PEDRO DE ALCÂNTARA DOS SANTOS NETO

MODEST: UM MÉTODO DE TESTE BASEADO EM MODELOS

Belo Horizonte
22 de dezembro de 2006

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

MODEST: UM MÉTODO DE TESTE BASEADO EM MODELOS

Tese apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

PEDRO DE ALCÂNTARA DOS SANTOS NETO

Belo Horizonte
22 de dezembro de 2006

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

MODEST: Um Método de Teste Baseado em Modelos

PEDRO DE ALCÂNTARA DOS SANTOS NETO

Tese defendida e aprovada pela banca examinadora constituída por:

Ph. D. Rodolfo Resende – Orientador
DCC – ICEx – UFMG

Ph. D. Clarindo Pádua – Co-orientador
DCC – ICEx – UFMG

Ph. D. Antônio Otávio Fernandes
DCC – ICEx – UFMG

Ph. D. Roberto Bigonha
DCC – ICEx – UFMG

Ph. D. Paulo César Masiero
DSC – ICMC – USP/São Carlos

Ph. D. José Carlos Maldonado
DSC – ICMC – USP/São Carlos

Ph. D. Guilherme Horta Travassos
PESC – COPPE – UFRJ

Belo Horizonte, 22 de dezembro de 2006

Resumo

Neste trabalho apresentamos contribuições relacionadas à melhoria das atividades de teste, em particular, apresentamos um método de Teste Baseado em Modelos, denominado MODEST, que foi concebido para utilizar informações simples sobre o comportamento do Sistema Sob Teste. O uso do MODEST pode reduzir o custo do desenvolvimento, em função da redução do esforço exigido durante a construção do software. Embora o uso do método aumente o tempo para desenho, ele reduz o esforço exigido nas atividades de teste. Essa redução propicia uma diminuição geral no tempo de desenvolvimento, que normalmente acarreta em ganhos em termos de custos. O uso do método também pode trazer uma maior capacidade de detecção de falhas. Isso foi evidenciado em um estudo experimental, que mostrou que os testes gerados automaticamente, utilizando a ferramenta baseada no método, foram mais eficazes na detecção de falhas que os testes manualmente produzidos pelos participantes do estudo.

Durante a criação do MODEST identificamos uma série de requisitos associados ao Teste Baseado em Modelos. Agrupamos esses requisitos na forma de um catálogo e o estendemos a partir da realização de oficinas de requisitos com diversos profissionais relacionados ao desenvolvimento de software. O catálogo de requisitos desenvolvido captura as necessidades, expectativas e restrições dos trabalhadores relacionados ao desenvolvimento de software considerados. Apesar desse catálogo ter sido influenciado diretamente por nosso conhecimento prévio no desenvolvimento de sistemas de informação, ele também pode ser útil para outros domínios. Os requisitos existentes no catálogo podem ser utilizados como guia para o desenvolvimento de novos métodos e ferramentas, assim como pode servir de base para a avaliação de métodos e ferramentas para o Teste Baseado em Modelos.

Abstract

In this work we present our contributions related to the improvement of testing activities, in particular, we present a model-based testing method called MODEST. The main characteristic of MODEST is the use of simple information about the behavior of the system under test. The use of MODEST can decrease the development costs, due to the reduction of the overall effort required in the development. Although the use of the method increases the design time, it decreases the effort required in the testing activities. This reduction generates an overall reduction in the development effort. The use of the method can also improve the failure detection capability. This was evidenced in a study showing that the automatically generated tests were more effective to detect the system failures than the manually generated ones.

During the development of MODEST we identified a series of requirements related to model-based testing. We grouped these requirements in a catalog and we extended it by using several software developers. Our requirements catalog captures the needs, expectations, and constraints of the considered software development workers in a high level of abstraction. Despite the influence of our previous knowledge of information systems development in setting up the catalog, the catalog can be useful in other domains as well. The catalog can be used as a basis for improving methods as well as a guide for the development of new methods and tools.

Dedico à minha filha, Sofia Magalhães Lapa, que está pra chegar neste mundo...

Agradecimentos

- Ao professor Rodolfo Resende, professor orientador e amigo, pelos diversos ensinamentos durante o doutorado.
- Ao professor Clarindo Pádua, que muito contribuiu para a realização deste trabalho através de inúmeras sugestões.
- Aos professores Paulo César Masiero, José Carlos Maldonado, Guilherme Travassos, Roberto Bigonha e Antônio Otávio, que prontamente atenderam ao pedido para serem membros da banca examinadora.
- Ao professor Guilherme Travassos, e seu aluno de doutorado Arilo Cláudio, por diversas sugestões relacionadas ao estudo experimental realizado neste trabalho.
- À minha esposa, Aracelly Magalhães, pelo amor, compreensão e paciência durante esse período.
- Aos meus pais, J. J. Lapa e Teresa Maria, pelo grande incentivo e carinho demonstrado durante toda minha vida.
- A todos meus familiares que não puderam contar comigo durante esse período, mas que sempre torceram por mim, em especial aos donos da Casa da Felicidade em Luís Correia - PI.
- Aos integrantes turma de pós-graduação de 2002, que tornaram a estadia em uma cidade desconhecida algo de grandes recordações.
- À Universidade Federal de Minas Gerais, nas pessoas dos professores e funcionários do DCC, que com seus ensinamentos, direta ou indiretamente, ajudaram-me a atingir este objetivo.
- À Universidade Federal do Piauí, por ter me liberado para realização do curso, em especial aos professores Francisco Vieira e Raimundo Moura, que muito me auxiliaram na resolução de questões burocráticas.
- Ao Synergia, pela oportunidade de vivenciar, na prática, várias das questões que procurei abordar - uma prova de que projetos de extensão produzem, sim, bons resultados de pesquisa para a Universidade.

- A todos que fizeram deste trabalho uma experiência válida.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este trabalho está diretamente relacionado à automação das atividades de teste de software. O teste é uma das atividades de garantia da qualidade. Segundo Pressman (2006), qualidade está associada à conformidade a requisitos funcionais e de desempenho, padrões e convenções de desenvolvimento pré-estabelecidos, e atributos implícitos que todo software desenvolvido profissionalmente deve possuir. A qualidade é um importante aspecto do software, assim como de qualquer outro produto.

As técnicas para Validação e Verificação (V&V) são algumas das atividades relacionadas à garantia da qualidade. Embora essas atividades sejam consideradas importantes por quase todas as organizações, boa parte delas ainda têm dúvidas sobre executar ou não tais atividades, dentre outros fatores, em função de uma percepção de altos custos (Pressman, 2006).

Quanto mais cedo um defeito for descoberto, menor serão os custos para corrigir os erros associados[1]. A utilização de atividades de garantia da qualidade no desenvolvimento de software aumenta os custos devido a introdução de atividades adicionais no desenvolvimento, mas propicia a descoberta de erros mais cedo, o que normalmente propicia uma diminuição geral de custos.

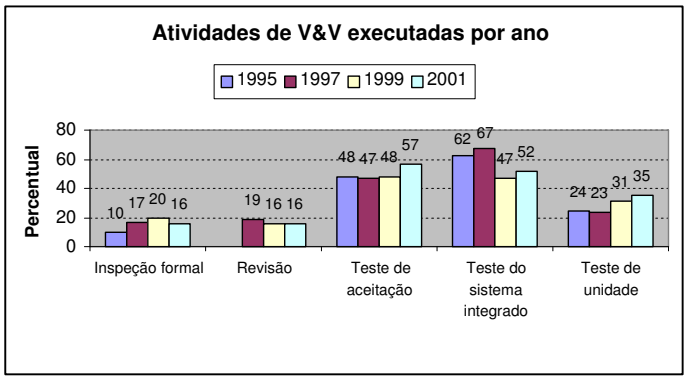

Figura 1.1: Atividades de V&V executadas pelas organizações brasileiras desenvolvedoras de software (MCT, 2004).

[1]Discutimos brevemente na Seção 2.2 a terminologia adotada neste trabalho.

Embora a exigência por produtos com maior qualidade seja cada vez maior, muitas organizações desenvolvedoras de software ainda não executam essas atividades. A Figura 1.1 mostra as atividades de V&V executadas pelas organizações brasileiras. Esses dados foram extraídos do Programa Brasileiro da Qualidade e Produtividade em Software (MCT, 2004). Conforme exibido na figura, ainda existem diversas organizações brasileiras que não utilizam técnicas de V&V durante o desenvolvimento de software.

Embora a inspeção (revisão técnica) seja uma das técnicas mais eficazes na detecção e remoção de defeitos (Basili et al., 1986; Lott e Rombach, 1996), o teste é importante para complementá-la e aferir o nível de qualidade conseguido. A realização de testes é limitada por restrições de cronograma e orçamento, uma vez que esses fatores determinam quantos testes poderão ser executados. É importante que os testes sejam bem planejados e desenhados para conseguir o melhor proveito possível dos recursos alocados para eles (Binder, 2000).

O *teste de software* é a verificação *dinâmica* do funcionamento de um programa utilizando um *conjunto finito* de casos de teste, adequadamente *escolhido* dentro de um domínio de execução, em geral, infinito, contra seu *comportamento esperado* (IEEE, 2004). Nesse conceito existem alguns elementos-chave para este trabalho:

- dinâmica: o teste exige a execução do produto, embora algumas de suas atividades sejam realizadas antes do produto estar operacional;
- conjunto finito: o teste exaustivo é geralmente impraticável, mesmo para produtos simples;
- escolhido: é necessário selecionar testes com alta probabilidade de encontrar falhas, preferencialmente com o mínimo de esforço;
- comportamento esperado: para que um teste possa ser executado é necessário saber o comportamento esperado, para que ele possa ser comparado com o obtido.

1.1 Motivação

Conforme mencionado na seção anterior, muitas organizações brasileiras ainda não executam atividades de V&V, em particular as atividades de teste. Isso ocorre porque uma parte significativa dessas atividades não é automatizada. Embora exista uma grande quantidade de ferramentas para auxiliar nessa tarefa, ainda existem importantes oportunidades de automação.

Para se ter uma idéia dos custo envolvidos, de acordo com um relatório publicado pelo NIST (2002), U$59.500.000.000,00 foi o valor relativo ao custo de faltas em softwares desenvolvidos nos Estados Unidos, apenas em 2002. Esse mesmo relatório estima que mais de 37% desse custo (U$22.200.000.000,00) poderia ter sido eliminado se a infra-estrutura para testes fosse melhorada. Essa economia aconteceria em virtude da descoberta de erros nos estágios iniciais do desenvolvimento.

Embora os testes tenham um importante papel na aferição da qualidade de um produto e sua execução de forma incompleta ou inadequada possa causar muitos problemas, eles ainda são deixados de lado por algumas organizações, conforme já frisado na Figura 1.1. Freqüentemente, atrasos nas atividades de desenvolvimento são compensados reduzindo ou eliminando as atividades de teste (Aranha e Borba, 2002).

As metodologias ágeis de desenvolvimento, embora muito recentes, enfatizam bastante a necessidade das atividades de teste dentro de um processo de software. Nessas metodologias, os testes constituem uma prática de programação chave para o sucesso de seu uso (Beck, 1999). Sem a sua correta utilização, torna-se difícil obter resultados favoráveis. De uma forma resumida, a atividade de teste permite a realização de melhorias no processo utilizado, ao mesmo tempo em que auxilia a garantia de qualidade do produto desenvolvido. Isso remete a uma interessante constatação: idealmente, o aperfeiçoamento das atividades de teste, em termos de maior automação, melhor integração e aproveitamento das práticas de desenvolvimento de software, deve contribuir não apenas para a melhoria da qualidade dos produtos, mas também para propiciar condições que garantam que as organizações melhorem seus processos e os instanciem de forma correta.

Embora as atividades de teste ainda não sejam utilizadas por todas as empresas desenvolvedoras, seu uso vem crescendo nos últimos anos, assim como a utilização de modelos no desenvolvimento de software. Um modelo é uma simplificação da realidade, permitindo uma validação ou proposição de teorias, com riscos e custos reduzidos. Utilizando modelos podemos entender melhor os sistemas sendo desenvolvidos. Embora haja processos ágeis com casos de sucesso, é amplamente aceito que, quanto maior e mais complexo o sistema, mais importante a modelagem se torna, uma vez que um dos motivos para a utilização de modelos ocorre por causa da dificuldade em se compreender um sistema grande e complexo (Booch et al., 1999).

Esses aspectos foram os principais motivadores para este trabalho, que podem ser resumidos em: (i) existência de uma grande quantidade de organizações que não realizam testes; (ii) alto custo das atividades de teste; (iii) baixa integração do processo de software com as atividades de teste; e (iv) crescente utilização de modelos no desenvolvimento de software.

1.2 Descrição do trabalho

Neste trabalho discutimos como melhorar a eficiência das atividades de teste utilizando modelos descrevendo o software, ou seja, utilizando o Teste Baseado em Modelo (TBM) (Binder, 2000). Particularmente, nossa pesquisa é centrada nos potenciais ganhos por meio do reuso de modelos similares aos prescritos pelo Processo Unificado (PU) (Jacobson et al., 1999), utilizando a UML (Rumbaugh et al., 1999). A escolha do PU e da UML como especificações de processo e linguagem para nossa pesquisa se deu por uma série de fatores, dentre eles o fato desse processo e linguagem serem bastante disseminados na indústria de software (Kobryn, 1999), e também por termos experiência no seu uso em projetos industriais e acadêmicos. Outros fatores relacionados a essa escolha serão apresentados à frente.

Neste trabalho, desenvolvemos um método de TBM, denominado MODEST (Santos-Neto e Resende,

2004; Santos-Neto et al., 2005b), juntamente com uma implementação do método no contexto da UML e do PU. Avaliamos o MODEST no contexto de uma disciplina de Engenharia de Software em um curso de Ciência da Computação. Essa avaliação mostrou que o uso do método pode trazer ganhos em termos de qualidade dos testes gerados e redução nos custos associados (Santos-Neto et al., 2006).

Embora o MODEST tenha sido inicialmente desenvolvido focalizando a UML e o PU, notamos que ele poderia ser generalizado e descrito de forma mais independente da linguagem de modelagem e do processo utilizado, mas antes dessa generalização, mostramos sua utilização no contexto de um processo concreto baseado no PU (Santos-Neto et al., 2005c). A partir dessa constatação passamos a descrever o método utilizando diferentes níveis de abstração. Neste trabalho apresentamos o método utilizando três níveis: conceitual, lógico e físico.

Ao longo deste trabalho fizemos uma pesquisa bibliográfica sobre o uso de modelos para auxílio às atividades de teste. Essa pesquisa indicou que a maioria dos trabalhos analisados propõe soluções para problemas específicos. Reunimos esses aspectos em um catálogo preliminar de requisitos para o TBM (Santos-Neto et al., 2005a). Depois disso realizamos algumas reuniões com alguns profissionais da área de desenvolvimento e expandimos o catálogo (Santos-Neto et al., 2007). Os requisitos existentes no catálogo podem ser utilizados como guia para o desenvolvimento de novos métodos e ferramentas, além de servir como mecanismo de avaliação de ferramentas e métodos para o TBM.

1.3 Limites do trabalho

Tão importante quanto descrever a motivação e os objetivos do trabalho é esclarecer os limites do mesmo, visando delimitar mais precisamente o escopo que está sendo tratado.

Em primeiro lugar, é importante ressaltar que embora possa ser desenvolvida uma implementação do MODEST utilizando mecanismos mais formais, a implementação da ferramenta de suporte ao método apresentada neste trabalho foi feita no contexto do PU e da UML, que é considerada uma linguagem semi-formal (Booch et al., 1999). Conforme discutido no próximo capítulo, utilizamos neste trabalho um processo concreto baseado no PU denominado PRAXIS (Filho, 2003). O PRAXIS utiliza aspectos das versões 1.x da UML e no tempo da elaboração deste trabalho não foram utilizados recursos das versões mais recentes da UML, tal como a UML 2.0.

Está fora do escopo deste trabalho a discussão de aspectos relacionados a decidibilidade e complexidade das técnicas adotadas na automação de teste. Embora esses aspectos sejam vitais na discussão de qualquer trabalho na área de informática, decidimos que utilizaríamos os resultados presentes na literatura, não colocando isso como foco da atividade de pesquisa. Por exemplo, a seguir discutimos a linguagem que foi adotada para expressar restrições em modelos. Conforme discutido abaixo, optamos por adotar uma linguagem baseada em um subconjunto de uma outra linguagem para a qual existem estudos sobre aspectos de decidibilidade e complexidade (Papadimitriou, 1993).

Como os modelos gráficos, baseados na UML, podem não apresentar a semântica necessária para a automação de testes, existe a necessidade de descrever restrições adicionais sobre os elementos dos modelos. Por exemplo, a linguagem natural não deveria ser utilizada nas descrições devido a possibilidade de especificação de restrições ambíguas. Neste caso, em particular, utilizamos uma linguagem simples para especificação de restrições nos modelos. Essa linguagem foi inspirada na Linguagem de Modelagem de Restrições (OCL - Object Constraint Language) (Warmer e Kleppe, 2003). Assim como a OCL, a linguagem utilizada em nossos modelos é livre de efeitos colaterais, ou seja, o estado do sistema modelado nunca muda por causa da avaliação de uma expressão descrita na linguagem. Além disso, todas as expressões avaliadas sempre resultam em um valor. Essas características reduzem o problema relacionado à interpretação da linguagem, tornando factível essa tarefa.

Durante o trabalho focalizamos no desenvolvimento de sistemas de informação, uma vez que possuímos experiência no desenvolvimento desse tipo de sistemas. Mais especificamente, focalizamos nos sistemas de informação categorizados como Sistemas de Processamento de Transação (SPT) (Cook, 1996a). Conforme discutido na Seção 2.8, embora esse tipo de sistema seja considerado o mais simples dos SI, ele serve de base para a construção dos demais sistemas de informação. Um outro fator limitante é que nosso método assume que a concepção do sistema de informação utilize uma camada de persistência como ponte entre o sistema e o mecanismo de armazenamento (Ambler, 1998).

Esta tese organiza-se da seguinte forma: o Capítulo 2 apresenta os principais conceitos envolvidos, incluindo ainda uma discussão dos trabalhos relacionados e dos requisitos para métodos de TBM; o Capítulo 3 detalha o MODEST, a integração do processo PRAXIS com o método e uma análise comparativa de três métodos de TBM, incluindo o próprio MODEST, utilizando como base o catálogo de requisitos para o TBM; o Capítulo 4 apresenta um estudo experimental realizado com intuito de caracterizar o uso do método; e o Capítulo 5 conclui o trabalho e apresenta direções para trabalhos futuros. Os apêndices A, B, C, apresentam as listas de conferência para revisão dos modelos considerados no estudo experimental realizado e um questionário de avaliação dos participantes. O Apêndice D apresenta a lista de abreviaturas e acrônimos utilizados no trabalho.

Capítulo 2

Referencial teórico

Neste capítulo apresentamos alguns referenciais teóricos associados ao trabalho. Iniciamos com a apresentação do processo de software utilizado como referência para criação do MODEST e também utilizado durante sua avaliação. Em seguida apresentamos alguns conceitos básicos de teste de software, apresentando também alguns níveis, objetivos e critérios de teste definidos pelo IEEE (2004). Apresentamos também os principais documentos associados à execução das atividades de teste. Em seguida, apresentamos de forma breve a UML (Rumbaugh et al., 1999) e um método para determinar e classificar necessidades e desejos de clientes. Apresentamos ainda alguns trabalhos relacionados, assim como um catálogo de requisitos associados ao TBM. Finalizando fazemos uma definição mais precisa do trabalho utilizando todos os termos apresentados neste capítulo.

2.1 O processo utilizado

Conforme mencionado no capítulo anterior, utilizamos o PU como referência para desenvolvimento do MODEST. No entanto, devido à necessidade de realização de um estudo experimental com o intuito de caracterizar o método, tivemos que desenvolver uma personalização do MODEST para um processo concreto, baseado no PU, facilitando assim seu uso em um ambiente experimental. Decidimos então utilizar o processo PRAXIS (Filho, 2003). O PRAXIS é um processo baseado no PU e foi escolhido por uma série de fatores: (i) ele é um processo desenhado para dar suporte a projetos didáticos, em disciplinas de Engenharia de Software; (ii) ele foi desenvolvido por um professor da UFMG e é muito utilizado nessa instituição, seja para fins acadêmicos ou industriais; (iii) possuímos um nível de experiência razoável na utilização do processo, tanto no ambiente acadêmico quanto no ambiente industrial; e (iv) o PRAXIS especifica algumas atividades diretamente relacionadas ao MODEST de uma forma mais detalhada que o PU, fato este que facilita sua personalização para uso em conjunto com o MODEST.

