity.setName("Produtividade");
16 productivity .setExpression(new DefaultNameExpression("experiência"));
17 developer.put(productivity);
18
```

```
19 domainModel.put(developer);
20
21 // Classe Atividade
22 MetaModelClass activity = new DefaultMetaModelClass();
23 activity.setName("Atividade");
24
25 MetaModelClassProperty duration = new DefaultMetaModelClassProperty();
26 duration.setName("duração");
27 duration.setDefaultValue(25.0);
28 activity.put(duration);
29
30 MetaModelClassStock timeToConclude = new DefaultMetaModelClassStock();
31 timeToConclude.setName("Tempo para Concluir");
32 timeToConclude.setExpression(new DefaultNameExpression("duração"));
33 activity.put(timeToConclude);
34
35 MetaModelClassAuxiliary production = new DefaultMetaModelClassAuxiliary();
36 production.setName("Produção");
37 production.setExpression(new DefaultNumberConstantExpression(1.0));
38 activity.put(production);
39
40 MetaModelClassRate work = new DefaultMetaModelClassRate();
41 work.setName("Trabalho");
42 work.setSource(timeToConclude);
43 Expression workExp = new DefaultExpression();
44 workExp.setOperator(NumberOperator.MINIMUM);
45 workExp.setLeftOperand(new DefaultNameExpression("Produção"));
46 workExp.setRightOperand(new DefaultNameExpression( "Tempo para Concluir"));
47 work.setExpression(workExp);
48 activity.put(work);
49
50 domainModel.put(activity);
51
52 MetaModelRelation team = new DefaultMetaModelSingleRelation();
53 team.setName("Equipe");
54 team.setSource(activity);
55 team.setTarget(developer);
56 domainModel.put(team);
57
58 MetaModelRelation precedence = new DefaultMetaModelMultiRelation();
59 precedence.setName("Precedente");
60 precedence.setSource(activity);
61 precedence.setTarget(activity);
```

```
62 domainModel.put(precedence);
```

Listagem C.1: Modelo de domínio construído programaticamente.

Os comandos das linhas 2 e 3 criam o modelo de domínio que recebe as classes e relacionamentos. O processo de criação de classes se assemelha muito à criação de um diagrama de estoque e fluxo e uma classe pode ser vista como um diagrama isolado. As linhas 5 a 19 definem a estrutura de uma classe Desenvolvedor através de uma propriedade e um auxiliar. As linhas 21 a 50 criam os elementos da classe Atividade, composta por um auxiliar, uma propriedade, um estoque finito e uma taxa.

Os relacionamentos estruturais que as instâncias de classes poderão ter entre si também são definidos no modelo de domínio. Um relacionamento simples, Equipe, associa uma atividade a um desenvolvedor nas linhas 52 a 56. Um relacionamento múltiplo, chamado Precedente, representa as dependências entre as Atividades, é criado nas linhas 58 a 62. Todos as classes e relacionamentos são inseridos no modelo de domínio após sua criação.

Uma implementação do JynaModelStorer específica para trabalhar com modelos de domínio permite que estes possam ser armazenados e acessados a partir de arquivos XML com extensão ".jymm". A estrutura deste arquivo foi desenvolvida de forma a ser legível e usa o padrão *Content MathML* (WORLD WIDE WEB CONSORTIUM, 2001) para representação das expressões matemáticas. O código da listagem C.2 apresenta o conteúdo do arquivo contendo a representação do mesmo modelo de domínio construído programaticamente na listagem C.1.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <metaModel xmlns:m="http://www.w3.org/1998/Math/MathML">
3    <name>Projeto de Software Simples</name>
4    <classes>
5      <class>
6        <name>Atividade</name>
7        <items>
8          <finiteStock>
9            <name>Tempo para Concluir</name>
10           <initialValue>
11             <m:math>
12               <m:ci>duração</m:ci>
13             </m:math>
14           </initialValue>
15         </finiteStock>
16         <rate>
```

```
17          <name>Trabalho</name>
18          <expression>
19            <m:math>
20                <m:apply>
21                  <m:max />
22                  <m:apply>
23                    <m:min />
24                    <m:ci>Tempo para Concluir</m:ci>
25                    <m:ci>Produção</m:ci>
26                  </m:apply>
27                  <m:cn>0.0</m:cn>
28                </m:apply>
29            </m:math>
30          </expression>
31          <source>Tempo para Concluir</source>
32        </rate>
33        <property>
34          <name>duração</name>
35          <value>20.0</value>
36        </property>
37        <auxiliary>
38          <name>Produção</name>
39          <expression>
40            <m:math>
41              <m:cn>1.0</m:cn>
42            </m:math>
43          </expression>
44        </auxiliary>
45      </items>
46    </class>
47    <class>
48      <name>Desenvolvedor</name>
49      <items>
50        <property>
51          <name>experiência</name>
52          <value>1.0</value>
53        </property>
54        <auxiliary>
55          <name>Produtividade</name>
56          <expression>
57            <m:math>
58              <m:cn>1.0</m:cn>
59            </m:math>
```

```
60              </expression>
61            </auxiliary>
62          </items>
63        </class>
64      </classes>
65      <relations>
66        <singleRelation>
67          <name>Equipe</name>
68          <source>Atividade</source>
69          <target>Desenvolvedor</target>
70        </singleRelation>
71        <multiRelation>
72          <name>Precedente</name>
73          <source>Atividade</source>
74          <target>Atividade</target>
75        </multiRelation>
76      </relations>
77  </metaModel>
```