O PRAXIS utiliza a UML em praticamente todos seus modelos, em particular nos modelos de análise e desenho. O PRAXIS é dividido em fases e fluxos. As fases são divisões maiores do processo, para fins gerenciais, que correspondem aos pontos principais de aceitação por

Fase	Iteração	Descrição
Concepção	Ativação	Levantamento e análise das necessidades dos usuários e conceitos da aplicação, em nível de detalhe suficiente para justificar a especificação de um produto de software.
Elaboração	Levantamento de Requisitos	Levantamento das funções, interfaces e requisitos não-funcionais desejados para o produto.
	Análise de Requisitos	Modelagem conceitual dos elementos relevantes do domínio do problema e uso desse modelo para validação dos requisitos e planejamento detalhado da fase de Construção.
	Desenho Implementável	Definições interna e externa dos componentes de um produto de software, em nível suficiente para decidir as principais questões de arquitetura e tecnologia e para permitir o planejamento detalhado das liberações.
Construção	Liberação 1	Implementação de um subconjunto de funções do produto que será avaliado pelos usuários.
	Liberação ...	Idem
	Testes Alfa	Realização dos testes de aceitação, no ambiente dos desenvolvedores, juntamente com elaboração da documentação de usuário e possíveis planos de Transição.
Transição	Testes Beta	Realização dos testes de aceitação no ambiente dos usuários.
	Operação Piloto	Operação experimental do produto em instalação piloto do cliente, com a resolução de eventuais problemas através de processo de manutenção.

Tabela 2.1: As Fases do PRAXIS (Filho, 2003).

Natureza	Fluxo	Descrição
Técnicos	Requisitos	Fluxo que visa a obter um conjunto de requisitos de um produto, acordado entre cliente e fornecedor.
	Análise	Fluxo que visa a detalhar, estruturar e validar os requisitos de um produto, em termos de um modelo conceitual do problema, de forma que eles possam ser usados como base para o planejamento e controle detalhados do respectivo projeto de desenvolvimento.
	Desenho	Fluxo que visa a formular um modelo estrutural do produto que sirva de base para a implementação, definindo os componentes a desenvolver e a reutilizar, assim como as interfaces entre si e com o contexto do produto.
	Implementação	Fluxo que visa a detalhar e implantar o desenho através de componentes de código e de documentação associada.
	Testes	Fluxo que visa a verificar os resultados da implementação, através do planejamento, desenho e realização de baterias de testes.
	Engenharia de sistemas	Fluxo que abrange atividades relativas ao desenvolvimento do sistema no qual o produto de software está contido; por exemplo, modelagem de processos de negócio, implantação, usabilidade e criação de conteúdo.
Gerenciais	Gestão de projetos	Fluxo que visa a planejar e controlar os projetos de software.
	Gestão da qualidade	Fluxo que visa a verificar e assegurar a qualidade dos produtos e processos de software.
	Engenharia de processos	Fluxo que visa a manter, dar suporte e promover melhorias nos próprios processos de software.

Tabela 2.2: Os Fluxos do PRAXIS (Filho, 2003).

parte do cliente. As fases existentes no PRAXIS são apresentadas na Tabela 2.1. Os fluxos correspondem a subprocessos caracterizados por um tema técnico ou gerencial. Os fluxos existentes no PRAXIS são apresentados na Tabela 2.2.

A divisão de fases do PRAXIS obedece ao modelo de ciclo de vida de entrega evolutiva (Royce, 1970; Mills, 1971). Na fase de construção é desenvolvida (desenhada, implementada e testada) uma liberação completamente operacional do produto, que atende aos requisitos especificados. Dentro dessa fase são implementadas uma ou mais liberações, que correspondem a um subconjunto de funções do produto, submetido à avaliação do cliente.

A documentação do PRAXIS inclui um exemplo de desenvolvimento de um software, denominado Merci. Ele foi construído utilizando Java e possui sua modelagem descrita em UML, seguindo as convenções existentes no PRAXIS. O Merci tem como missão o apoio informa-

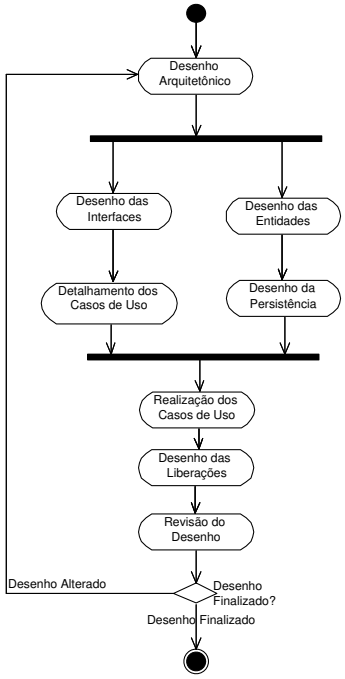

Figura 2.1: O Fluxo de Desenho (Filho, 2003).

tizado ao controle de vendas, de compras, de fornecedores e de estoque de uma mercearia. Neste trabalho, utilizamos o Merci para apresentar alguns exemplos de aplicação do método, além de termos utilizado parte dele no estudo experimental que avaliou o MODEST. Um detalhamento mais aprofundado sobre ele pode ser encontrado no sítio de apoio ao PRAXIS[1]. A seguir descrevemos brevemente os fluxos técnicos de desenho e teste do PRAXIS padrão, que são os fluxos diretamente associados a este trabalho.

2.1.1 O Fluxo de Desenho

O Fluxo de Desenho tem por objetivo definir uma estrutura implementável para um produto de software que atenda aos requisitos especificados. Esse fluxo é apresentado na Figura 2.1. Ele inicia pela atividade de Desenho arquitetônico, que trata de aspectos estratégicos de desenho. Essa atividade é responsável pela divisão do produto em subsistemas e pela escolha das tecnologias mais adequadas.

O Desenho das interfaces trata do desenho das interfaces definitivas do produto em seu ambiente de implementação. Os campos e comandos das interfaces de usuário (GUI - Graphical User Interface) são definidos, juntamente com um diagrama de estados que descreve as habilitações em cada estado da interface. Além disso, deve ser criado um diagrama que apresente a navegação (transferência de controle) entre os objetos da interface de usuário.

[1] http://www.wppf.uaivip.com.br/praxis/

O Detalhamento dos casos de uso resolve os detalhes dos fluxos dos casos de uso de desenho, considerando os componentes reais das interfaces. Todos os fluxos alternativos devem ser detalhados, inclusive os que consistem de exibição de mensagens de exceção ou erro.

A atividade de Desenho das entidades transforma as classes de entidade do Modelo de Análise nas classes correspondentes do Modelo de Desenho.

O Desenho da persistência trata da definição de estruturas externas de armazenamento persistente, como arquivos e bancos de dados.

A Realização dos casos de uso determina como os objetos das classes de desenho colaborarão para realizar os casos de uso. As colaborações representam interações entre objetos de classes correlatas, representadas por meio de diagramas de classes, seqüência e colaboração. Geralmente é criado um diagrama de interação para cada roteiro de um caso de uso. Os roteiros correspondem à tradução dos fluxos de um caso de uso em termos de interações entre objetos das classes envolvidas.

O Desenho das liberações determina como a construção do produto será dividida em liberações. Finalmente, a Revisão do desenho valida o esforço de Desenho, confrontando-o com os resultados dos Requisitos e da Análise. Ela também averigua se os artefatos construídos seguem as normas constantes na organização.

2.1.2 O Fluxo de Teste

O Fluxo de Teste (Figura 2.2) tem dois grandes grupos de atividades para cada conjunto de testes (bateria) a ser desenvolvida: preparação e realização. O plano da bateria é elaborado durante a preparação, juntamente com o desenho das especificações de cada teste. Durante a realização, os testes são executados, os erros encontrados são, se possível, corrigidos e os relatórios de teste são redigidos. A preparação e realização de cada bateria correspondem a uma execução completa do Fluxo de Teste.

O grupo de preparação compreende as atividades de Planejamento, que produz o plano de testes da bateria, e Desenho, que produz as respectivas especificações. O grupo de realização é formado pelas demais atividades: Implementação, que monta o ambiente de testes; Execução, que os executa efetivamente, produzindo os principais relatórios de testes; Verificação do término, que determina se a bateria pode ser considerada como completa; e Balanço final, que analisa os resultados da bateria, produzindo um relatório de resumo.

2.2 Conceitos básicos

Nesta seção apresentamos alguns dos principais conceitos básicos utilizados neste trabalho. A descrição desses conceitos foi baseada no Glossário de Terminologia de Engenharia de Software do IEEE (1990) e no Guia do Corpo de Conhecimento em Engenharia de Software (IEEE, 2004).

Um *caso de teste* especifica como deve ser testado uma parte do software. Essa especificação inclui as entradas, saídas esperadas, e condições sob as quais o teste deve ocorrer. Um exemplo para um caso de teste para um login inválido é apresentado na Figura 2.3. Um

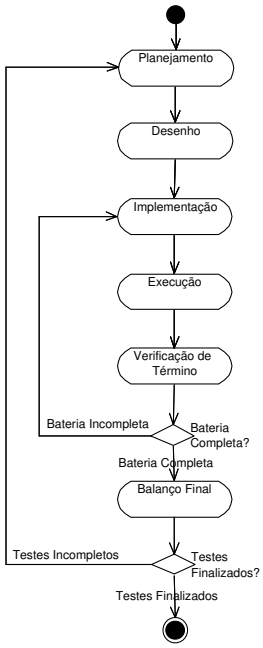

Figura 2.2: O Fluxo de Teste (Filho, 2003).

Identificação	Login inválido com caracteres não permitidos	
Itens a testar	Verificar se a tentativa de login utilizando um identificador inválido, contendo caracteres não permitidos, exibe a mensagem apropriada.	
Entradas	**Campo**	**Valor**
	Login	Pasn!
	Senha	aaaa
Saídas esperadas	**Campo**	**Valor**
	Login	Pasn!
	Senha	aaaa (exibida com *'s)
	Mensagem	O campo login deve ter no mínimo 2 e no máximo 8 caracteres alfabéticos.
Estado alcançado	SEM USUARIO	
Ambiente	Banco de dados de teste.	
Procedimentos	Procedimento de Teste Login	
Dependências	Não aplicável.	

Figura 2.3: Caso de teste para login inválido.

Identificação	Procedimento de Teste Login
Objetivo	Efetuar o login de um usuário no Merci.
Requisitos especiais	A TelaPrincipal deve estar no estado SEM USUARIO.
Fluxo	1. Preencher o campo login e senha 2. Pressionar o botão login.

Figura 2.4: Procedimento de Teste Login.

procedimento de teste detalha as ações a serem executadas em um determinado caso de teste. A Figura 2.4 exibe o Procedimento de Teste de Login. Observe que um caso de teste só faz sentido se for especificado o(s) procedimento(s) de teste associado(s). Sem essa associação não é possível determinar o que deve ser feito com as entradas especificadas no caso de teste. Observe também que o procedimento de teste é independente dos dados utilizados nos casos de teste (Filho, 2003).

A *rastreabilidade* é o grau de relacionamento que pode ser estabelecido entre dois ou mais produtos do processo de desenvolvimento.

Um *oráculo* é qualquer agente (humano ou não) que decide se um software funcionou corretamente em um determinado teste. A geração de um oráculo é provavelmente a parte mais difícil e cara da automação de testes (Binder, 2000).

Um *critério de teste* define quais propriedades de um programa devem ser exercitadas para constituir um teste completo. Muitas vezes esse termo é utilizado como sinônimo para *técnica de teste*, uma vez que critérios podem ser utilizados para selecionar os casos de teste. Na intenção de sermos mais consistentes neste trabalho utilizaremos sempre o termo critério ao invés de técnica. Critérios também podem ser utilizados para decidir se o teste pode ou não ser considerado completo e, por conseqüência, finalizado (Goodenough e Gerhart, 1975).

Alguns termos relacionados ao mau funcionamento em Engenharia de Software geram problemas na inteligibilidade de textos. Dentre esses termos destacamos: falta e falha, erro e defeito. A distinção entre falta e falha é simples: (i) falta é um problema bem determinado em um produto de software, (ii) falha é a conseqüência de uma ou mais faltas e (iii) a falha é a percepção de que o produto não comportou-se como o esperado. Portanto, observar que um software falhou é um nível de percepção externa ao software, enquanto a falta é a identificação interna, no software, de um problema. Na área de Validação e Verificação utiliza-se erro para denotar qualquer diferença entre um valor obtido, independente da sua forma de obtenção, e o valor teoricamente correto, assim como um engano que resulta em uma incorreção no software. Um defeito denota a percepção externa de que o artefato não se comportou como o esperado. Isso inclui problemas não somente no código mas também no projeto e nos requisitos. Defeitos podem levar a falhas.

2.3 Níveis e objetivos de teste

Existem dois conceitos de teste ortogonais que muitas vezes são considerados correlatos: o alvo do teste (target) e seu objetivo (objective) (IEEE, 2004). O alvo do teste apresenta a subdivisão geralmente utilizada em produtos de software complexos. O alvo de um teste pode ter diferentes níveis: pode ser um módulo único, correspondendo ao teste de unidade, um agrupamento de módulos, correspondendo ao teste de integração, ou o sistema completo, correspondendo ao teste de sistema.

O objetivo do teste está associado às condições e propriedades específicas do Software Sob Teste (SST). Testes podem ser criados para verificar se as especificações funcionais estão corretamente implementadas (teste funcional), podendo ou não haver a participação dos desenvolvedores (teste de aceitação); testes podem verificar se o desempenho do software está dentro do aceitável (teste de desempenho e estresse); se o software é adequado ao uso (teste de usabilidade); testes podem ter como objetivo mostrar que o software não regrediu (teste de regressão); se os procedimentos de instalação são corretos e bem documentados (teste de instalação); qual o seu nível de segurança (teste de segurança); ou para verificar seu funcionamento, via a liberação do software para pequenos grupos de usuários trabalhando em um ambiente controlado (teste alfa) ou em um ambiente não controlado (teste beta).

2.4 Critérios de teste

Um dos objetivos do teste é revelar falhas e para isso muitos critérios foram desenvolvidos. O princípio básico por trás de um critério é ser sistemático, a ponto de identificar um conjunto representativo de comportamentos dos programas.

Esses critérios são freqüentemente classificados como caixa branca ou caixa preta. Critérios de caixa branca têm por objetivo encontrar falhas na estrutura interna do produto, por meio de testes que exercitem suficientemente os possíveis caminhos de execução. Esses testes também são conhecidos como teste estrutural ou teste baseado no código. Os critérios de teste caixa preta têm por objetivo determinar se os requisitos foram totais ou parcialmente satisfeitos pelo produto. Esses critérios também são conhecidos como critérios funcionais ou critérios baseados na especificação.

Existem diversas classificações para os critérios existentes. Neste trabalho utilizamos a classificação proposta pelo IEEE (2004), nos atendo a alguns critérios baseados na especificação, e critérios baseados em erros. Esses critérios estão associados a este trabalho e serão discutidos onde forem necessários.

2.4.1 Critérios baseados na especificação

Os dois principais critérios de teste baseados na especificação e associados a este trabalho são: o particionamento de equivalência e a análise de valor limite.

1. if (x > 0) 2. doThis(); 3. if (x > 10) 4. doThat();	1. if (x < 0) 2. doThis(); 3. if (x > 10) 4. doThat();
1. if (x > 0) 2. doThis(); 3. if (x < 10) 4. doThat();	1. if (x >= 0) 2. doThis(); 3. if (x > 10) 4. doThat();

Tabela 2.3: Exemplo de mutações.

No **Particionamento de equivalência** o domínio de entrada é subdividido em uma coleção de subdomínios, ou classes equivalentes, e um conjunto de testes representativos é extraído a partir de cada classe identificada.

Na **Análise de valor-limite** os casos de teste são escolhidos próximo ou nas fronteiras dos domínios de entrada, uma vez que boa parte das falhas tende a se concentrar próximo a esses valores.

2.4.2 Critérios baseados em erros

Os critérios de teste baseados em erros utilizam informação sobre os erros mais comumente detectadas no processo de desenvolvimento e sobre tipos específicos de erros que se deseja revelar. Dois critérios bem conhecidos são: a semeadura de erros (Budd, 1981) e análise de mutantes (DeMillo et al., 1978).

Na **Semeadura de erros** uma quantidade específica de erros é inserida artificialmente no programa. Após o teste, do total de erros encontrados, verificam-se quais são naturais e quais são artificiais. Usando estimativas de probabilidade, o número de erros naturais ainda existentes pode ser estimado (Goel, 1985).

A **Análise de mutantes** é um critério que utiliza um conjunto de programas ligeiramente modificados (mutantes), obtidos a partir do programa original P, para avaliar o quanto um conjunto de testes T é adequado ao programa P. O objetivo é encontrar um conjunto de testes T capaz de revelar todas as diferenças de comportamento entre P e seus mutantes. A Tabela 2.3 apresenta o exemplo de um programa original (acima e à esquerda) e três mutantes. Observe que a mutação gera programas diferentes e que deveriam ser detectados por casos de teste para testar o programa original. Caso os mutantes gerados tivessem um comportamento idêntico ao programa original diríamos que os mutantes são equivalentes. Se o conjunto de testes T detecta a diferença existente no mutante, dizemos que T matou o mutante. A partir dos mutantes mortos e do total de mutantes não equivalente podemos obter o escore de mutação (Em), que é o número de mutantes mortos (Mm) dividido pelo número total de mutantes gerados (Tm) menos os mutantes equivalentes (Me), ou seja: $Em = Mm/(Tm - Me)$.

2.5 Documentos de teste

Independentemente do processo de software utilizado durante o desenvolvimento, existem artefatos diretamente relacionados à atividade de teste (IEEE, 1998). Dentre esses artefatos destacamos:

- Plano de teste. Documento que detalha o escopo, abordagem, recursos e agenda das atividades de teste. Além disso, o Plano de Teste identifica os itens de teste, construções a serem testadas, tarefas a serem realizadas, pessoal para realização e riscos associados. Esse documento é produzido durante a atividade *Planejamento*.
- Especificação de casos de teste. Documento especificando entradas, resultados esperados e um conjunto de condições de execução para um item de teste. Esse documento é produzido durante a atividade *Desenho*.
- Especificação dos Procedimentos de Teste. Documento contendo os passos para execução de um conjunto de casos de teste. Esse documento também é produzido durante a atividade *Desenho*.
- Relatório de incidentes. Documento relatando qualquer evento ocorrido durante o processo de teste que requer investigação. Esse documento é produzido durante a atividade *Execução*.
- Log de testes. Registro cronológico de detalhes sobre a execução dos testes. Esse documento também é produzido durante a atividade *Execução*.
- Relatório de testes. Documento resumindo as atividades de teste e os resultados obtidos, incluindo uma avaliação dos testes executados. Esse documento é produzido durante a atividade *Verificação de término* e *Balanço final*.

2.6 Teste Baseado em Modelos

O Teste Baseados em Modelos (TBM) consiste em uma técnica para geração automática de um conjunto de casos de testes, com entradas e saídas esperadas, utilizando modelos extraídos a partir dos requisitos do software (Binder, 2000). Para que o TBM possa ser utilizado é necessário que a especificação do software seja definida utilizando modelos em um formato apropriado para a automação das atividades de teste. São exemplos desses formatos o uso de modelos representados utilizando métodos formais, máquinas de estado finito e a UML.

O TBM surge como uma abordagem aplicável para controlar a qualidade do software, assim como reduzir os custos associados ao processo de testes, visto que casos de teste podem ser gerados a partir da especificação do software, paralelamente ao seu desenvolvimento, utilizando procedimentos automáticos que podem ser menos suscetíveis a erros.

O TBM pode gerar melhorias nas atividades de teste por meio da simplificação do seu planejamento, da semi-automação de suas atividades, e controle, a partir da gerência dos testes e possibilidade de re-execução automática após modificações.

Conforme mencionado anteriormente, diversos tipos de modelos podem ser utilizados no TBM. Neste trabalho utilizamos o TBM associado a modelos UML, sendo esses modelos considerados semi-formais. Para que o teste possa ser baseado em tais modelos utilizamos uma linguagem para especificação de expressões que seja livre de efeitos colaterais, conforme será discutido no próximo capítulo.

2.7 A linguagem UML

A Unified Modeling Language (UML) (Booch et al., 1999; Rumbaugh et al., 1999) é uma linguagem de modelagem não proprietária de terceira geração. A UML permite que desenvolvedores visualizem os produtos de seu trabalho em diagramas padronizados. Junto com uma notação gráfica, a UML também especifica significados, isto é, semântica. É uma notação independente de processos, embora alguns processos bem conhecidos tenham sido especificamente desenvolvidos utilizando a UML, como é o caso do Rational Unified Process (Kruchten., 2003).

O desenvolvimento de um sistema envolve vários aspectos, tais como: o aspecto funcional, não funcional e aspectos organizacionais. Devido a isso, a UML possui o conceito de visões. Cada visão é descrita por diagramas que contêm informações dando ênfase a aspectos particulares do sistema. Os diagramas que compõem as visões contêm os modelos de elementos do sistema. As visões da UML são descritas a seguir:

- Visão de Casos de uso. Essa visão modela a funcionalidade do sistema como percebido por seus usuários externos, chamados de atores. Um caso de uso é uma unidade coerente de funcionalidade expressa como uma transação entre atores e o sistema. O objetivo dessa visão é listar os atores e os casos de uso e mostrar que atores participam em cada caso de uso. Os diagramas de casos de uso provêm a notação visual para representação dos conceitos nessa visão.

- Visão de Interação. Essa visão descreve as seqüências de mensagens trocadas entre papéis que implementam comportamento no sistema. Um papel *classificador* é a descrição de um objeto que age como uma parte específica dentro de uma interação. Essa visão provê uma visão holística do comportamento do sistema, ou seja, ela mostra o fluxo de controle entre diversos objetos. Os diagramas de interação provêm a notação visual para os conceitos dessa visão. A parte dinâmica das colaborações, descritas por diagramas de interação, que podem ser diagramas de seqüência ou de colaboração, realizam os casos de uso.

- Visão Estática. A visão estática modela conceitos do domínio da aplicação, assim como conceitos internos criados como parte da implementação de uma aplicação. Essa visão é estática por que ela não descreve o comportamento do sistema dependente do tempo. Os principais constituintes da visão estática são as classes e seus relacionamentos: *associação*, *generalização* e vários tipos de *dependência*, assim como *realização* e *uso*. Uma

classe é a descrição de um conceito do domínio da aplicação ou da solução. Classes são o centro no qual a visão é organizada; os outros elementos são de propriedade ou associados às classes. A visão estática é exibida em diagramas de classes, assim chamados por que seu principal foco é a descrição das classes.

- Visão de Máquinas de estados. Uma máquina de estados modela o possível ciclo de vida de uma instância de uma classe. A máquina de estado contém estados conectados por transições. Cada estado modela um período de tempo durante a vida de um objeto, durante o qual ele satisfaz certas condições. Quando um evento ocorre, ele pode causar a execução de uma transição que leva o objeto para um novo estado. Quando uma transição é executada, uma ação associada pode ser também executada. Máquinas de estados são exibidas em diagramas de estados.

- Visão de Atividades. Um grafo de atividade é uma variante de uma máquina estado. Ele mostra as atividades computacionais envolvidas na execução de um cálculo. Um *estado atividade* representa uma atividade: um trabalho, passo ou execução de uma operação. Um grafo de atividade descreve grupos de atividades seqüenciais ou concorrentes. Grafos de atividades são exibidos como diagramas de atividades.

- Visão de Implementação. Essa visão modela componentes em um sistema, isto é, unidades de software a partir da qual a aplicação é construída, assim como as dependências entre componentes, de tal forma que o impacto de uma mudança proposta em um componente possa ser avaliado. Ele também modela a atribuição de classes e de outros elementos do modelo a componentes. A visão de implementação é exibida em diagramas de componentes.

- Visão de Desdobramento. Essa visão representa o arranjo das camadas suportadas nas instâncias dos nodos computacionais. Um nodo é um recurso de execução, como um computador, um dispositivo ou memória. Essa visão permite a avaliação das conseqüências relacionadas à distribuição ou alocação de recursos.

- Visão de Gestão de modelo. Essa visão modela a organização do próprio modelo. Um modelo compreende um conjunto de pacotes que mantém elementos de modelo, como classes, máquinas de estados e casos de uso. Pacotes podem conter outros pacotes: assim, um modelo é um pacote raiz que indiretamente contém todos os outros elementos do modelo. Pacotes são unidades para manipulação de conteúdo do modelo, assim como unidades para controlar o acesso e a configuração. Cada elemento de modelo pertence a um pacote ou outro elemento.

2.8 Sistemas de Informação

Existem quatro categorias básicas de sistemas de informação. Cada uma dessas categorias está associada a um nível da organização e possui um perfil de usuários bem definido. A Figura

2.5 apresenta as diferentes categorias dos sistemas de informação, apresentando também o perfil dos usuários associados (Cook, 1996a).

Os sistemas do nível operacional dão suporte as transações existentes em uma organização. Eles fornecem respostas a perguntas simples relacionadas ao funcionamento da organização. Os Sistemas de Processamento de Transações (SPT) representam o principal tipo de SI associado a esse nível. Esses sistemas suportam as operações do dia a dia de uma organização, mantendo os dados resultantes dessas operações. Tais sistemas servem de base para os outros sistemas de informação.

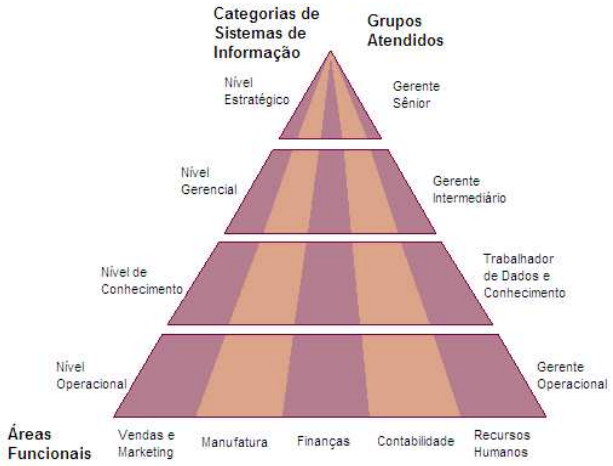

Figura 2.5: Categorias de Sistemas de Informação (Cook, 1996a).

Os sistemas no nível de conhecimento facilitam a integração de novos conhecimentos nos negócios e na organização. Os sistemas de automação de escritório, projetados para aumentar a produtividade das pessoas que trabalham com os dados elementares presentes na maioria das organizações e providos pelos SPT, são os grandes tipos de sistemas nesse nível.

Os sistemas no nível gerencial apóiam a tomada de decisão por parte dos gerentes da organização, por meio do fornecimento de relatórios periódicos ao invés de informações simples obtidas instantaneamente a partir das operações realizadas. Alguns dos principais tipos de sistema nesse nível são os sistemas de informações gerenciais, que dão apoio às atividades de controle, e os sistemas de suporte a decisão, que auxiliam gerentes na tomada de decisão, geralmente baseado em dados obtidos junto aos SPT.

Os sistemas no nível estratégico auxiliam os gerentes nas atividades de planejamento, dando subsídios para responder questões relativas à estimativas e tendências futuras associadas aos negócios realizados pela organização. Os sistemas de informação executivo são os principais tipos de sistema nesse nível. Esses sistemas auxiliam a tomada de decisão utilizando recursos avançados de análise dos dados.

2.8.1 Sistemas de Processamento de Transações

Neste trabalho nos concentramos nos sistemas de informação no nível operacional, representados pelos SPT, que constituem a base para construção de qualquer outro sistema de informação. Os SPT possuem algumas características interessantes que favorecem a automação dos testes. Esse tipo de sistema geralmente possui uma grande quantidade de dados de entrada, que de alguma forma devem ser processados e armazenados. Existe uma grande necessidade de armazenamento, geralmente criando uma grande quantidade de saída. O processamento é muita das vezes repetitivo, realizando sempre tarefas envolvendo cálculos simples (+, -, * e /). Esses sistemas são centrados nos dados e estão presentes na maioria das organizações. Mesmo com essa restrição temos um conjunto muito grande de sistemas, contendo quatro grandes atividades associadas ao seu funcionamento:

- Entrada de dados. Os dados são inseridos no sistema podendo ou não ser agregados. Essa entrada pode ser feita de forma manual ou automática.
- Manipulação de dados. Os dados sofrem algum tipo de processamento, seja por meio de transformação, redução ou classificação.
- Armazenamento de dados. Os dados processados são armazenados em algum mecanismo de armazenamento.
- Produção de documentos. Os dados são utilizados para produzir documentos que facilitem a visualização das informações associadas, seja por meio de documentos textuais ou gráficos.

2.9 O Desdobramento da Função Qualidade

O Desdobramento da Função Qualidade (QFD - Quality Function Deployment) (Cheng, 1995) é um método que auxilia na gestão do desenvolvimento de novos produtos, administrando o processo de transformação da informação em conhecimento tecnológico, e organizando o trabalho humano envolvido nesse processo. Assim, o QFD pode aumentar a eficiência da empresa no projeto de produtos e processos consonantes com os elementos básicos de competitividade escolhidos por ela, como estratégia de mercado.

O QFD tem por principal objetivo auxiliar na definição de uma qualidade projetada para o novo produto capaz de atender plenamente, e melhor que a concorrência, as necessidades dos clientes, aumentando assim o valor agregado do mesmo. Nesse sentido, é importante ressaltar que, agrega-se valor ao produto tornando-o desejado e cobiçado pelo mercado por meio do atendimento das necessidades dos clientes (Campos, 1992). O QFD, para garantir a obtenção dessa qualidade projetada, "coordena"o processo de tomada de decisões nas etapas subseqüentes do processo de desenvolvimento, de modo a garantir a adequação destas com a qualidade projetada, com os custos definidos e com a confiabilidade de desempenho do produto desejada.

O uso do QFD no desenvolvimento de software iniciou-se no Japão com o objetivo de estabelecer a qualidade de modo eficiente no produto e no seu desenvolvimento, visto que para esse fim o QFD foi identificado como uma abordagem de sucesso em outros ambientes. Essa adaptação é intitulada Desdobramento da Função Qualidade do Software (Software Quality Function Deployment - SQFD) (Haag et al., 1996). O SQFD focaliza a melhoria de qualidade do processo de desenvolvimento de software pela implementação de técnicas de melhoria de qualidade durante a fase de levantamento de requisitos. É uma técnica útil no levantamento de requisitos, adaptável a qualquer metodologia da Engenharia de Software, que levante e defina quantitativamente os requisitos do cliente.

2.10 Trabalhos relacionados

Conforme detalhado no próximo capítulo, o MODEST é um método de TBM descrito em três perspectivas. No nível lógico temos que mapear os conceitos do método para um processo de software e uma linguagem de modelagem. Decidimos realizar esse mapeamento utilizando o Processo Unificado e a UML, conforme prescrito pelo PU, visto que esse processo é um processo abstrato e foi utilizado como base para construção de diversos processos concretos. Por causa disso, focalizamos na Subseção 2.10.1 na descrição de trabalhos relacionados que utilizem a UML para fins de automação de testes.

Embora existam muitos trabalhos sobre a avaliação de métodos e ferramentas de apoio, até com guias sobre como realizar tal tipo de avaliação (Kitchenham et al., 1995), na área de TBM isso não é tão comum. Os trabalhos discutidos na Subseção 2.10.1 não apresentam uma avaliação experimental seguindo as recomendações da Engenharia de Software Experimental. Na Subseção 2.10.2 discutimos alguns trabalhos que realizaram estudos experimentais similares ao que realizamos para avaliar o MODEST, tentando evidenciar as conclusões obtidas por eles.

2.10.1 Trabalhos relacionados à automação de testes utilizando a UML

Offutt e Abdurazik (1999) reivindicam ser o primeiro grupo a formalizar uma técnica de teste baseada na UML. Eles formalizaram quatro critérios de teste específicos para diagramas de estados da UML. Os critérios são:

- Cobertura de transições: dado um modelo M contendo a especificação do sistema, o conjunto de testes T deve conter testes que exercitem cada transição T contida em M.
- Cobertura completa de predicados: dado um modelo M contendo a especificação do sistema, o conjunto de testes T deve conter testes que façam com que cada cláusula C, para cada predicado P contido em M, resultem em um par de saídas em que o valor de P seja diretamente relacionado ao valor de C.

- Cobertura de par de transições: dado um modelo M contendo a especificação do sistema, o conjunto de testes T deve conter testes que exercitem os pares de transições em seqüência.

- Seqüência completa: dado um modelo M contendo a especificação do sistema, o conjunto de testes T deve criado de forma a contemplar uma seqüência de transições definida pelo Engenheiro de Testes.

Embora essa tenha sido uma contribuição relevante, esses critérios são baseados em diversos trabalhos publicados sobre o teste de máquinas de estados finitos (Lee e Yannakakis, 1996). Além disso, alguns critérios não são detalhados o suficiente: o critério de cobertura completa dos predicados pode ser indecidível em certos casos e nada sobre isso foi considerado. Os autores ainda desenvolveram um protótipo de uma ferramenta para implementação dos critérios, denominada UMLTest e fizeram uma avaliação experimental do seu uso, tentando avaliar a capacidade de detecção de falhas semeadas artificialmente, utilizando os testes gerados pela ferramenta. Os resultados indicaram que os testes gerados pela ferramenta foram muito efetivos na descoberta das falhas, porém, não existe uma definição precisa do estudo, como por exemplo uma caracterização das falhas utilizadas, perfil dos participantes e outras questões que podem ameaçar bastante as conclusões obtidas.

Além do trabalho descrito anteriormente, Abdurazik e Offutt (2000) desenvolveram um critério de teste baseado em diagramas de colaboração. A técnica foi considerada inovadora pelo fato de ser um dos primeiros trabalhos de definição de critérios de teste a utilizar descrições de projeto. No trabalho eles ainda apresentaram um algoritmo para instrumentação de programas que tenham diagramas de colaboração expressando seu projeto. De acordo com os autores, essa instrumentação poderia auxiliar a avaliação dos testes implementados, podendo auxiliar também a rastreabilidade entre projeto e código. No entanto, a instrumentação não parece ser útil no desenvolvimento de software, assim como o próprio critério, que mais se assemelha a uma diretriz para geração de testes do que um critério propriamente dito.

Briand e Labiche (2001) desenvolveram um método para o teste funcional de sistemas denominado TOTEM (**T**esting **O**bject-orien**T**ed syst**E**ms with the unified **M**odeling language). Esse método deriva requisitos de teste de artefatos criados durante o fluxo de análise de um processo baseado no PU. Esses artefatos incluem: diagrama de casos de uso, descrições dos casos de uso, diagramas de seqüência ou colaboração para cada caso de uso, diagramas de classe contendo as classes do domínio de aplicação, dicionário de dados descrevendo cada classe, método e atributo.

O método exige que cada classe seja caracterizada com uma invariante expressa em OCL e que cada operação seja descrita por um contrato, também expresso em OCL, detalhando pré e pós-condições. O TOTEM ainda exige a criação de um diagrama de atividade mostrando as dependências seqüenciais entre casos de uso para permitir a geração de testes. Essas dependências devem mostrar que ordem deve ser levada em consideração para a geração dos testes.

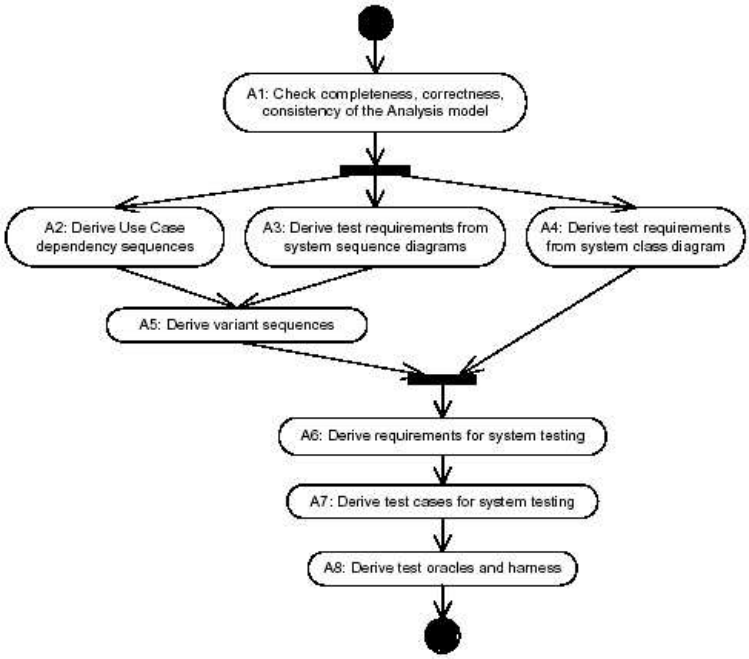

Figura 2.6: Diagrama de atividades do TOTEM.

A Figura 2.6 apresenta as atividades contidas no TOTEM. A atividade A1 é a primeira atividade onde as especificações devem ser verificadas para garantir sua completude e atendimento aos requisitos de testabilidade impostos pelo método. O trabalho focaliza as atividades A2, A3 e A5 e deixa uma avaliação do seu custo para trabalhos futuros, trabalhos esses ainda não realizados. As atividades A2, A3 e A5 constituem o método propriamente dito, apresentando como derivar requisitos de teste a partir da especificação, embora muitos passos sejam descritos de forma abstrata. Após a geração inicial desses requisitos, que basicamente dizem o que deve ser testado, eles são agrupados em um conjunto de testes. As atividades A7 e A8 estão relacionadas à geração dos casos de teste e código para oráculos, além de integrá-los em dispositivos executáveis.

Existem diversos problemas relacionados ao TOTEM. Seus autores não descrevem completamente como gerar os casos de teste, como iniciar o teste se for necessário algum dado previamente existente, e como reduzir, efetivamente, a grande quantidade de testes gerados pela técnica. Além disso, eles não relatam a existência de uma ferramenta de apoio ao método, nem de uma avaliação dos possíveis ganhos associados ao uso de tal técnica.

Farias e Machado (2003) desenvolveram um método de teste aplicável a componentes baseado no TOTEM, herdando algum de seus problemas. Barbosa et al. (2004) estenderam este trabalho e desenvolveram uma ferramenta de apoio ao método. Esse método de teste utiliza a técnica de análise de riscos para definir quais funcionalidades devem ser testadas e quantos casos de teste devem ser criados. Algumas convenções existentes no método podem tornar difícil o seu uso no ambiente industrial, como por exemplo o formato dos diagramas de seqüência ilustrando os roteiros dos componentes sob testes. Não existe uma avaliação

empírica do uso do método, que poderia auxiliar a análise do nível de dificuldade associado às convenções existentes.

O projeto AGEDIS (Hartman e Nagin, 2004) criou uma metodologia e ferramentas para geração e execução automática de TBM para sistemas distribuídos. O projeto inclui um ambiente integrado para modelagem, geração e execução de testes, e outras atividades relacionadas.

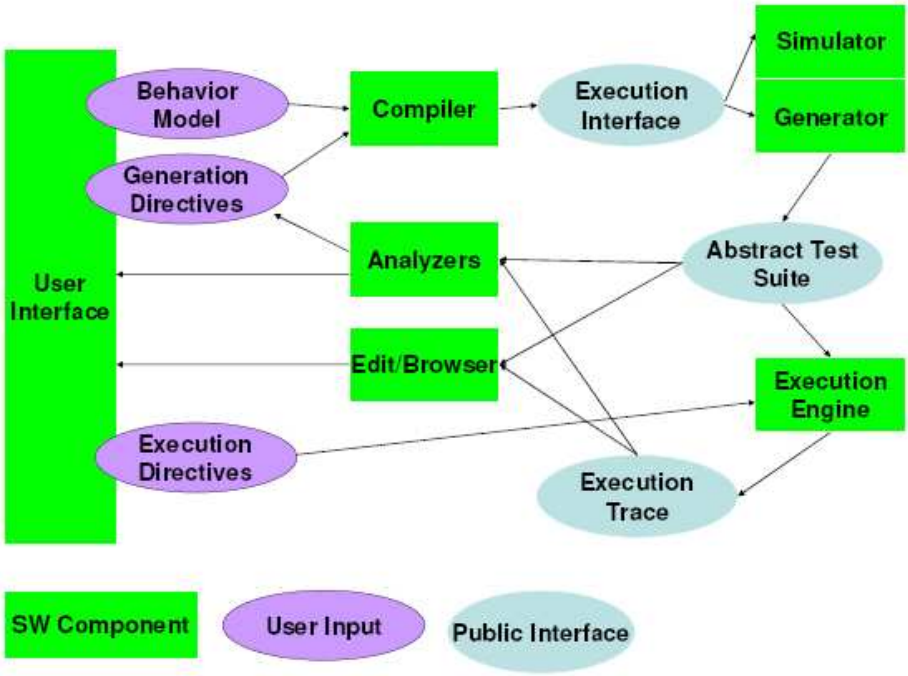

Figura 2.7: Arquitetura do AGEDIS.