Listagem C.2: Modelo de domínio em XML.

Os modelos de instância são criados a partir das regras definidas em um modelo de domínio. O código da listagem C.3 cria um modelo de instância contendo dois desenvolvedores e duas atividades. Ele também define os relacionamentos estruturais entre as instâncias criadas.

```
1  // Modelo de Instância
2  MetaModelInstance instanceModel = new DefaultMetaModelInstance();
3  instanceModel.setMetaModel(domainModel);
4  instanceModel.setName("Instância de Projeto com Cenários");
5
6  instanceModel.addNewClassInstance("D1", "Desenvolvedor");
7  instanceModel.addNewClassInstance("D2", "Desenvolvedor");
8
9  instanceModel.getClassInstances().get("D1").setProperty(
10     "experiência", 1.0);
11 instanceModel.getClassInstances().get("D2").setProperty(
12     "experiência", 0.6);
13
14 instanceModel.addNewClassInstance("Projeto", "Atividade");
15 instanceModel.addNewClassInstance("Codificação", "Atividade");
16
17 ClassInstance design = instanceModel.getClassInstances().get(
```

```
18      "Projeto");
19  design.setProperty("duração", 20.0);
20  design.setLink("Equipe", "D1");
21
22  ClassInstance coding = instanceModel.getClassInstances().get(
23      "Codificação");
24  coding.setProperty("duração", 15.0);
25  coding.setLink("Precedente", "Projeto");
26  coding.setLink("Equipe", "D2");
```

Listagem C.3: Modelo de instância construído programaticamente.

As instâncias da classe Desenvolvedor são criadas nas linhas 6 e 7 e as da classe Atividade nas linhas 14 e 15. Os valores das propriedades de cada instância de desenvolvedor são alteradas nas linhas 9 a 12. O mesmo é feito para a propriedade duração das instâncias de atividades nas linhas 19 e 24. Os relacionamentos entre instâncias são feitos a partir das atividades nas linhas 20, 25 e 26.

Este modelo de instância também pode ser criado via arquivo XML, com extensão ".jymmi". O código da listagem C.4 apresenta o conteúdo de um arquivo contendo o modelo de instância descrito anteriormente.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <metaModelInstance xmlns:m="http://www.w3.org/1998/Math/MathML">
3    <name>Projeto em Cascata</name>
4    <metaModelFile>Projeto de Software Simples.jymm</metaModelFile>
5    <classInstances>
6      <classInstance>
7        <name>Projeto</name>
8        <class>Atividade</class>
9        <items>
10         <property>
11           <name>duração</name>
12           <value>20.0</value>
13         </property>
14         <singleRelation>
15           <name>Equipe</name>
16           <target>D1</target>
17         </singleRelation>
18       </items>
19     </classInstance>
20     <classInstance>
```