A arquitetura do AGEDIS foi projetada para permitir a participação e intercâmbio de componentes de software quando necessário. Isso proporciona a interoperabilidade com as ferramentas do AGEDIS, sendo possível adaptar outros elementos na arquitetura base. Essa arquitetura é apresentada na Figura 2.7. O modelo comportamental do SST deve ser especificado pelos usuário do AGEDIS utilizando a AGEDIS Modelling Language (AML). Utilizando a ferramenta de modelagem Objecteering's UML é possível gerar um arquivo XML contendo o modelo do SST para então utilizar o compilador de modelos do AGEDIS, que consegue ler esse formato. O compilador converte o modelo do SST em um formato intermediário que serve como interface de execução. As diretivas de geração, incluindo metas de cobertura, restrições na suite de testes e objetivos dos testes são compilados juntos com o modelo e carregados no gerador de testes. As diretivas de geração de testes encapsulam a estratégia de teste. A interface de execução pode ser executada por um simulador manual ou um gerador de casos de teste automático. O simulador manual é utilizado para verificar o comportamento do modelo assim como para produzir objetivos adicionais para o teste. O gerador de teste gera uma suite de teste abstrata (Abstract Test Suite - ATS), que consiste de caminhos e grafos através do modelo, satisfazendo o critério de cobertura indicado. Esse formato contém toda a informação

necessária para executar os testes e verificar os resultados em cada estágio de cada teste. As diretivas de execução do teste (Test Execution Directives - TED) funcionam como ponte entre as abstrações utilizadas no modelo comportamental e os detalhes de implementação do SST. A máquina de execução lê a suite de teste abstrata e as diretivas de execução dos testes para executar a suite no SST. Cada estímulo para cada transição modelada na forma de diagramas de estados no modelo do SST é utilizada nos testes. A máquina de execução espera por respostas da aplicação para verificar se o estado da aplicação e do modelo conferem. Esses resultados são armazenados para posterior análise e visualização.

O Visualizador é capaz de mostrar o resultado da execução e a suite de teste abstrata de forma visual, facilitando assim a análise dos testes. O Analisador de Cobertura lê os dados sobre a execução dos testes e identifica áreas do modelo que não foram suficientemente cobertas. Existe uma integração dessa ferramenta com o gerador de testes para permitir a geração de testes adicionais. A interface de usuário do AGEDIS permite invocar a ferramenta de modelagem, permitindo assim a edição do modelo do SST, assim como também permite a edição das diretivas de teste e diretivas de execução.

O método proposto pelo AGEDIS é baseado em um processo interativo composto por diversos passos. A Fig. 2.8 apresenta uma visão geral do método para sintetizar casos de teste a partir de especificações UML.

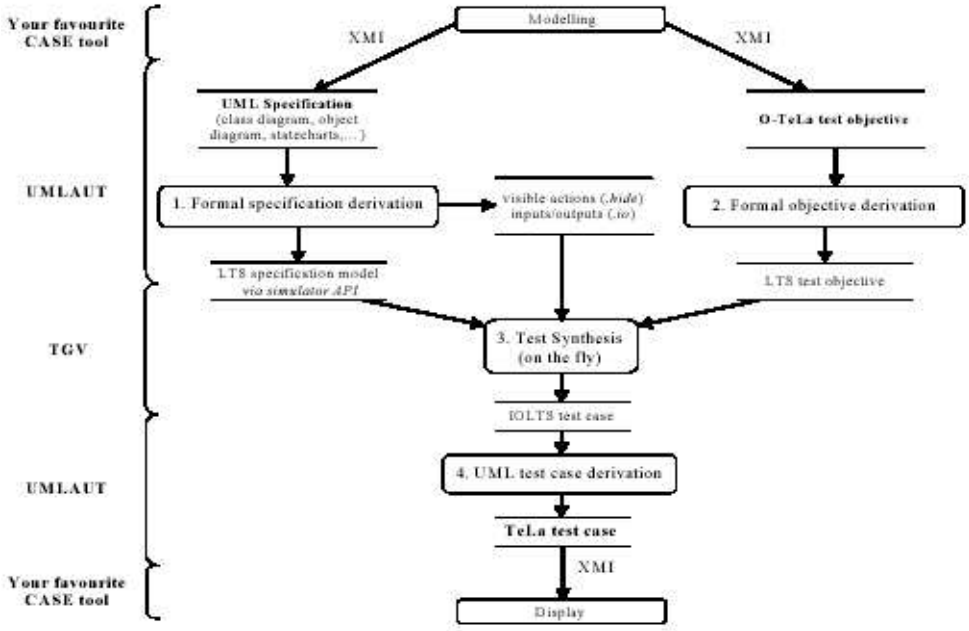

Figura 2.8: Atividades do AGEDIS.

As entradas do método são: i) uma especificação UML contendo diagramas de classe, diagramas de estados e diagramas de objeto e de implementação descrevendo o SST, e ii) um objetivo de teste na forma de um cenário descrito na linguagem O-TeLa. A saída é um conjunto de casos de teste abstratos na forma de cenário, descrito em TeLa mas que pode ser gerado de forma executável para uma plataforma específica. O método é dividido em quatro

passos:

- Derivação da especificação formal;
- Derivação formal de objetivos;
- Síntese de testes a partir dos modelos formais;
- e Derivação de casos de teste.

O usuário do AGEDIS deve entrar com três tipos de informação diferente sobre o SST: i) o modelo comportamental; ii) as diretivas de execução dos testes, que descrevem a arquitetura do SST; e iii) as diretivas de geração de testes. Tanto o modelo comportamental quanto as diretivas de execução dos testes são informados utilizando uma ferramenta de modelagem UML equipada com o perfil UML do AGEDIS. As diretivas de geração de testes são informadas utilizando um editor XML.

Várias ferramentas foram integradas ao arcabouço do AGEDIS, incluindo: i) uma ferramenta de modelagem UML, ii) um compilador de modelos, iii) um simulador de modelos, iv) uma máquina de geração de testes, v) uma máquina de execução de testes, vi) um navegador e editor de testes, vii) uma ferramenta de análise de defeitos, e viii) um gerador de relatórios. Todas essas ferramentas são ativadas a partir de uma interface gráfica de usuário, que possui facilidades para o gerenciamento dos vários artefatos produzidos no processo de teste.

Alguns problemas são relatados em trabalhos do projeto, tais como as convenções da linguagem de modelagem utilizada, o uso de diagramas de estados como sendo o principal meio de descrição do funcionamento do SST, e a linguagem utilizada como linguagem de ação. O projeto apresenta alguns estudos de caso relatando a aplicação do método mas não se aprofunda muito nessa descrição. Esses estudos de caso mostram que o uso do método pode trazer ganhos na qualidade dos testes gerados.

O IRISA desenvolveu um método e uma ferramenta para geração automática de casos de teste a partir de especificações de desenho UML (Nebut et al., 2003). O método prevê a criação de diversos diagramas que podem inviabilizar seu uso, devido as restrições associadas à sua criação. Esse trabalho é usado como base para construção de um outro trabalho fundamentado no uso de diagramas de caso de uso anotado com contratos para a geração de testes (Pickin et al., 2002). Existe uma avaliação preliminar descrita em outro trabalho que mostra que os testes gerados de forma automática são melhores que os gerados manualmente, mas a descrição dos estudos experimentais é bastante superficial (du Bousquet et al., 2001).

Andrews et al. (2003) definiram critérios de adequação de testes para diagramas de classe e de colaboração. Esse trabalho também define um método de teste utilizando os critérios de teste definidos por eles. A Tabela 2.4 mostra os critérios para diagramas de classes, e a Tabela 2.5 mostra os critérios para diagramas de colaboração definidos pelo método. Alguns critérios possuem instruções ambíguas e que podem gerar diferentes interpretações durante sua implementação, como por exemplo o critério de Multiplicidade de associação, que fala de multiplicidades representativas mas não deixa claro o que isso significa. Por causa disso,

Critério	Descrição
Multiplicidade de associação	Dado um conjunto de testes T e um modelo M, T deve causar a criação de cada par de multiplicidades representativas em M.
Generalização	Dado um conjunto de testes T e um modelo M, T deve causar a criação de cada especialização em uma generalização.
Atributo de classe	Dado um conjunto de testes T, um modelo M e uma classe C, T deve causar a criação de um conjunto representativo de combinações de valores de atributos em cada instância de C.

Tabela 2.4: Critérios de teste para diagramas de classe definidos por Andrews et al. (2003)

Critério	Descrição
Cobertura de condição	Dado um conjunto de testes T e um diagrama de colaboração D, T deve causar cada condição em cada decisão ser avaliada como VERDADEIRO e FALSO.
Cobertura completa de predicados	Dado um conjunto de testes T e um diagrama de colaboração D, T deve causar cada cláusula em cada condição em D ser avaliada como VERDADEIRO e FALSO enquanto todas as outras cláusulas no predicado tenham valores tais que o valor do predicado seja sempre o mesmo da cláusula sendo testada.
Todas as mensagens	Dado um conjunto de testes T e um diagrama de colaboração D, T deve causar o acionamento de cada mensagem entre os objetos de D.
Cobertura de coleção	Dado um conjunto de testes T e um diagrama de colaboração D, T deve causar a criação de cada multiplicidade representativa de uma coleção de objetos de D.

Tabela 2.5: Critérios de teste para diagramas de colaboração definidos por Andrews et al. (2003)

esses critérios se assemelham bastante com um guia para a geração de teste ao invés de um critério propriamente dito. No entanto, iremos utilizar o termo critério quando nos referirmos a eles, visto que alguns são utilizados pelo MODEST. O trabalho não apresenta um estudo experimental do uso método e dos seus critérios, mas descreve a aplicação do método para um exemplo particular.

Pilskalns et al. (2003) apresenta uma abordagem para integrar duas visões complementares de projetos UML, por meio do uso de diagramas de classe e diagramas de seqüência. O trabalho usa informação de ambos para derivar e executar testes que satisfaçam a critérios de adequação de testes definidos por Andrews et al. (2003) para as duas visões envolvidas. Para fazer isso, atributos e parâmetros dos diagramas de classes são extraídos, diagramas de seqüência são transformados em um grafo acíclico e um algoritmo que combina as representações em uma tabela de execução de testes é acionado. Não existe um estudo experimental do uso do método, indicando a aplicabilidade prática do mesmo.

Ghosh et al. (2003) apresenta a descrição de semânticas UML executáveis e seus usos na execução de testes. O trabalho usa os critérios de adequação de testes definidos por Andrews et al. (2003). O foco do artigo é o uso dos critérios de teste para definir objetivos que auxiliem na criação e melhoria dos casos de teste. O trabalho demonstra como casos de teste podem cobrir diversos elementos em múltiplos domínios de cobertura. A efetividade do

método e uma avaliação do esforço em seu uso não são examinadas.

Archetest é uma técnica de TBM que trabalha com casos de uso e modelos de domínio descritos em UML. A ferramenta para o Archetest é disponibilizada na forma de um plug-in para o Rational Rose [2]. O modelo de domínio é um diagrama de classe que serve para indicar as classes que podem ser instanciadas pelo sistema. Os casos de uso são formalizados para produzir uma especificação que possa ser utilizada para a geração de testes. A partir de um modelo no formato exigido pelo Archetest diversas transformações ocorrem até a geração dos casos de teste. Um estudo preliminar da ferramenta indica que ela pode aumentar tanto a qualidade dos testes gerados, assim como reduzir os custos associados (Williams, 2001).

2.10.2 Trabalhos relacionados à avaliação de métodos de TBM

Sinha e Smidts (2006) desenvolveram HOTTest, uma técnica de TBM baseada em modelos descritos utilizando HaskellDB. Eles realizaram um estudo experimental comparando o HOTTest com uma técnica de TBM baseada em máquinas de estados finitos estendidas (EFSM - Extended Finite State Machines). Os aspectos de desempenho dos testes gerados utilizando as duas técnicas foram comparados com relação a usabilidade. O principal resultado dessa análise é que o HOTTest possibilita a geração de testes mais efetivos que os testes gerados pela técnica com EFSM. Além disso, mesmo os participantes do estudo tendo achado o HOTTest mais difícil de usar que a técnica baseada em EFSM, eles aprenderam a utilizar HOTTest mais rápido que a técnica baseada em EFSM e cometeram menos erros, além dos erros serem menos críticos.

Pretschner et al. (2005) realizaram um estudo experimental no ambiente de um controlador de rede automotiva para avaliar diferentes baterias de testes geradas, em termos de capacidade de detecção de erros, cobertura do modelo e cobertura de código. Algumas dessas baterias de teste foram geradas automaticamente com e sem modelos, puramente aleatórias e com critérios de seleção de testes funcionais. Outras baterias foram derivadas manualmente, com e sem um modelo disponível. As baterias de testes geradas a partir do uso de modelos detectaram significativamente mais erros nos requisitos do que as baterias de teste geradas diretamente pelos requisitos, sem o auxílio de modelos. O número de falhas detectadas não dependeu do uso de modelos. Os testes automaticamente gerados utilizando modelos detectaram tantos erros quanto os testes gerados manualmente a partir dos modelos, com a mesma quantidade de testes. Um aumento da ordem de seis vezes no número de testes gerados automaticamente, em relação aos testes gerados manualmente, levou a um aumento de 11% na quantidade de falhas detectadas.

2.11 Requisitos para o Teste Baseado em Modelos

A indústria de software tem o desafio de desenvolver uma grande variedade de produtos de software com qualidade cada vez maior e custo cada vez menor. Muitos tipos de iniciativas

[2] http://www-306.ibm.com/software/rational/

têm sido promovidas para tentar auxiliar essa tarefa desafiadora. Uma importante contribuição nesse sentido é a proposição de catálogos de requisitos capturando as metas, necessidades, expectativas e restrições de desenvolvedores ou usuários de uma certa área, tal qual foi realizado por Hoffmann et al. (2004). Eles definiram um catálogo de requisitos para ferramentas de gestão de requisitos. Catálogos de requisitos auxiliam usuários e desenvolvedores a comparar e selecionar métodos e ferramentas. Eles também auxiliam projetistas de métodos e ferramentas a direcionar seus esforços.

Apesar do uso de métodos e ferramentas de teste no desenvolvimento de software, muitos sistemas ainda são entregues com uma grande quantidade de falhas. Essas falhas não são encontradas, em parte, por causa de métodos e ferramentas de teste inadequadas (NIST, 2002). Nesta seção, apresentamos um catálogo de requisitos diretamente associados a métodos para o TBM, focalizando em SI, uma vez que tal tipo de sistema influenciou bastante a proposição desse catálogo.

Esta parte do trabalho iniciou quando preparamos um catálogo de requisitos preliminar baseado em alguns dos mais importantes trabalhos nessa área (Offutt e Abdurazik, 1999; Abdurazik e Offutt, 2000; Briand e Labiche, 2001; Andrews et al., 2003; Hartman e Nagin, 2004). Utilizamos esse catálogo preliminar para desenvolver nosso método de TBM e sua ferramenta de apoio. A relevância do catálogo ficou comprovada nesse momento, visto que alguns requisitos não foram considerados nos métodos analisados. Todos os aspectos do nosso método de TBM são de alguma forma capturados no catálogo, mas o contrário não é verdade. Conforme enfatizado por Hoffmann et al. um catálogo não é um documento definitivo, ao invés disso ele deve ser sujeito a revisões periódicas e modificações devido a novas tendências tecnológicas ou novas tendências nos processos de desenvolvimento. Essas modificações podem gerar novos requisitos ou podem enfraquecer ou mesmo invalidar outros.

O catálogo de requisitos e o desenvolvimento do nosso método de TBM influenciaram um ao outro, contudo o catálogo é um esforço mais abrangente, uma vez que, após um esforço inicial ele foi estendido por meio da participação de diversos profissionais ligados ao desenvolvimento de software.

Utilizamos dois métodos de TBM, além do nosso próprio método, para validar, informalmente, o catálogo de requisitos, levando em consideração o atendimento aos requisitos constantes no catálogo nos métodos considerados. Nesta seção não discutiremos os mecanismos utilizados pelo MODEST para atendimento aos requisitos identificados. Deixamos isso para o próximo capítulo, onde descrevemos de forma detalhada o método.

O método TOTEM (Briand e Labiche, 2001) foi escolhido por ele ter sido o primeiro método de TBM baseado no uso de modelos UML. Embora ele não possua uma ferramenta de apoio, ele introduziu diversos conceitos interessantes sobre como usar modelos UML para a automação de testes. O AGEDIS (Hartman e Nagin, 2004) é um método para o TBM desenvolvido por um consórcio liderado pelo laboratório de pesquisa da IBM em Haifa e que possui diversos colaboradores. Diversos componentes e ferramentas foram incorporados ao arcabouço do AGEDIS.

Nós estruturamos o catálogo de forma similar a estrutura utilizada por Hoffmann et al.

(2004). Eles descreveram o catálogo de requisitos para ferramentas de gestão de requisitos utilizando uma estrutura hierárquica. Nós utilizamos uma hierarquia similar, com três níveis. O primeiro nível corresponde aos trabalhadores diretamente associados ao requisito. Como o PRAXIS não possui o conceito de trabalhador associado às atividades de teste, utilizamos os trabalhadores do PU, visto que as atividades de teste desse processo e do PRAXIS são praticamente iguais. O segundo nível é representado pelo título da subseção e dá nome a um objetivo, expectativa ou restrição. O terceiro nível é um texto discutindo diversos aspectos relacionados ao segundo nível, juntamente com uma discussão sobre a implementação desses requisitos nos métodos de TBM analisados.

Os requisitos são apresentados com uma prioridade que foi estabelecida com o auxílio da equipe de profissionais que participaram de sessões de levantamento de requisitos. Essas prioridades foram identificadas no segundo nível para simplificar a descrição. As prioridades "alta", "média"e "baixa"são indicadas por "(+++)", "(++)"e "(+)"respectivamente.

Os trabalhadores utilizados para estruturar a apresentação dos requisitos são os trabalhadores de teste do PU (Jacobson et al., 1999) e o Engenheiro de Processos, que é uma figura presente no processo de software da Rational (Kruchten., 2003). Os principais trabalhadores da disciplina de teste do PU são: (i) o Engenheiro de Teste, (ii) o Engenheiro de Componente e (iii) o Testador do Sistema. O Engenheiro de Teste é responsável pelo planejamento dos testes, decidindo as metas de teste e uma agenda para os testes; é responsável ainda pela descrição dos casos de testes e procedimentos de testes, e por avaliar os testes quando eles forem executados. O Engenheiro de Componentes é responsável pela criação dos componentes de teste que automatizarão os procedimentos de teste. O Testador do Sistema é responsável pela execução dos testes e geração dos relatórios de execução. O Engenheiro de Processo é responsável, dentre diversas atividades, pelo desenvolvimento do processo de software. Isto inclui a melhoria contínua do processo, a partir da recomendação da adoção de práticas chaves no desenvolvimento de software.

Os trabalhadores selecionados capturam os objetivos, necessidades, expectativas e restrições relacionadas a outras pessoas com interesse nas atividades de teste, como por exemplo os clientes. Ainda que nossos trabalhadores sejam baseados no PU, é possível a adaptação do catálogo para outros contextos, como por exemplo, métodos ágeis, apesar do fato que alguns aspectos possam envolver restrições ainda não avaliadas.

2.11.1 Requisitos do Engenheiro de Teste

2.11.1.1 Verificação do relacionamento entre o software e o armazenamento++

Um sistema consiste do software e de outros componentes. A Figura 2.9 apresenta uma possível organização para um sistema (Filho, 2003). O software é o núcleo do sistema. Ele implementa estrutura complexas e flexíveis. Os outros componentes são também importantes: as plataformas de hardware (Hardware), os recursos de comunicação (Comunicação) e o mecanismo de armazenamento (Armazenamento). A maioria dos sistemas, notadamente os SI, exigem algum tipo de Armazenamento e geralmente utilizam Comunicação. Isso leva

a existência de um relacionamento entre o software e o mecanismo de armazenamento. Os métodos de TBM devem testar essa relação.

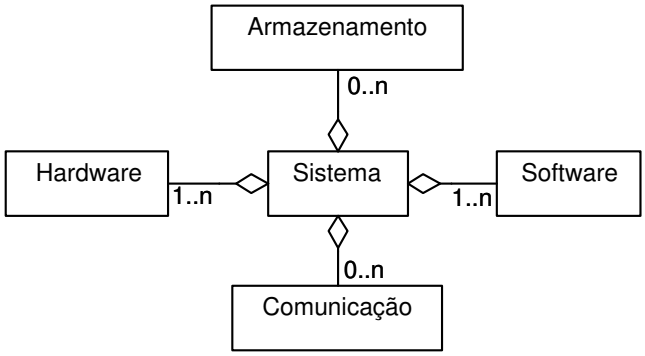

Figura 2.9: Uma organização para sistema.

Na prática a maioria dos métodos para o TBM abordam essa questão relacionada ao Armazenamento. Eles contêm atividades que prescrevem a identificação dos dados persistentes e dos seus relacionamentos. Os métodos de TBM deveriam usar todos os aspectos da descrição dos dados persistentes para garantir a corretude dos dados do sistema. Contudo, testar apenas as funcionalidades do software não necessariamente garante o funcionamento correto do relacionamento entre o Software e o Armazenamento, uma vez que o software pode utilizar um dado que não é gerado por ele mesmo, assim como um software pode gerar dados utilizados em outros softwares.

Requisito 1 Um método de TBM deveria prescrever o teste dos aspectos envolvendo o relacionamento entre o Software e o Armazenamento.

O TOTEM não trata, de forma explícita, o relacionamento entre o Software e o Armazenamento, mas ele prescreve o uso da descrição dos dados persistentes para geração de testes. Isto é feito por meio de diagramas de classe mostrando os dados persistentes e suas associações.

O AGEDIS não trata, de maneira explícita, o relacionamento entre o Software e o Armazenamento. Esta relação é inferida a partir do modelo comportamental descrevendo o SST, modelo esse criado utilizando diagramas de classes e diagramas de estados. O funcionamento de cada classe é descrito em um diagrama de estados integrado com uma linguagem de ações. As classes persistentes são descritas nesse modelo.

2.11.1.2 Geração de testes para requisitos funcionais+++

Métodos de TBM deveriam especificar mecanismos para a geração automática de casos de teste. Isso é muito difícil de ser capturado em um simples requisito, uma vez que ele envolve o dilema entre especificar um requisito e discutir mecanismos específicos que o realizam, de forma similar ao dilema *o quê* x *como*. Existem diversos aspectos relacionados à questão sobre como realizar a geração de testes para requisitos funcionais em SI: quais interfaces de

usuário deveriam ser consideradas, como as transferências entre essas interfaces deveriam ser verificadas, e como os campos de dados nas interfaces de usuários deveriam ser utilizados para a geração de testes. Esses aspectos estão diretamente associados ao conceito de critério: um critério de teste é um mecanismo para decidir *o que/quais* propriedades um conjunto de casos de teste deveria possuir para ser considerado adequado. A instância de um critério define *como* os dados de entrada deveriam ser utilizados para gerar teste. Dessa forma, critérios podem ser utilizados para selecionar casos de teste ou para decidir se um conjunto de testes é adequado (Goodenough e Gerhart, 1975). Um método de TBM deveria prescrever o teste dos requisitos funcionais de um sistema, e, além disso, prescrever o uso de critérios ou definir mecanismos para a definição de critérios a serem utilizados durante a geração de testes.

Requisito 2 Um método de TBM deveria prescrever a geração de testes para os requisitos funcionais.

Requisito 3 Um método de TBM deveria prescrever a geração de testes utilizando critérios de teste.

O TOTEM não discute de forma explícita os critérios utilizados na geração de testes. As mensagens nos diagramas de seqüência descrevendo o sistema são utilizadas na geração de testes para os requisitos funcionais, mas não existem detalhes sobre como isso é utilizado.

O AGEDIS prescreve o uso de diretivas para a geração de testes para os requisitos funcionais indicando o critério a ser utilizado. Existem parâmetros enviados ao gerador de teste que podem ser utilizados para especificar o grau de cobertura desejado. Existem cinco diretivas para geração de testes que podem ser utilizadas.

Uma característica dos SI é a integração com outros sistemas de uma organização. Geralmente isso é feito por meio de interfaces de hardware ou de software. Essas interfaces são mecanismos para comunicação com outros produtos, que definem a fronteira a partir da qual duas entidades independentes podem interagir uma com a outra. Elas podem ser utilizadas para entrada ou saída de dados do sistema, portanto, métodos de TBM deveriam especificar a modelagem dessas interfaces, no intuito de gerar testes utilizando critérios relacionados a esse tipo de informação.

Uma outra característica de um SI é o uso GUI. Essas interfaces podem possuir diferentes formatos de exibição, ativando e desativando elementos contidos nas interfaces, como comandos e campos. Uma outra características dessas GUI é a possibilidade de existência de regras de controle para o seu acesso, exigindo que diversas restrições sejam atendidas para que elas possam se tornar navegáveis. Ou seja, existem diversas características relacionadas às GUI, assim, conforme mencionado anteriormente, é necessário que os métodos para o TBM prescrevam a modelagem dessas GUI, incluindo suas diferentes formas de exibição e restrições relacionadas à navegação, para que testes possam ser gerados utilizando critérios que levem em consideração essa informação.

De forma resumida, os critérios de teste deveriam focar nas questões-chave relacionadas a SI, como por exemplo a integração com outros sistemas e modelagem de aspectos das GUI,

para garantir uma maior capacidade de detecção de falhas pelos testes gerados. No entanto isso requer a modelagem de aspectos relacionados a essas questões-chave.

O TOTEM prescreve o uso da descrição das pós-condições dos métodos para geração de testes. Isso inclui as pós-condições dos métodos relacionados as GUI, mas não existe menção a uma verificação quanto às possíveis formas de exibição e sobre as restrições associadas à navegação entre essas GUI.

O AGEDIS prescreve a descrição de todas as classes do sistema no modelo comportamental, mas não existe uma prescrição explícita relacionado às GUI. Não existe prescrição sobre o teste de transferência de controle entre instâncias das GUI.

Um método de TBM direcionado para SI deveria focar na questão associada à modelagem de GUI, uma vez que isso é uma característica crítica em SI. Um método de TBM genérico não enfatiza tanto esses aspectos por que sua intenção é tentar abranger uma variedade maior de aplicações.

2.11.1.3 Geração de testes para requisitos não-funcionais+++

Conforme discutido anteriormente, Comunicação e Hardware são componentes de um Sistema. Esses elementos são diretamente relacionados a Arquitetura do Sistema (AS) (Kruchten, 1995). AS é um ponto chave no desenvolvimento e por isso seu teste é importante. A definição da AS deve levar em consideração os Requisitos Não-Funcionais (RNF), portanto uma correta implementação da AS pode garantir o atendimento a alguns RNF. Os métodos de TBM deveriam prescrever o teste da AS, uma vez que isso pode abranger o próprio teste de alguns RNF.

Alguns métodos de TBM restringem as classes de aplicações suportadas para facilitar o teste da AS e o teste de RNF, mas isso pode levar a uma estrutura rígida suportada pelos métodos. Por outro lado, métodos que não restrinjam as aplicações suportadas podem tornar inviáveis, em termos de custo/benefício, o teste da AS.

Requisito 4 Um método de TBM deveria prescrever o teste da arquitetura do sistema

Conforme mencionado anteriormente, uma característica comum dos SI é a capacidade de gerenciar uma grande quantidade de dados persistentes sendo acessado por um grande número de usuários. Isso está diretamente relacionado ao teste de RNF. Alguns testes de RNF, como por exemplo desempenho e estresse, podem ser automatizados utilizando modelos. Outros RNF, como usabilidade, dependem de uma avaliação humana e são difíceis de serem completamente automatizados. Métodos de TBM deveriam prescrever a geração de testes para RNF, quando isso for possível, e prescrever o auxílio à automação de testes em outros casos.

Requisito 5 Um método de TBM deveria prescrever a geração de testes para RNF.

O TOTEM não especifica qualquer mecanismo relacionado ao teste de RNF. Além disso, não existe qualquer restrição sobre a arquitetura dos sistemas suportados pelo método, dando a entender que ele é um método genérico.

O AGEDIS prescreve a especificação de diretivas para a execução de teste que contém informação sobre a arquitetura utilizada na execução dos testes, incluindo a configuração de recursos para sistemas distribuídos. Utilizando esse mecanismo é possível especificar parâmetros de configuração para o teste de um conjunto restrito de arquiteturas. Esse mesmo mecanismo pode ser utilizado para executar múltiplas instâncias dos casos de teste, atuando como um teste de estresse gerado a partir dos testes funcionais criados pelo gerador de teste.

2.11.1.4 Geração de oráculo+++

Conforme mencionado anteriormente, critérios são utilizados para a geração de testes, incluindo a geração dos resultados esperados, uma vez que sem essa informação o teste não pode ter seu veredicto determinado. Isso torna necessária a existência de um oráculo, para a determinação dos resultados esperados em um teste. O oráculo é a parte mais cara e difícil dos métodos de TBM (Binder, 2000). Métodos de TBM deveriam especificar os requisitos que permitem a geração de um oráculo, preferencialmente indicando o custo/benefício relacionado ao uso prático desses requisitos. Além disso, é importante a existência de ferramentas de apoio aos métodos que implementem tais requisitos.

Requisito 6 Um método de TBM deveria prescrever mecanismos viáveis para automatizar a geração de um oráculo.

Apesar do TOTEM descrever aspectos relacionados a geração automática de oráculos, não existe uma ferramenta de apoio ao método que valide os aspectos descritos.

O AGEDIS também prescreve os requisitos que permitem a geração automática do oráculo. A máquina de execução do AGEDIS contém um oráculo gerado a partir da composição do modelo comportamental do sistema. Após a execução de um teste o estado do sistema é analisado para verificar sua corretude. Embora a abordagem seja descrita como genérica para qualquer tipo de aplicação ela parece ser proibitiva para certos domínios.

2.11.1.5 Documentação de teste++

A geração de teste utilizando critérios pode garantir um certo grau de cobertura, porém é necessário avaliar os testes gerados e sua execução para decidir sobre a necessidade de criação de testes adicionais. O julgamento humano é fundamental nessa decisão. Métodos de TBM deveriam prescrever a geração dos principais artefatos de teste para auxiliar essa avaliação. Independentemente dos processo utilizado durante o desenvolvimento, existem diversos artefatos diretamente relacionado às atividades de teste, conforme apresentado na Seção 2.5.

Requisito 7 Um método de TBM deveria prescrever a geração automática dos artefatos de teste.

O TOTEM não prescreve a geração de artefatos de teste.

O AGEDIS não especifica a geração de nenhum relatório compatível com os prescritos pelo IEEE, contudo a máquina de execução do AGEDIS armazena os resultados de uma execução para análise futura. A ferramenta de geração de relatório permite a criação de documentos de gerenciamento descrevendo os casos de teste, falhas, modelos e outros artefatos do processo de teste.

2.11.1.6 Avaliação do teste+++

A avaliação do teste está diretamente relacionada à atividade Verificação do término no PRAXIS (Subseção 2.1.2). O objetivo dessa atividade é avaliar os resultados obtidos com a execução dos testes para analisar a necessidade de planejamento, desenho e implementação de novos testes. A maioria dos métodos de TBM implementa mecanismos para essa avaliação. Isso é feito a partir de relatórios detalhando a cobertura alcançada utilizando critérios para a avaliação de cobertura. Diversas ferramentas comerciais incorporam mecanismos para ajudar essa avaliação, permitindo a criação de relatórios de teste configuráveis.

Requisito 8 Um método de TBM deveria prescrever mecanismos para a avaliação de testes.

O TOTEM não especifica mecanismos para auxiliar a avaliação da execução dos testes.

O AGEDIS prescreve como obter dados sobre a cobertura dos testes executados. O conjunto de ferramentas do AGEDIS é integrado com uma ferramentas de cobertura chamada FoCus[3]. Utilizando essa ferramenta é possível gerar relatórios de teste configuráveis.

2.11.1.7 Planejamento de testes++

O planejamento dos testes é a primeira atividade da disciplina de teste. Essa atividade requer a determinação da equipe de teste e da agenda para execução das demais atividades de teste.

Os métodos de TBM deveriam prescrever mecanismos para auxiliar o planejamento dos testes, possivelmente incluindo a identificação da complexidade do SST, e apontando as partes que merecem maior atenção durante as atividades de teste. Adicionalmente, métodos de TBM deveriam prescrever a existência de bases de dados históricas, para garantir o registro do esforço nas atividades, facilitando assim a realização de estimativas futuras.

Requisito 9 Um método de TBM deveria prescrever mecanismos para facilitar o planejamento de testes.

Os métodos considerados neste trabalho não possuem mecanismos para auxiliar o planejamento dos testes. No entanto, a atividade de planejamento do teste é fundamental e sua consideração sobre o escopo de TBM pode gerar benefícios muito interessantes.

[3] http://www.alphaworks.ibm.com/tech/focus

2.11.2 Requisitos do Engenheiro de Componentes

2.11.2.1 Suporte para manutenção e regressão+++

Diversos tipos de manutenção são possíveis durante o ciclo de vida de um sistema, como por exemplo manutenção corretiva, adaptativa e evolutiva (IEEE, 1990). As mudanças produzidas pela manutenção e por qualquer outra mudança no sistema, durante outras fases, podem tornar inúteis parte dos testes. É necessário identificar as partes afetadas para gerar testes adicionais, descartando as partes obsoletas. Um método de TBM deveria prescrever mecanismos para auxiliar a identificação das partes afetadas por manutenções e por mudanças no sistema.

Idealmente, todos os testes de regressão que não são afetados pela manutenção deveriam ser mantidos. No entanto, os recursos e o tempo disponível para o teste geralmente determinam que testes devem ser considerados. Um método de TBM deveria prescrever mecanismos para auxiliar a seleção de testes para compor uma bateria de regressão.

Requisito 10 Um método de TBM deveria prescrever mecanismos para auxiliar a manutenção e mudanças no sistema.

Os métodos de TBM considerados neste trabalho não prescrevem mecanismos para manutenção e mudanças no sistema. Essa é a grande diferença entre os requisitos identificados neste trabalho e os requisitos implementados pelos métodos analisados. Nas nossas seções de levantamento de requisitos identificamos a prioridade desse requisito como alta (+++). Nenhum dos métodos analisados consideraram tal requisito. Mudanças no sistema não são exclusivas da fase de manutenção; isso é normal durante todo o desenvolvimento, por isso a grande necessidade por esse tipo de apoio.

2.11.2.2 Interoperabilidade entre métodos++

A infraestrutura de teste, similarmente ao software, depende de tecnologias específicas. No caso de mudanças tecnológicas, os testes provavelmente devem ser atualizados, o que causa custos adicionais. Uma interessante alternativa para esse cenário é a criação de testes utilizando Modelos Independentes de Plataforma (MIP) e permitindo transformação para Modelos Específicos de Plataforma (MEP), conforme prescrito pela Arquitetura Dirigida por Modelos (MDA - Model Driven Architecture) (OMG, 2003a). Uma abordagem promissora para os métodos de TBM é seguir os conceitos MDA, gerando testes em um alto nível de abstração, independente de plataforma, e permitindo transformações entre esses formatos abstratos e tecnologias específicas.

A UML Testing Profile (OMG, 2003b) é uma iniciativa nessa direção. Ela define uma linguagem para projetar, visualizar, especificar, analisar, construir e documentar os artefatos relacionados ao teste de sistema. A UML Testing Profile é uma linguagem de modelagem de teste que pode ser utilizada com as maiores tecnologias e aplicada ao teste de sistema em vários domínios de aplicações.

Requisito 11 Um método de TBM deveria prescrever mecanismos permitindo a interoperabilidade entre métodos, ferramentas e linguagens.

O TOTEM não prescreve mecanismos relacionados à padronização de testes.

O AGEDIS desenvolveu uma linguagem de modelagem própria com dois grandes objetivos: dar suporte à criação de uma abstração do SST e estabelecer diretivas de teste significativas para o processo de teste. Essa linguagem é utilizada pelo AGEDIS para descrever o SST e o modelo de teste.

2.11.2.3 Geração de teste manual++

Por mais que sejam efetivos, os métodos de TBM não podem garantir sempre uma cobertura ideal. Existem diversos aspectos a serem considerados durante a geração de testes, como por exemplo complexidade, criticalidade, prioridade e tamanho. Adicionalmente às questões de cobertura, a avaliação de testes envolve seres humanos que podem sempre utilizar aspectos subjetivos para justificar a criação de testes adicionais.

É importante para métodos de TBM, independentemente da razão, a prescrição de mecanismos para apoiar a incorporação de novos testes, em adição aos testes automaticamente gerados. Uma forma comum de fazer isso, implementado por muitas ferramentas comerciais, é permitir a gravação de diversas ações durante o uso do sistema, gerando um script/programa capaz de reproduzir as ações gravadas. Uma outra alternativa é gerar os principais componentes de teste utilizando uma linguagem que suporte o reuso, permitindo assim a extensão do programas de teste para criar novos programas de teste.

Requisito 12 Um método de TBM deveria prescrever mecanismos para apoiar a incorporação de novos testes.

O TOTEM não prescreve o apoio para criação de testes adicionais.

O AGEDIS prescreve o apoio para a criação de testes adicionais a partir de uma ferramenta específica para essa tarefa.

2.11.3 Requisitos para o Testador do Sistema

2.11.3.1 Execução de teste automática+++

A execução automática de teste é responsável pela execução das ações prescritas nos procedimentos de teste utilizando os dados especificados nos casos de teste, comparando os resultados obtidos com os resultados esperados. Existem diversas ferramentas comerciais que apóiam essa atividade. No contexto de SI, a execução automática abrange diversos passos:

- Povoamento do mecanismo de armazenamento, utilizando os dados exigidos para a execução dos testes.
- Instanciação dos elementos para entrada de dados.

- Atribuição de dados aos respectivos elementos das interfaces gráficas.
- Execução das ações relacionadas ao teste.
- Verificação dos resultados, comparando os resultados obtidos com os resultados esperados.

Métodos de TBM deveriam prescrever todos os elementos exigidos para a execução automática, como por exemplo convenções de codificação e o uso de padrões. É também importante a capacidade de selecionar testes para criar baterias, assim como o agendamento da execução.

Requisito 13 Um método de TBM deveria prescrever a execução automática dos testes.

Apesar do fato do TOTEM relatar alguns aspectos relacionados à execução automática, não existe ferramenta de apoio ao método para validar os aspectos descritos.

O AGEDIS prescreve mecanismos que permitem a execução automática dos testes. Existe uma máquina de execução que utiliza o conjunto de testes abstratos e as diretivas de execução de testes para essa tarefa. Nem o MODEST nem o AGEDIS dão suporte para o agrupamento e agendamento de testes.

2.11.3.2 Acompanhamento de falhas+

A execução de teste pode encontrar diversas falhas, demandando apoio automático para o registro e acompanhamento dessas falhas. Os sistemas de acompanhamento de falhas permitem aos desenvolvedores efetivamente acompanhar as falhas registradas, facilitando as atividades de gerenciamento. A ausência de um mecanismo para acompanhamento de falhas pode por em perigo o TBM, por isso é interessante que os métodos de TBM ofereçam mecanismo para automatizar ou auxiliar essa tarefa.

Requisito 14 Um método de TBM deveria prescrever mecanismos de apoio para o registro e acompanhamento de falhas.

O TOTEM não prescreve apoio para o acompanhamento de falhas.

O AGEDIS prescreve o acompanhamento de falhas e registro automático de falhas encontradas durante a execução dos testes. Uma ferramenta de análise e registro de falhas foi criada no conjunto de ferramentas do AGEDIS.

2.11.4 Requisitos do Engenheiro de Processo

2.11.4.1 Aumento da qualidade e produtividade+++

O principal objetivo dos métodos de TBM é o aumento da qualidade dos produtos, preferencialmente com diminuição dos custos associados. A maioria dos métodos de TBM focaliza nessa questão e possuem estudos experimentais que mostram que existe uma redução no esforço necessário para as atividades de teste quando esses métodos são utilizados. Eles também

apresentam benefícios em termo de qualidade, criando testes com uma maior capacidade de detecção de erros. No entanto, um fato surpreendente é que poucos dos métodos de TBM com estudos experimentais levam em consideração os custos relacionados à criação do modelo do SST. Esse aspecto é um elemento crítico nessa avaliação, uma vez que o esforço para criação do modelo pode ser maior que o esforço para a execução manual das atividades de teste.

Embora esse requisito, assim como muitos outros discutidos aqui, possam ser considerados óbvios, existem muitos métodos de TBM que não os atendem.

Para aumentar produtividade e qualidade por meio do uso de métodos de TBM é importante que o esforço para criação dos modelos do SST possuam um bom custo/benefício. Para isso ações como o uso de uma linguagem de modelagem bem difundida, como a UML, poderia ajudar bastante. A restrição do método de TBM, para trabalhar com um domínio específico de aplicações também poderia auxiliar essa tarefa.

Requisito 15 Um método de TBM deveria diminuir os custos do desenvolvimento e aumentar a qualidade dos testes.

O TOTEM não descreve uma avaliação do seu uso.

O AGEDIS apresenta diversos estudos de caso que indicam que o uso do método pode trazer mais qualidade aos testes gerados. Não encontramos uma discussão sobre esforço relacionado ao uso do método.

2.12 Considerações finais

Neste capítulo apresentamos os principais referenciais teóricos ligados a este trabalho. Em particular, apresentamos na Seção 2.11 um conjunto de requisitos para métodos de TBM. Até onde sabemos, não existe um trabalho apresentando um catálogo de requisitos para métodos de TBM. Os requisitos existentes no catálogo podem ser utilizados como guia para o desenvolvimento ou avaliação de métodos e ferramentas.