```
21        <name>Codificação</name>
22        <class>Atividade</class>
23        <items>
24          <property>
25            <name>duração</name>
26            <value>15.0</value>
27          </property>
28          <singleRelation>
29            <name>Equipe</name>
30            <target>D2</target>
31          </singleRelation>
32          <multiRelation>
33            <name>Precedente</name>
34            <targets>
35              <target>Projeto</target>
36            </targets>
37          </multiRelation>
38        </items>
39      </classInstance>
40      <classInstance>
41        <name>D1</name>
42        <class>Desenvolvedor</class>
43        <items>
44          <property>
45            <name>experiência</name>
46            <value>1.0</value>
47          </property>
48        </items>
49      </classInstance>
50      <classInstance>
51        <name>D2</name>
52        <class>Desenvolvedor</class>
53        <items>
54          <property>
55            <name>experiência</name>
56            <value>0.6</value>
57          </property>
58        </items>
59      </classInstance>
60    </classInstances>
61    <scenarios></scenarios>
62  </metaModelInstance>
```

Listagem C.4: Modelo de instância em XML.

Os modelos de cenários descrevem alterações estruturais em instâncias de classes. Estes podem ser criados programaticamente de acordo com o código da listagem C.5.

```
1  MetaModelScenario sceActTeam = new DefaultMetaModelScenario();
2  sceActTeam.setName("Produção Baseada na Equipe");
3  sceActTeam.setMetaModel(domainModel);
4
5  MetaModelScenarioConnection connProdByTeam = new
       DefaultMetaModelScenarioConnection();
6  connProdByTeam.setName("AAtividade");
7  connProdByTeam.setClassName("Atividade");
8  MetaModelScenarioAffect affEquipe = new DefaultMetaModelScenarioAffect();
9  affEquipe.setName("Produção");
10 affEquipe.setExpression(new DefaultNameExpression(
11      "Equipe.Produtividade"));
12 connProdByTeam.getAffectList().add(affEquipe);
13 sceActTeam.put(connProdByTeam);
14
15 domainModel.putScenario(sceActTeam);
16
17
18 MetaModelScenario sceActPreced = new DefaultMetaModelScenario();
19 sceActPreced.setName("Precedência de Atividades");
20 sceActPreced.setMetaModel(domainModel);
21
22 MetaModelScenarioConnection connTaskPred = new
       DefaultMetaModelScenarioConnection();
23 connTaskPred.setName("AAtividade");
24 connTaskPred.setClassName("Atividade");
25 MetaModelScenarioAffect affProduction = new DefaultMetaModelScenarioAffect
       ();
26 affProduction.setName("Trabalho");
27 Expression expIf = new DefaultExpression(NumberOperator.IF);
28 Expression expGt = new DefaultExpression(BooleanOperator.LOWERTHAN);
29 Expression expGroup = new DefaultExpression(NumberOperator.GROUPSUM);
30 expGroup.setLeftOperand(new DefaultNameExpression("Precedente"));
31 expGroup.setRightOperand(new DefaultNameExpression("Tempo para Concluir"));
32 expGt.setLeftOperand(expGroup);
33 expGt.setRightOperand(new DefaultNumberConstantExpression(0.041));
```

```
34 expIf.setLeftOperand(expGt);
35 expIf.setMiddleOperand(new DefaultNameExpression("Trabalho"));
36 expIf.setRightOperand(new DefaultNumberConstantExpression(0.0));
37 affProduction.setExpression(expIf);
38 connTaskPred.getAffectList().add(affProduction);
39 sceActPreced.put(connTaskPred);
40
41 domainModel.putScenario(sceActPreced);
42
43 coding.setScenarioConnection("Produção Baseada na Equipe",
44     "AAtividade");
45 design.setScenarioConnection("Produção Baseada na Equipe",
46     "AAtividade");
47 coding.setScenarioConnection("Precedência de Atividades",
48     "AAtividade");
49 design.setScenarioConnection("Precedência de Atividades",
50     "AAtividade");
```

Listagem C.5: Modelos de cenários construídos programaticamente.

As linhas de 1 a 15 criam o cenário que associa o trabalho realizado em uma atividade à produtividade do desenvolvedor associado a ela. O cenário que limita a execução de uma tarefa à conclusão das tarefas anteriores é descrito nas linhas 18 a 41.

Os modelos de cenários, da mesma forma que os anteriores, também podem ser representados em arquivos XML. Estes são guardados, isoladamente, em arquivos usando a extensão ".jymms". As listagens C.6 e C.7 apresentam o conteúdo destes arquivos.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <scenario xmlns:m="http://www.w3.org/1998/Math/MathML">
3    <name>çãProduo Baseada na Equipe</name>
4    <metaModelName>Projeto de Software Simples</metaModelName>
5    <connections>
6      <connection>
7        <name>AAtividade</name>
8        <className>Atividade</className>
9        <classInstanceItems>
10       </classInstanceItems>
11       <affects>
12         <affect>
13           <name>Produção</name>
14           <expression>
```

```
15              <m:math>
16                <m:ci>Equipe.Produtividade</m:ci>
17              </m:math>
18            </expression>
19          </affect>
20        </affects>
21      </connection>
22    </connections>
23  </scenario>
```