Realizamos uma análise de dois trabalhos relacionados ao MODEST, e que nos influenciaram bastante, para verificar o nível de atendimento aos requisitos identificados. Pudemos notar que os métodos que possuem ferramentas de apoio, e que conseqüentemente puderam ser avaliados através de estudos de casos, atendem de forma mais efetiva os requisitos identificados. O uso de tais requisitos para avaliação de métodos de teste pode indicar o nível de benefício associado a adoção do método por parte de uma organização.

Quando iniciamos o desenvolvimento do MODEST estávamos convencidos da importância de criar um catálogo de requisitos para o TBM. As versões iniciais desse catálogo foram principalmente baseadas nos trabalhos discutidos aqui. Essas versões foram utilizadas para guiar o desenvolvimento do nosso método. Esse desenvolvimento, junto com nossa experiência no desenvolvimento de métodos e ferramentas de teste para a indústria e academia, e o auxílio de diversos desenvolvedores de SI, melhoraram o catálogo.

O desenvolvimento do nosso método e a análise dos outros dois métodos de TBM validam informalmente o catálogo, contudo, estamos estudando estratégias para validá-lo de forma

mais rigorosa. Acreditamos que uma validação mais formal do catálogo deve ser feita em instanciações para domínios específicos. Essas instanciações podem alterar as priorizações, enfraquecendo alguns requisitos e fortalecendo outros.

O catálogo de requisitos captura as necessidades, expectativas e restrições dos trabalhadores relacionados ao desenvolvimento de software. Apesar desse catálogo ter sido influenciado diretamente pelo nosso conhecimento prévio no desenvolvimento de SI, ele pode ser útil como base para outros domínios.

No próximo capítulo apresentamos o MODEST utilizando três perspectivas diferentes. Além disso, apresentamos uma breve comparação do MODEST com os outros dois métodos analisados neste capítulo.

2.13 Contextualização do trabalho

Neste trabalho focalizamos na automação dos testes realizados pelas equipes de teste, durante a construção do sistema. Particularmente, nossa pesquisa está associada a métodos e ferramentas que apóiem o teste de sistema, na forma de apoio à criação de testes funcionais, de desempenho e estresse, embora esses testes possam ser eventualmente utilizados na forma de testes de regressão.

Ao longo do trabalho definimos um catálogo de requisitos para o TBM e atualmente estamos refinando-o, no contexto de ferramentas de apoio às atividades de teste, utilizando o QFD.

No próximo capítulo apresentamos o método de TBM MODEST. Durante o desenvolvimento do MODEST criamos alguns critérios de teste baseados na especificação, além de termos utilizados critérios definidos em outros trabalhos. O MODEST é apresentado em três níveis diferentes: conceitual, lógico e físico. No nível conceitual apresentamos as prescrições do método em alto nível, enquanto que no nível lógico apresentamos nossas decisões de mapeamento para um processo de software e uma linguagem de modelagem. No nível físico apresentamos nossas decisões de implementação, correspondendo à ferramenta de apoio ao MODEST, chamada MODESToo. Exibiremos como o MODEST utiliza as várias visões da UML para fins de automação de teste. Exibimos ainda um estudo experimental realizado com o intuito de caracterizar o MODEST, em termos do custo associado ao seu uso e da qualidade dos testes gerados automaticamente. A avaliação de qualidade utilizou dois critérios de teste baseados em erros: semeadura de erros e análise de mutantes.

Capítulo 3

O Método MODEST

Neste capítulo apresentamos o MODEST. A sua apresentação é feita utilizando três níveis. Inicialmente fazemos uma descrição conceitual onde apresentamos os conceitos e características do método, em alto nível, abstraindo os detalhes relacionados à integração com uma linguagem de modelagem e um processo específico. Em seguida apresentamos uma descrição lógica onde exibimos as decisões utilizadas para atender as prescrições do método no contexto de processos baseados no Processo Unificado, onde a linguagem de modelagem adotada é a UML. Finalizando, apresentamos uma descrição física exibindo os aspectos específicos da implementação do método, detalhando as tecnologias utilizadas no desenvolvimento da sua ferramenta de apoio.

Apresentamos também neste capítulo um exemplo da personalização do processo PRAXIS para integração com o MODEST, detalhando as informações adicionais exigidas pelo método, mapeadas dentro das atividades do processo. Apresentamos ainda uma análise do MODEST com relação aos requisitos identificados no capítulo anterior, fazendo uma breve comparação dele com os outros dois métodos analisados.

Utilizamos a seguinte organização neste capítulo: a Seção 3.1 apresenta o modelo conceitual do método; a Seção 3.2 apresenta o modelo lógico; a Seção 3.3 apresenta o modelo físico; a Seção 3.4 apresenta a personalização do processo PRAXIS para integração com o MODEST; a Seção 3.5 apresenta uma análise do MODEST em relação aos requisitos para métodos de TBM descritos no capítulo anterior; e, finalizando, a Seção 3.6 apresenta as considerações finais do capítulo.

3.1 O Modelo Conceitual do MODEST

O MODEST é um método de TBM cuja sigla significa **M**eth**OD** to h**E**lp **S**ystem **T**esting. O principal objetivo do MODEST é automatizar as atividades relacionadas ao teste de sistema. Ele utiliza os conceitos associados ao TBM, utilizando modelos para gerar, executar e avaliar testes.

O MODEST pode ser integrado a um processo que prescreva testes de unidade e testes de integração, embora testes nesses níveis não sejam requisitos para uso do método. Assim,

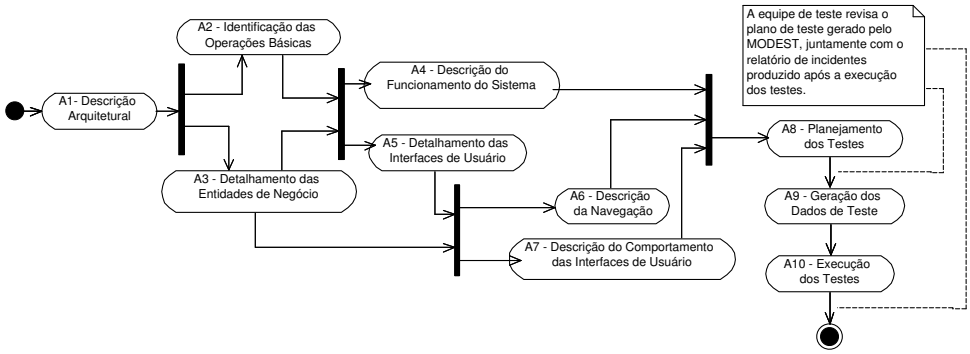

Figura 3.1: Diagrama de atividades do MODEST.

cada organização que venha adotar o método é livre para decidir como serão utilizados outros níveis de teste durante o desenvolvimento. O MODEST apenas se propõe a ajudar as tarefas relacionadas ao teste de sistema, automatizando os testes funcionais e prescrevendo mecanismos que podem também automatizar o teste de desempenho e estresse, embora essa parte ainda não esteja implementada.

O MODEST gera testes para verificar a corretude de uma implementação. Por causa disso assumimos que o modelo utilizado como entrada para o método está correto. A correção do modelo pode ser garantida por outros mecanismos como por exemplo revisões ou inspeções (Pressman, 2006). Apesar disso o MODEST prescreve algumas verificações de consistência que podem identificar eventuais erros de modelagem, conforme veremos a seguir.

A Figura 3.1 apresenta as atividades contidas no método. As atividades do MODEST são divididas em dois grandes grupos: (i) atividades de preparação e (ii) atividades de execução. As atividades A1 a A7 fazem parte do grupo de preparação e estão diretamente relacionadas à modelagem do comportamento do sistema. Essas atividades detalham quais informações devem ser modeladas para atender os requisitos de testabilidade impostos pelo método. As atividades A8 a A10 fazem parte do grupo de execução e prescrevem como gerar os procedimentos de teste, casos de teste e como executar os testes gerados. A seguir detalhamos cada uma das atividades, descrevendo as prescrições do método.

3.1.1 Descrição Arquitetural

Nesta atividade o MODEST prescreve a modelagem de alguns aspectos da arquitetura do SST, para que testes não-funcionais possam ser gerados. A versão atual do MODEST é apropriada para sistemas que seguem uma arquitetura composta por uma camada de apresentação, contendo as interfaces de usuário, uma ou mais camadas responsáveis pela implementação das regras de negócio, e um mecanismo de armazenamento abstraído por uma camada de persistência, conforme ilustrado na Figura 3.2. Essa estrutura, embora simples, está presente na arquitetura de muitos SI. O MODEST prescreve nesta atividade a modelagem do arranjo das camadas (apresentação, regras de negócio, persistência) nas instâncias dos nodos compu-

Figura 3.2: A arquitetura suportada pelo MODEST.

tacionais. O MODEST ainda prescreve que seja modelado, no nodo computacional associado à camada de apresentação, a quantidade de acessos concorrentes, possibilitando a realização de testes de desempenho e estresse.

Quando o número de usuários concorrentes suportado pelo sistema é fornecido, o MODEST prescreve a criação de testes atendendo ao seguinte critério:

- Critério de estresse: dado um conjunto de testes T e um modelo D descrevendo o sistema, T deve incluir testes que executem o sistema, com a quantidade de usuários concorrentes especificada em D, e por ordens de grandezas superiores ao especificado em D, até que o sistema não consiga atender corretamente as requisições enviadas.

Observe que o critério de estresse não define os testes a serem executados. Ele apenas prescreve características que os testes devem ter, na forma de uma diretiva para a geração de teste. Conforme descrevemos nas próximas seções, é necessário decidir como essa prescrição será atendida no domínio de uma linguagem de modelagem e de um processo de software (nível lógico), além de detalhar como a implementação do critério será realizada, em termos de tecnologias (nível físico).

3.1.2 Identificação das Operações Básicas

Nesta atividade o MODEST prescreve a modelagem das operações básicas, relacionadas às operações *CRUD* (**C**reate, **R**ead, **U**pdate, **D**elete) usualmente providas por uma camada de persistência. Essa modelagem deve ser realizada de forma a tornar possível a identificação dessas operações dentro dos roteiros associados às funcionalidades do sistema. Conforme descrito na Subseção 3.1.4, a modelagem das funcionalidades do SST é uma das prescrições do MODEST. Nessa modelagem das funcionalidades do sistema deve ser possível a identificação das operações de persistência utilizadas.

A identificação dessas operações permite, por exemplo, inferir qual operação precisa ser invocada para criar ou remover dados persistentes, assim como permite identificar os resultados esperados de um teste, visto que a execução das operações pode resultar em mudanças nos dados persistentes. A execução da operação *Create*, por exemplo, geralmente causa a criação de novos dados no mecanismo de armazenamento, assim como a execução da operação *Delete* geralmente causa a remoção de dados.

3.1.3 Detalhamento das Entidades de Negócio

Nesta atividade o MODEST prescreve a modelagem das entidades de negócio e dos seus relacionamentos. Essas entidades geralmente representam informação persistente e são necessárias para cumprir alguma responsabilidade do produto. A partir dessa descrição é possível entender a estrutura dos dados persistentes suportados pelo SST, possibilitando assim seu manuseio de forma automática.

Além da prescrição sobre a modelagem das entidades do negócio, o MODEST também prescreve a especificação de algumas propriedades relacionadas aos atributos dessas entidades. Essas propriedades devem detalhar se o atributo é uma chave na identificação da entidade, seu tipo, valores válidos, valores mínimo e máximo, e se o atributo é obrigatório.

Conforme mencionado na Subseção 3.1.5, é possível criar uma ligação entre um elemento de modelo representando um atributo de uma entidade de negócio e um elemento de modelo representando o campo de uma interface de usuário, habilitando assim a geração de testes utilizando a informação sobre o formato do atributo referenciado.

O MODEST prescreve ainda a especificação de algumas propriedades não-obrigatórias relacionadas às entidades do negócio. Essas propriedades podem ser utilizadas para geração de testes não-funcionais. Essas propriedades são: (i) cardinalidade média e (ii) tempo máximo de processamento de transações. Caso tais informações sejam modeladas, o MODEST prescreve a verificação do tempo de processamento de transações de uma entidade, estando essa povoada com a cardinalidade média especificada. Isso é traduzido na forma de um outro critério definido pelo MODEST:

- Critério de desempenho: dado um conjunto de teste T e um modelo M descrevendo o SST, T deve executar o SST utilizando as entidades de negócio de M, povoadas com a cardinalidade média indicada em M, verificando se o tempo de execução das operações CRUD para a entidade é menor que o tempo máximo informado para processamento de transações.

De forma semelhante ao critério de estresse, o MODEST não define quais testes devem ser executados no critério de desempenho. Ele apenas prescreve características que os testes devem ter. As instanciações do método devem definir como esse critério será implementado.

3.1.4 Descrição do Funcionamento do Sistema

Nesta atividade o MODEST prescreve a descrição das funcionalidades do sistema, por meio da modelagem das interações entre os elementos que implementam uma determinada funcionalidade.

O MODEST especifica algumas regras para determinar se uma notação de modelagem pode ser utilizada para esse fim. Essa notação deve ser expressiva o suficiente para permitir a criação dos procedimentos de teste. Para isso, a descrição feita com essa notação deve permitir a identificação dos dados relacionados à funcionalidade e dos comandos acionados para execução dessa funcionalidade. Por exemplo, para autenticar um usuário em um sistema

geralmente é necessário fornecer um identificador de usuário e uma senha, além de acionar o comando para entrar no sistema. Nesse caso, o modelo gerado com a descrição dessa funcionalidade deve deixar claro quais dados são utilizados nessa funcionalidade (identificador e senha), assim como o comando acionado (entrar no sistema).

Além das informações mencionadas anteriormente, todas as eventuais restrições associadas à funcionalidade sendo descrita devem ser modeladas. Também devem ser modeladas as operações de persistência executadas na funcionalidade. Ainda utilizando o exemplo da funcionalidade para autenticação em um sistema, uma possível restrição associada a essa funcionalidade poderia ser: se for fornecido um identificador de usuário não existente, uma mensagem indicando que esse identificador não existe deve ser exibida. A operação de persistência associada a essa funcionalidade é a operação *Read*, visto que dados deverão ser lidos do mecanismo de armazenamento para comparação com os dados informados pelo usuário.

A descrição das restrições relacionadas à funcionalidade geralmente exige o uso de uma linguagem formal. O MODEST prescreve que a linguagem utilizada para tal descrição seja livre de efeitos colaterais e que a avaliação de suas expressões sempre resultem em um valor. Isso garante a possibilidade de interpretação dos predicados expressos pela linguagem utilizada, tornando possível a automação de algumas atividades diretamente relacionadas à interpretação de restrições.

3.1.5 Detalhamento das Interfaces de Usuário

Nesta atividade o MODEST prescreve o detalhamento das interfaces de usuário. Essas interfaces são utilizadas para a entrada e saída de dados em um sistema de informação. Por causa disso, o MODEST prescreve o mapeamento dos elementos das interfaces de usuário visíveis para os usuários finais, com o intuito de tornar possível a geração de testes envolvendo tais elementos.

Os elementos das interfaces de usuário podem ser campos ou comandos. Os campos são elementos utilizados para entrada ou saída de dados. Os comandos são elementos utilizados para iniciar (ou continuar) a execução de uma das funcionalidades do sistema. O MODEST prescreve o uso da especificação de diversas características associadas a campos e comandos para tornar possível a geração e execução dos casos de teste. Um campo deve detalhar o estilo usado para sua implementação na linguagem alvo (*TextField*, *PasswordField*, *CheckBox*, etc.), juntamente com a fonte e o destino dos dados. Por exemplo, a fonte de um campo pode ser um atributo de uma entidade de negócio, um valor pré-definido, como por exemplo a data do sistema, ou um valor calculado, como a soma de dois atributos de uma entidade de negócio.

Um comando de uma interface de usuário deve definir o estilo usado na linguagem alvo (como *Button* ou *MenuItem*), as pré-condições exigidas para a ativação do comando, e as pós-condições indicando o estado do sistema após sua execução. O MODEST não determina uma linguagem para especificação dessas condições. Qualquer linguagem pode ser usada, desde que tenha o poder de expressão adequado e que atenda às mesmas restrições descritas na subseção anterior: a linguagem deve ser livre de efeitos colaterais e a avaliação de todos os predicados expressos nessa linguagem deve resultar em um valor.

3.1.6 Descrição da Navegação

Nesta atividade o MODEST prescreve a modelagem das possíveis transferências de controle entre as interfaces de usuário. Tal informação deve ser descrita de forma que seja possível navegar por todo o sistema, visto que os usuários realizam solicitação de operações por meio das interfaces de usuário. Caso haja alguma condição relacionada à navegação, ela deve ser explicitamente descrita, utilizando-se para isso uma linguagem formal, seguindo as restrições apresentadas na subseção anterior. O MODEST prescreve que o Critério de navegação seja utilizado na geração de testes:

- Critério de navegação: dados um conjunto de teste T e um modelo M, T deve conter testes que exercitem todas as transferências de controles existentes em M.

3.1.7 Descrição do Funcionamento das Interfaces de Usuário

O comportamento esperado de um SI está diretamente relacionado à interação dos seus usuários com as interfaces de usuário. Nós assumimos que em um desenho típico as exceções são sempre associadas com mensagens exibidas nas interfaces de usuário. Nós também assumimos que as exceções são associadas a cláusulas que devem ser explicitamente modeladas.

Nesta atividade o MODEST prescreve que as interfaces de usuário tenham seu comportamento descrito. A descrição do comportamento de uma interface de usuário é a modelagem das diferentes formas de exibição que elas podem assumir, com diferentes configurações de habilitação e visibilidade de campos e comandos, juntamente com os estímulos que causam mudanças nessa exibição. Além disso, também consideramos como parte do funcionamento das interfaces de usuário a modelagem de todas as mensagens que podem ser exibidas para os usuários do sistema

Assim como a própria descrição do comportamento do sistema, não existe um formato específico para essa descrição. Qualquer mecanismo pode ser utilizado, desde que seja possível identificar os diferentes formatos de exibição da interface de usuário, indicando os campos e comandos ativos, inativos e invisíveis, assim como os eventos que causam mudanças nessa exibição, detalhando as condições associadas e, quando possível, ações resultantes. O MODEST prescreve que todos os formatos modelados sejam exercitados, assim como todas as transições possíveis entre formatos e todas as mensagens exibidas ao usuário.

3.1.8 Planejamento dos Testes

Nesta atividade o MODEST prescreve o planejamento dos casos de teste, ou seja, os casos de teste são criados, associando-os às diversas partes do SST a serem exercitadas, mas sem determinar os dados de entrada e os resultados esperados. A geração dos dados de entrada e saídas esperadas é prescrita na próxima atividade.

As seqüências de casos de teste geradas a partir dessa atividade irão compor o *Plano de Testes* do sistema. O planejamento dos casos de teste é um ponto muito relevante no

MODEST, uma vez que ele prescreve a seguinte cobertura da especificação: devem ser exercitadas as diferentes formas de exibição das interfaces de usuários, todos os procedimentos de testes, todas as condições excepcionais presentes no sistema e as navegações entre as interfaces de usuário.

Algorítmo 1 Planejamento dos testes (nível conceitual).

```
Procedimento PlanejadorDeTestes
  para cada funcionalidade do sistema fs faça
    PlanejadorDeTestesDeFuncionalidades(fs)
  fim para

Procedimento PlanejadorDeTestesDeNavegação(GUI
  t)
  para cada navegação n possível a partir de t faça
    ct ← novo caso de teste para exercitar n
  fim para

Procedimento PlanejadorDeTestesDeTelas(formato de
  exibição fe)
  se fe já foi marcada como exercitado então
    abandone
  fim se
  para cada possível transição t a partir de fe faça
    c ← comando associado à transição t
    pt ← procedimento de teste relacionado ao comando
    c
    ct ← novo caso de teste para exercitar t usando pt
    marque a transição t como exercitada
    nfe ← formato de exibição alcançado a partir de t
    se nfe ≠ fe então
      PlanejadorDeTestesDeTelas(nfe)
    fim se
  fim para
  marque o formato de exibição fe como exercitado

Procedimento PlanejadorDeTestesDeFuncionalidades
  (funcionalidade do sistema fs)
  para cada GUI t relacionada a fs faça
    marque todos os possíveis formatos de exibição de t
    como não exercitados
    marque todas as transições para outros possíveis for-
    matos de exibição de t como não exercitados
    fei ← formato de exibição inicial de t
    PlanejadorDeTestesDeTelas(fei)
    PlanejadorDeTestesDeNavegação(t)
  fim para
```

O Algoritmo 1 detalha o planejamento dos testes. O MODEST prescreve que seja analisada a descrição do comportamento do sistema, verificando as interfaces de usuário associadas. Para cada interface de usuário, o MODEST prescreve que sejam exercitadas todas as possíveis formas de exibição, por meio da execução das ações associadas às mudanças nessa exibição, conforme descrito na Subseção 3.1.7. Tais ações devem estar relacionadas aos procedimentos de testes gerados a partir da descrição do comportamento do sistema (Subseção 3.1.4). Após o planejamento dos testes para uma interface de usuário, cobrindo todas suas diferentes formas de exibição e condições excepcionais, o MODEST prescreve o planejamento de testes que exercitem todas as navegações possíveis a partir da tela atual, conforme descrito na Subseção 3.1.6.

3.1.9 Geração de Dados de Teste

Nesta atividade o MODEST prescreve a geração das entradas e saídas para cada um dos casos de teste planejados. Tal geração deve ser feita por meio da interpretação das condições envolvidas nos testes planejados, caso haja uma condição. Por isso existe a prescrição do uso de uma linguagem para descrever as condições associadas ao SST. Tal tipo de linguagem e descrição são discutidos à frente, no nível lógico e físico.

É importante frisar que nem todas as linguagens podem ser utilizadas para a descrição das condições nos modelos descrevendo o software. O MODEST prescreve o uso de linguagens com poder de expressão adequado e que possam ser interpretadas para fins de geração de testes. Para diversas linguagens isso não é verdade, visto que em alguns casos a interpretação de expressões de uma linguagem pode ser um problema indecidível (Papadimitriou, 1993). A

linguagem utilizada deve ser livre de efeitos colaterais e a avaliação de suas expressões deve sempre resultar em um valor. Isso garante a possibilidade de interpretação dos predicados expressos utilizando a linguagem.

O Algoritmo 2 descreve a geração dos dados de teste. O procedimento de teste associado ao caso de teste, descreve, dentre outras coisas, as operações de persistência executadas durante o seu uso, assim como todas as entradas necessárias para sua execução. A partir da informação dos dados de entrada exigidos para o teste é possível saber o que precisa ser gerado. A análise da condição associada ao teste apenas indica como os dados associados à condição precisam ser gerados. Caso não haja uma condição associada ao teste a ser gerado, então os dados para teste podem ser gerados sem restrições, basta utilizar a especificação do seu formato. Para cada dado a ser gerado é necessário implementar os critérios prescritos pelo MODEST. Nesse caso, para um caso de teste planejado, diversos casos de teste podem ser gerados, pois a geração de testes utilizando critérios geralmente requer a criação de diferentes entradas.

Algorítmo 2 Geração de dados de teste (nível conceitual).

Procedimento GeradorDeDadosDeTeste
  para cada caso de teste ct planejado **faça**
    se existe um condição c associada a ct **então**
      gere entradas baseadas na condição c utilizando os critérios prescritos pelo MODEST
      gere dados para as outras entradas associadas ao procedimento de teste do caso de teste ct utilizando os critérios prescritos pelo MODEST
      gere dados persistentes a partir da condição c utilizando os critérios prescritos pelo MODEST
    senão
      gere dados para as entradas associadas ao procedimento de teste do caso de teste ct utilizando os critérios prescritos pelo MODEST
    fim se
    se se existe pós-condição associada aos comandos a serem executados **então**
      gere os resultados esperados baseados nas pós-condições existentes
    fim se
  fim para

Utilizando a descrição dos campos da interface de usuário (Figura 3.10) podemos identificar seus relacionamentos com as entidades de negócio (Figura 3.5), permitindo assim a geração de dados seguindo diferentes critérios. O MODEST prescreve que durante a geração dos teste sejam utilizados critérios como por exemplo Particionamento em classes de equivalência e Análise de valor limite (Binder, 2000), além dos critérios já definidos pelo próprio MODEST.

No nível conceitual, o MODEST especifica a geração de testes utilizando alguns critérios, conforme mencionado nesta seção. No nível lógico, quando é definida a linguagem de modelagem e o processo de software específico, é possível definir novos critérios.

3.1.10 Execução dos Testes

O MODEST prescreve a execução automatizada dos casos de teste, com registro de falhas caso elas aconteçam. O Algoritmo 3 descreve a execução dos testes. A execução consiste na

realização de diversos passos, tais como:

- povoamento do mecanismo de armazenamento, com dados necessários ao teste;
- carga e instanciação dos elementos para entrada de dados;
- atribuição dos dados especificados nos testes nos respectivos elementos das interfaces de usuário;
- execução das ações associadas ao teste;
- determinação de um veredicto para o teste, com base nos resultados obtidos.

Conforme mencionado anteriormente, o MODEST prescreve a geração de um veredicto para cada caso de teste executado. Esse veredicto basicamente indica se o houve sucesso ou categoriza a ocorrência de insucessos na execução do caso de teste. Uma especificação para veredictos é discutida no nível lógico.

Algorítmo 3 Execução dos testes (nível conceitual).

```
Procedimento ExecutorDeTestes
  para cada caso de teste ct faça
    para cada dado persistente d associado ao caso de teste ct faça
      crie o dado persistente d
    fim para
    crie uma instância das GUI associadas a ct
    para cada entrada especificada em ct faça
      atualize o campo da GUI com o valor especificado
    fim para
    acione o comando associado ao procedimento de teste pt de ct
    veredicto ← aprovado
    se forma de apresentação resultante não é a forma de apresentação associada a ct então
      veredicto ← reprovado
    fim se
    para cada resultado esperado re especificada em ct faça
      obtenha o valor v do campo
      se v ≠ re então
        veredicto ← reprovado
      fim se
    fim para
    para cada operação de persistência invocada em pt faça
      verifique o resultado das operação de persistência
      se existe algum resultado inconsistente então
        veredicto ← reprovado
      fim se
    fim para
    grave o veredicto do teste
  fim para
```

Os procedimentos de teste utilizados para executar um caso de teste definem quais entradas devem ser realizadas e quais comandos devem ser invocados. Os procedimentos de teste também especificam quais operações de persistência são utilizadas na execução do teste.

Na concepção do MODEST assumimos que o desenvolvedor do SST não modela ações artificiosas ou desnecessárias. Por exemplo, assumimos que se existe uma operação *Read* modelada na descrição de uma funcionalidade os resultados dessa operação irão atualizar um ou mais campos da interface envolvida, e, se existe uma operação *Delete* será removida, do mecanismo de armazenamento, a entidade de negócio correspondente. Com isso, conhecendo as operações de persistência invocadas, é possível determinar o estado do sistema após a execução do teste. Para isso o MODEST prescreve a seguinte convenção, que é implementada no algoritmo por meio do passo "verifique o resultado da operação de persistência":

- Se o procedimento de teste executado possui uma operação *Read*, os campos das interfaces de usuário associadas a atributos das entidades de negócio envolvidas na leitura devem ser atualizados com os valores obtidos do mecanismo de armazenamento.
- Se o procedimento de teste executado possui uma operação *Create* ou *Update*, os atributos das entidades de negócio associadas às interfaces de usuário envolvidas na criação ou atualização devem ser atualizadas com os dados provenientes dos campos das interfaces de usuário.
- Se o procedimento de teste executado possui uma operação *Delete*, as entidades de negócio associadas à interface de usuário envolvida na execução do procedimento de teste devem ser removidas.

As regras descritas acima correspondem a parte principal do oráculo do MODEST. Utilizando essas regras é possível verificar se o estado geral do sistema, após a execução de um teste, continua consistente, possibilitando assim a determinação de um veredicto para o teste.

3.2 O Modelo Lógico do MODEST

Nesta seção apresentamos a descrição lógica do MODEST, onde exibimos as decisões utilizadas para atender as prescrições do método no contexto de processos de software baseados no Processo Unificado, utilizando a UML como linguagem de modelagem.

3.2.1 Descrição arquitetural

No nível lógico, o MODEST prescreve o uso da visão de desdobramento da UML para a descrição arquitetural. A Figura 3.3 apresenta um exemplo dessa descrição. Cada elemento integrante da arquitetura do sistema deve ser representado por um nodo computacional. Para que testes de estresses sejam automatizados pelo MODEST, o nodo representando a camada de apresentação deve ter o estereótipo ≪*boundary*≫ e deve descrever também o número de usuários concorrentes suportados pelo sistema. Com isso, caso seja solicitado a execução de

testes de desempenho e estresse, é possível iniciar várias sessões de execução dos testes, verificando se o tempo de resposta às operações ocorre dentro dos limites especificados, conforme veremos na Subseção 3.2.3.

Abaixo apresentamos o mapeamento do critério de estresse do nível conceitual, para o nível lógico.

- Critério de estresse: dado um conjunto de testes T e um modelo descrevendo o sistema M, T deve incluir testes que executem o sistema, com a quantidade de usuários concorrentes especificada em M, no nodo computacional com estereótipo ≪*boundary*≫, e por ordens de grandeza superiores ao especificado, até que o sistema não consiga atender corretamente as requisições enviadas.

Conforme mencionado anteriormente, se o desenvolvedor fornecer o parâmetro "usuários concorrentes"o MODEST prescreve a execução de testes de estresse. A implementação do critério de estresse pode ser realizada por meio da execução concorrente do conjunto de testes gerado segundo as prescrições do MODEST. O número de execuções simultâneas deve seguir o parâmetro "usuários concorrentes"especificado no sistema. Uma vez que os testes tenham sido executados com sucesso, a ordem de grandeza representando o número de usuários concorrentes deve ser aumentada em uma ordem, como por exemplo de 100 para 1.000, e os testes devem ser executados novamente. Isso deve ser repetido até que o sistema não seja capaz de responder corretamente as requisições, seja por causa de tempo limite atingido ou por causa de respostas diferentes do resultado esperado. Conforme mencionado no Capítulo 5, essa parte do MODEST ainda não foi desenvolvida e já visualizamos possibilidades de modelagens alternativas para esse aspecto.

3.2.2 Identificação das operações básicas

No nível lógico, o MODEST prescreve o uso da visão de interação da UML para descrever as operações CRUD do mecanismo de persistência, utilizando diagramas de interação. Esses diagramas devem seguir convenções de nomeação e de formato, facilitando assim a identificação do mecanismo de invocação das operações de persistência. As convenções utilizadas não somente facilitam a identificação dessas operações, como também aumentam a qualidade da documentação do projeto, pois detalham como o mecanismo de persistência funciona.

Como exemplo apresentamos o diagrama de seqüência da Figura 3.4, utilizado para descrever a operação Update. Tal diagrama deve estar contido em um pacote com estereótipo ≪*persistence*≫. O primeiro método invocado deve detalhar como usar a operação CRUD sendo descrita, devendo haver um diagrama para cada operação, com o nome do diagrama condizendo com a operação descrita. A partir desse diagrama podemos notar que a operação de persistência responsável pela alteração de valores de um objeto é invocada a partir da chamada ao método *update* de um *PersistentObject*. Assim, qualquer invocação ao método update de um PersistentObject indica uma atualização nos seus dados. Essa inferência é utilizada para determinar o resultado esperado de algumas operações realizadas durante os

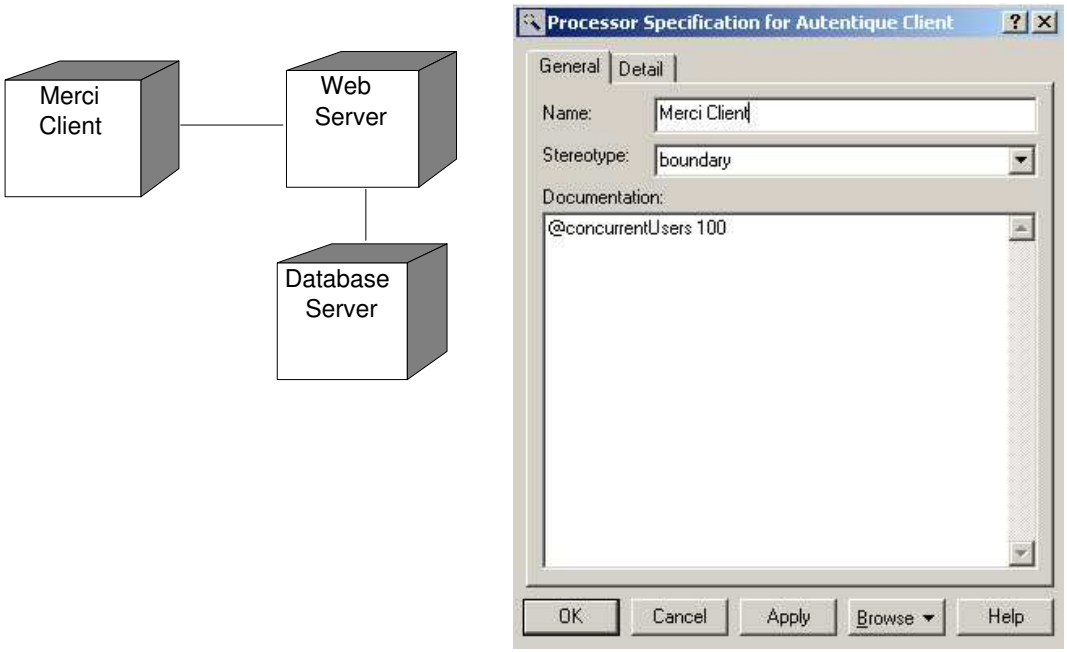

Figura 3.3: Diagrama de desdobramento com dados prescritos pela MODESTOo.

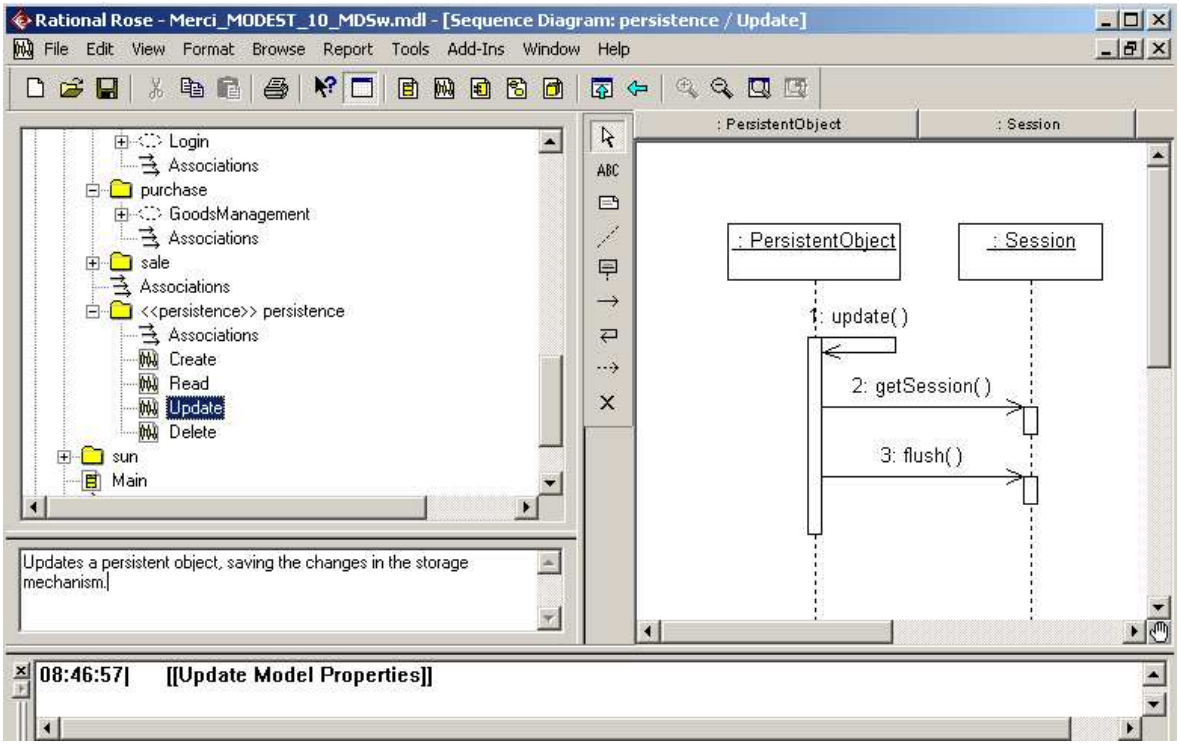

Figura 3.4: Descrição da operação de persistência *Update*.

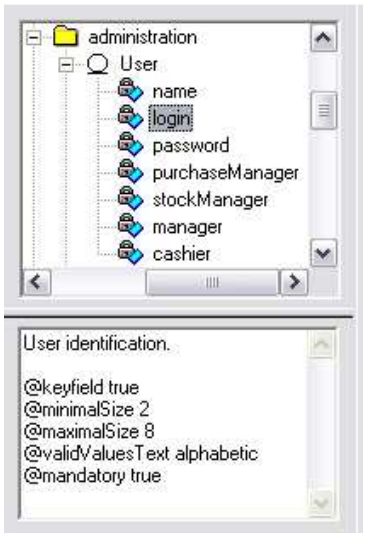

Figura 3.5: Propriedades relacionadas ao atributo *login* da entidade *User*.

testes. A partir do nome do diagrama descrevendo uma operação de persistência o MODEST pode inferir que operação está sendo descrita.

3.2.3 Detalhamento das Entidades de Negócio

No nível lógico, o MODEST prescreve o uso da visão estática da UML para descrição das classes persistentes, utilizando diagramas de classe com a especificação de alguns elementos adicionais. Assim, o MODEST exige que sejam criados diagramas de classes mostrando todas as associações entre classes. Além disso, as classes persistentes devem ser estereotipadas utilizando o estereótipo ≪*entity*≫, uma vez que esse estereótipo identificará tais classes. Os atributos das classes devem detalhar algumas propriedades adicionais, conforme convenção parcialmente exibida na Figura 3.5. Nesse caso apresentamos as propriedades relacionadas ao atributo *login* da classe persistente *User*, utilizando um formato semelhante à programação orientada a atributos (Wada et al., 2005). Esse mesmo formato é utilizado para descrever as propriedades não-obrigatórias das classes persistentes, para a geração de testes não-funcionais. As propriedades "@averageCardinality"e "@transactionProcessingTime"devem ser especificadas no campo para documentação da classe, modelando, respectivamente, a cardinalidade média de uma classe de entidade e o tempo máximo para execução de operações CRUD envolvendo tal entidade.

Na Seção 2.10, apresentamos os critérios de teste definidos por Andrews et al. (2003) para diagramas de classe da UML. Conforme apresentamos na Seção 3.2, o MODEST prescreve a geração de teste utilizando esses critérios. Além disso, o MODEST prescreve a utilização de outros critérios durante a geração de testes, sendo um deles o critério de composição:

- Critério de composição: dado um conjunto de teste T, um modelo M descrevendo o SST e uma classe de entidade E com o conceito de composição, T deve conter testes que

causem a remoção de uma instância de E que represente o *todo*, possuindo esse *partes*, verificando se houve a remoção das *partes* associadas a E.

Para que a implementação dos critérios de Andrews et al. (2003) seja possível, convencionamos que as multiplicidades representativas seriam os limites inferiores e, no caso de multiplicidades com um lado n, utilizamos um valor pré-definido no MODEST, que atualmente é cinco.

Uma classe de entidade pode possuir associações com outras entidades ou pode ser composta por entidades consideradas partes. O uso desse critério para geração de testes garante que as entidades com o conceito todo-parte foram corretamente implementadas, e uma remoção do todo implica na remoção das partes associadas. Note que embora essa verificação seja considerada trivial, esse critério não foi definido, por exemplo, no trabalho de Andrews et al. (2003), nem em outros trabalhos utilizados como referência (McQuillan e Power, 2005). É uma contribuição do MODEST a formalização desse critério, assim como dos outros aqui apresentados. Apesar disso, sabemos que é necessária uma avaliação do critério levando em consideração a capacidade de detecção de falhas e o custo do seu uso, embora ainda não tenhamos planejado isso.

O mapeamento do critério de desempenho descrito no nível conceitual é praticamente direto no nível lógico, conforme descrito a seguir:

- Critério de desempenho: dado um conjunto de teste T e um modelo M descrevendo o SST, T deve causar a execução de testes que utilizem as classes de entidades de M, cada uma delas povoadas com a cardinalidade média indicada em M, com o número de usuários concorrentes especificados em M, verificando se o tempo de execução das operações CRUD para a entidade é menor que o tempo máximo para processamento de transações informado.

Esse critério pode ser implementado de forma similar à implementação do critério de estresse, por meio da execução concorrente de diversas baterias de testes. O número de execuções simultâneas deve seguir o parâmetro de usuários concorrente especificado no sistema e as classes persistentes devem ser povoadas com a cardinalidade especificada na propriedade "@averageCardinality". Os testes gerados automaticamente devem então verificar se o tempo da execução das operações CRUD para as classes é inferior ao tempo especificado na propriedade "@transactionProcessingTime". De forma semelhante à implementação do critério de estresse, enfatizamos que essa parte da ferramenta ainda não foi desenvolvida e já visualizamos possibilidades de modelagens alternativas para esse aspecto.