Listagem C.6: Modelo de cenário de produção com base na equipe em XML.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <scenario>
3    <name>Dependência de Atividades</name>
4    <metaModelName>Projeto de Software Simples</metaModelName>
5    <connections>
6      <connection>
7        <name>AAtividade</name>
8        <className>Atividade</className>
9        <classInstanceItems>
10       </classInstanceItems>
11       <affects>
12         <affect>
13           <name>Trabalho</name>
14           <expression>
15             <m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
16               <m:apply>
17                 <m:if />
18                 <m:apply>
19                   <m:leq />
20                   <m:apply>
21                     <m:groupsum />
22                     <m:ci>Precedente</m:ci>
23                     <m:ci>Tempo para Concluir</m:ci>
24                   </m:apply>
25                   <m:cn>0.041</m:cn>
26                 </m:apply>
27                 <m:apply>
28                   <m:min />
29                   <m:ci>Produção</m:ci>
30                   <m:ci>Tempo para Concluir</m:ci>
31                 </m:apply>
32                 <m:cn>0.0</m:cn>
```

```
33                  </m:apply>
34                </m:math>
35              </expression>
36            </affect>
37          </affects>
38        </connection>
39      </connections>
40  </scenario>
```

Listagem C.7: Modelo de cenário de dependência entre atividades em XML.

As conexões dos cenários às instâncias de classe, devem ser incluídas no modelo de instância. Para os modelos descritos programaticamente, podemos realizar a conexão da forma descrita na listagem C.8.

```
coding.setScenarioConnection("Produção Baseada na Equipe",
    "AAtividade");
design.setScenarioConnection("Produção Baseada na Equipe",
    "AAtividade");
coding.setScenarioConnection("Precedência de Atividades",
    "AAtividade");
design.setScenarioConnection("Precedência de Atividades",
    "AAtividade");
```

Listagem C.8: Conexão de cenários construídos programaticamente.

Para o modelo descrito anteriormente em XML, na listagem C.4, é necessário modificar o arquivo de acordo com a listagem C.9. Neste fragmento de código carregamos os cenários e realizamos as conexões à instâncias.

```
<?xml version="1.0" encoding="UTF-8"?>
<metaModelInstance xmlns:m="http://www.w3.org/1998/Math/MathML">
  <name>Projeto em Cascata</name>
  <metaModelFile>Projeto de Software Simples.jymm</metaModelFile>
  <classInstances>
  (...)
  </classInstances>
  <scenarios>
    <files>
      <file>Produção Baseada na Equipe.jymms</file>
      <file>Dependência de Atividades.jymms</file>
```

```
12      </files>
13      <connects>
14        <connect>
15          <scenario>Produção Baseada na Equipe</scenario>
16          <name>A Atividade</name>
17          <instance>Projeto</instance>
18        </connect>
19        <connect>
20          <scenario>Produção Baseada na Equipe</scenario>
21          <name>A Atividade</name>
22          <instance>Codificação</instance>
23        </connect>
24        <connect>
25          <scenario>Dependência de Atividades</scenario>
26          <name>A Atividade</name>
27          <instance>Projeto</instance>
28        </connect>
29        <connect>
30          <scenario>Dependência de Atividades</scenario>
31          <name>A Atividade</name>
32          <instance>Codificação</instance>
33        </connect>
34      </connects>
35    </scenarios>
36  </metaModelInstance>
```

Listagem C.9: Alteração do modelo de instância em XML para incluir os cenários.