3.2.4 Descrição do Funcionamento do Sistema

Uma colaboração em UML descreve como um caso de uso é implementado, exibindo as interações de objetos que implementam um comportamento dentro de um contexto Rumbaugh et al. (1999). O MODEST prescreve o uso de casos de uso e colaborações para descrever o funcionamento do sistema.

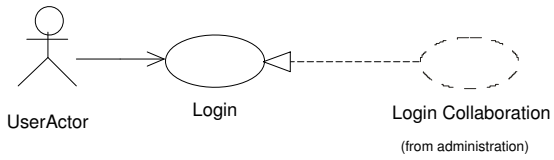

Figura 3.6: Exemplo de caso de uso e sua colaboração.

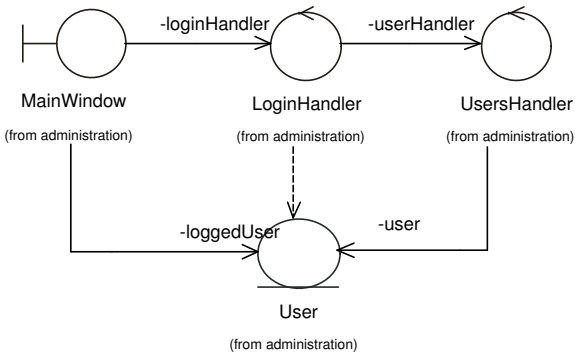

Figura 3.7: Parte estática da colaboração Login.

As colaborações possuem uma parte estática e uma parte dinâmica. Na parte estática, a colaboração descreve os papéis que objetos podem desempenhar em uma instanciação da colaboração. Na parte dinâmica a colaboração descreve uma ou mais interações que mostram o fluxo de mensagens sobre o tempo. As interações da parte dinâmica das colaborações são descritas na visão de interação, por meio de diagramas de seqüência ou de colaboração.

A Figura 3.6 apresenta uma colaboração para o caso de uso Login. A parte estática da colaboração é apresentada na Figura 3.7. Conforme prescrição do MODEST, devem existir mecanismos para identificar as interfaces de usuário, representadas por classes de fronteiras neste nível, associadas a um caso de uso. A parte estática da colaboração permite identificar não só as classes de fronteiras associadas, mas também as diversas outras classes envolvidas.

A parte dinâmica da colaboração Login é apresentada na Figura 3.9. Nela podemos notar que a primeira operação invocada corresponde a um comando da instância da classe de fronteira *MainWindow*. Os dados exigidos para execução dessa operação são *login* e *password*. Existe a descrição das diversas exceções que podem ser lançadas durante o roteiro, além de ser fácil identificar a operação persistente invocada, pois existe uma chamada para o método *Read* do objeto *User*, que é uma especialização de *PersistentObject*. Nesse exemplo, assim como nos demais locais em que o uso de uma linguagem formal é exigida, o MODEST convenciona o uso de uma linguagem simples, baseada na camada de persistência que já utilizamos em diversos projetos desenvolvidos (Santos-Neto et al., 2005b). A linguagem atende as restrições impostas pelo MODEST, sendo livre de efeitos colaterais e todas suas expressões podem ser avaliadas, resultando em um valor. Na Tabela 3.8 apresentamos um excerto dessa linguagem, que foi escolhida por ser enxuta, facilitando sua implementação. Conforme discutiremos no Capítulo

```
COMP ⇒< | <= | > | >= | == |! =
EXP ⇒ FIELD | ENTITY | NUMBER | STRING | NULL | EXP OP EXP
OP ⇒ +| − | ∗ |/
FIELD ⇒ <List of user interface fields>
ENTITY ⇒ ENT_ID.read(EXP)OTHER | ENT_ID.getFirst()OTHER |
ENT_ID.getLast()OTHER
OTHER ⇒ ξ | .GET_METHODS | .GET_OTHERS
ENT_ID ⇒ <List of system entities>
GET_METHODS ⇒ getATTRIBUTE()
GET_OTHERS ⇒ getPrevious() | getNext()
ATTRIBUTE ⇒ <List of entity attributes, starting with a capital letter>
```

Figura 3.8: Excerto da linguagem utilizada pelo MODESTtoo para definição de restrições.

5, pretendemos utilizar a linguagem OCL (Warmer e Kleppe, 2003) em futuras versões do MODEST.

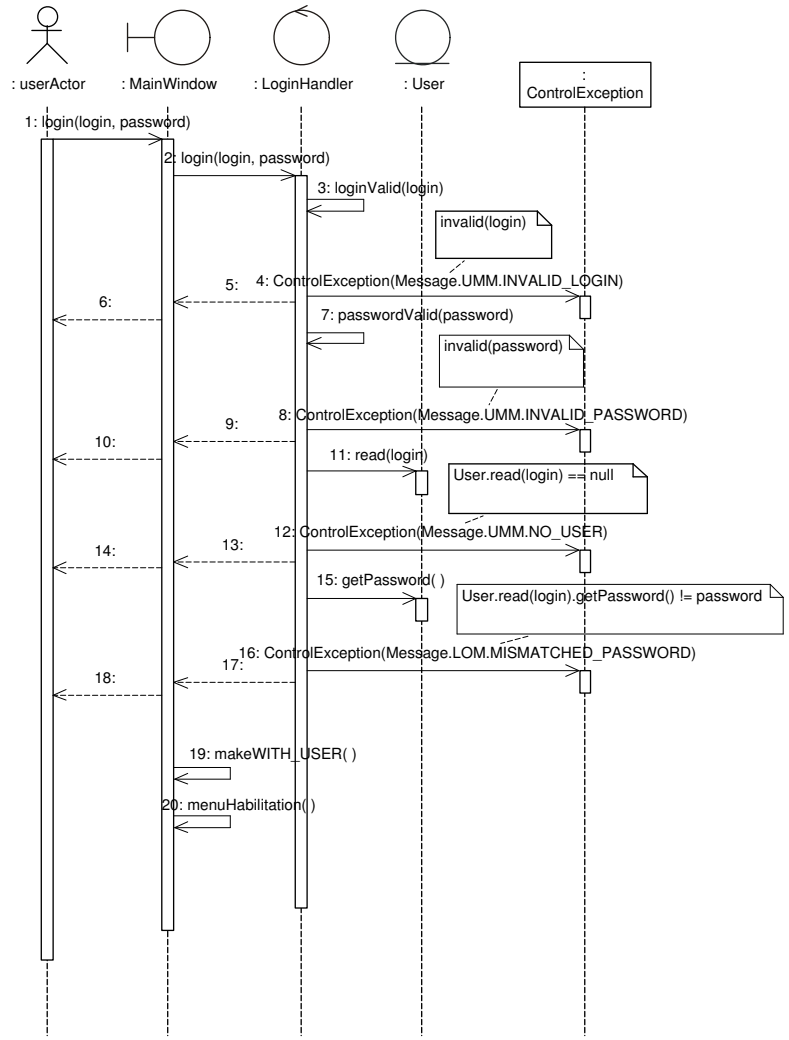

Figura 3.9: O roteiro Login.

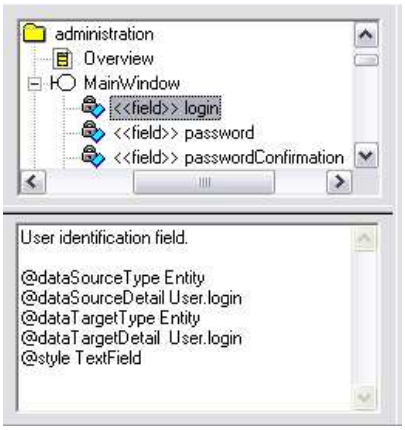

Figura 3.10: Propriedades do campo *login*.

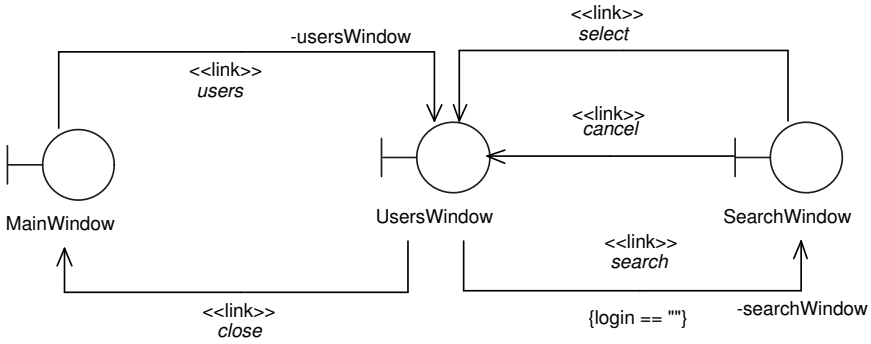

Figura 3.11: Exemplo de transferências de controle entre telas.

3.2.5 Detalhamento das Interfaces de Usuário

No nível lógico o MODEST prescreve que as interfaces de usuários sejam representadas como classes de fronteira, que são classes estereotipadas com o estereótipo ≪*boundary*≫. Isso facilita seu reconhecimento e extração dos dados relacionados. Adicionalmente, o MODEST prescreve o uso do estereótipo ≪*field*≫ nos atributos das telas, representando os campos, e o estereótipo ≪*command*≫ nas operações da tela, representando os comandos. A Figura 3.10 apresenta a descrição de uma interface de usuário, exibindo os detalhes referentes ao campo *login*. Podemos notar que existe uma ligação entre o campo *login* da interface de usuário e o atributo *login* da classe *User*. De forma similar à descrição das classes persistentes, o MODEST prescreve a especificação das propriedades relativas a campos e comandos.

3.2.6 Descrição da Navegação

No nível lógico o MODEST prescreve que a descrição da navegação seja feita utilizando diagramas de classe com as transferências de controle modeladas por associações estereotipadas com estereótipo ≪*link*≫, de forma semelhante ao que é recomendado em algumas versões do processo PRAXIS (Filho, 2003). O MODEST prescreve o uso da mesma linguagem utilizada

para descrever as condições excepcionais presentes no sistema. Restrições nas associações são exibidas dentro de chaves próximas à associação.

A Figura 3.11 mostra as transferências de controle utilizando tal descrição. A transferência de controle entre a *UsersWindow* e a *SearchWindow* ocorre somente quando o botão *search* é clicado e o campo *login* está vazio.

O mapeamento do critério de navegação do modelo conceitual para o modelo lógico também é quase direto, conforme apresentado a seguir:

- Critério de navegação: dados um conjunto de teste T e um modelo M, T deve conter testes que exercitem todas as transferências de controles entre as classes de fronteiras modeladas em D.

3.2.7 Descrição do Funcionamento das Interfaces de Usuário

No nível lógico o MODEST prescreve o uso visão de estados para descrever o comportamento das classes de fronteira, utilizando para isso diagramas de estado. Cada classe de fronteira precisa ter pelo menos um diagrama de estado mostrando o funcionamento normal e suas exceções. As condições associadas às exceções devem ser modeladas como condições de guarda.

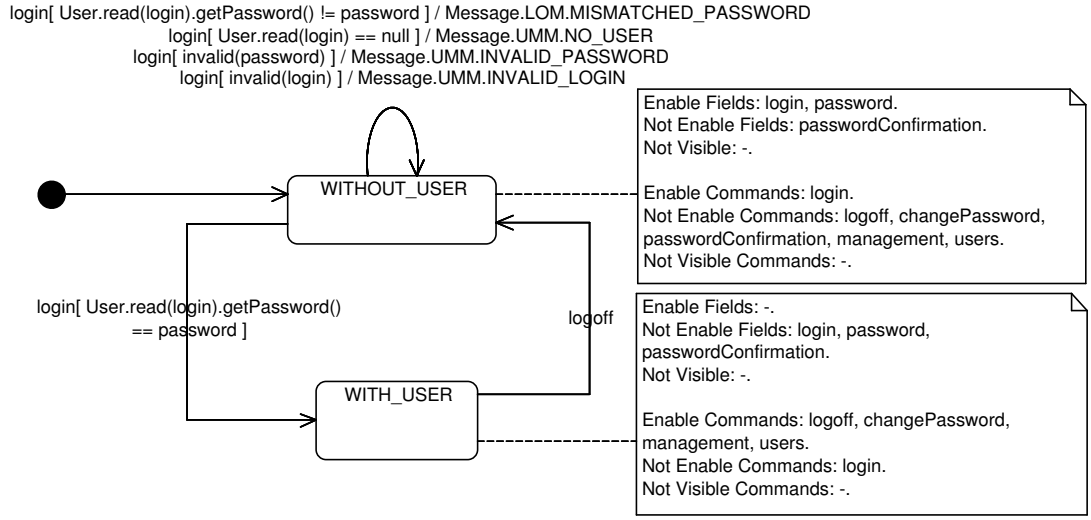

Figura 3.12: Trecho do diagrama de estados da *MainWindow*.

A Figura 3.12 mostra um exemplo dessas descrições para a *MainWindow*. O estado inicial da tela é WITHOUT_USER. Este estado descreve uma possível apresentação da tela, indicando os campos e comandos ativos, inativos e invisíveis. Cada uma das possíveis apresentações de uma tela precisa ser verificada, pois ela indica a possibilidade ou não de execução de certos comandos. Todas as exceções relacionadas ao estado WITHOUT_USER estão modeladas no diagrama de estado da Figura 3.12. Por exemplo, quando o usuário clica no botão *login* e a senha informada não confere com a armazenada, uma mensagem descrevendo isto

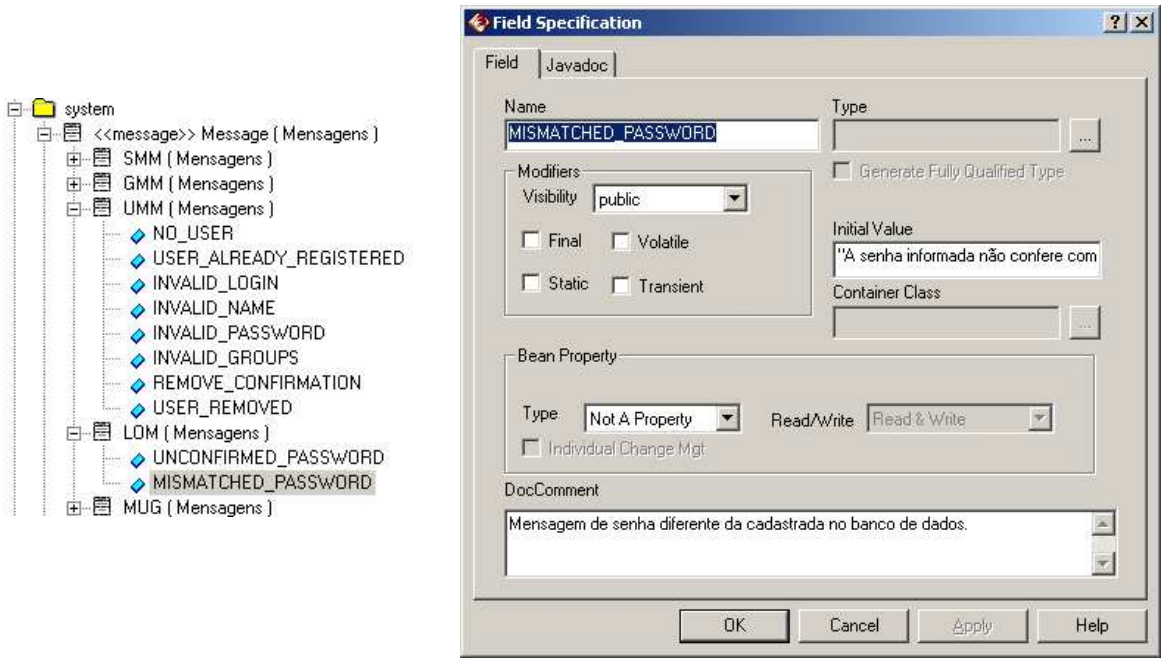

Figura 3.13: Classe do sistema contendo as mensagens exibidas aos usuários.

é mostrada (Message.LOM.MISMATCHED_PASSWORD). Essa sigla é uma outra convenção do MODEST. Isto é representado pela primeira transição do estado WITHOUT_USER na Figura 3.12. Todas as mensagens exibidas no sistema devem estar especificadas em uma classe de mensagens, nesse caso, existe uma classe chamada *Message* que contém tais mensagens. Elas estão agrupadas por caso de uso. A sigla "LOM"significa *Login Management*. A Figura 3.13 exibe a modelagem de tal classe.

No nível conceitual, o MODEST prescreve que todas as formas de apresentação das interfaces de usuário sejam utilizadas nos testes. No nível lógico, o MODEST prescreve a utilização do critério de teste denominado Cobertura de transição, definido por Offutt e Abdurazik (1999).

3.2.8 Planejamento dos Testes

No nível lógico, o MODEST prescreve que o planejamento dos testes inicie pela seleção de uma das colaborações que descrevem o sistema. Deve ser analisada a parte estática da colaboração, para serem inferidas as interfaces de usuário envolvidas. A parte dinâmica da colaboração deve ser utilizada para geração dos procedimentos de teste aplicáveis a essa tela. O MODEST prescreve então a análise do comportamento da classe de fronteira (Figura 3.12). Devem ser planejados testes que exercitem cada um dos estados existentes. Para isso, deve ser planejada a criação de testes que executem os comandos que causam a mudança do estado da tela. Após o planejamento de todos os testes envolvendo a classe de fronteira, seus diferentes estados, condições excepcionais e procedimentos de teste associados, o MODEST prescreve o planejamento de testes que permitam a navegação para todas as classes de fronteira alcançáveis

a partir da interface atual.

Algorítmo 4 Planejamento dos testes (nível lógico).

```
Procedimento PlanejadorDeTestes
  para cada caso de uso c do SUT faça
    para cada classe de fronteira t participante na colaboração de c faça
      marque os estados de t como não exercitados
      marque as transições nos diagramas de estados de t como não exercitados
      para cada estado inicial e da classe de fronteira t faça
        PlanejadorDeTesteDeGUI(e)
      fim para
      para cada navegação n possível a partir de t faça
        ct ← novo caso de teste para exercitar n
      fim para
    fim para
  fim para

Procedimento PlanejadorDeTesteDeGUI(estado et)
  se et já foi exercitado então
    abandone
  fim se
  para cada transição ta possível a partir de et faça
    c ← comando associado à transição ta
    pt ← procedimento de teste relacionado a c
    ct ← novo caso de teste para exercitar ta utilizando pt
    marque ta como exercitada
    para cada próximo estado pe alcançado a partir da transição ta faça
      se pe ≠ et então
        PlanejadorDeTesteDeGUI(pe)
      fim se
    fim para
  fim para
  marque o estado et como exercitado
```

O Algoritmo 4 apresenta, em uma forma simplificada, como o MODEST prescreve o planejamento dos casos de teste para um caso de uso. A idéia geral é sempre procurar por uma parte da especificação do software ainda não utilizada para criar um caso de teste. A principal diferença desse algoritmo com relação ao mesmo algoritmo no nível conceitual é o uso das construções da UML e dos estereótipos definidos pelo PU.

Para cada caso de uso, devem ser identificadas as classes de fronteiras associadas e analisados os diagramas de estados correspondentes. Os estados iniciais determinam quais comandos podem ser ativados. Baseado nessa informação é possível determinar os procedimentos de teste que devem ser utilizados. O MODEST prescreve então a criação de um caso de teste associado ao procedimento de teste que pode ser executado, associando ainda o teste a uma condição a ser exercitada. Na Seção 3.4.2.2 explicamos o algoritmo com um exemplo. Esses passos são repetidos até que todas as transições nos diagramas de estado e todos os procedimentos de teste tenham sido utilizados. Após isso, o MODEST prescreve o planejamento de testes que exercitem todas as possíveis navegações para outras classes de fronteiras associadas ao caso de uso utilizado para planejar testes, atendendo assim ao critério de Navegação.

3.2.9 Geração de Dados de Teste

No nível lógico o MODEST prescreve a geração das entradas para os casos de teste a partir da análise das condições a serem exercitadas, caso essas existam, junto com as propriedades dos campos envolvidos. Por exemplo, um dos casos de testes planejados pelo MODEST poderia estar relacionado ao procedimento de teste *Login*, com o objetivo de exercitar as transições e exceções contidas no diagrama de estado da *MainWindow*. A transição contendo a condição *User.read(login).getPassword() != password*, especificada utilizando nossa linguagem, seria interpretada da seguinte forma pelo gerador de dados de teste: 1 - é necessário gerar um login e uma senha válidos, 2 - deve existir um usuário no mecanismo de armazenamento com o login gerado como entrada para o caso de teste, 3 - a senha desse usuário deve ser diferente da senha gerada como entrada para o caso de teste.

Algorítmo 5 Geração de dados de teste (nível lógico).

```
Procedimento GeradorDeDadosDeTeste
  para cada caso de teste ct planejado faça
    se existe um condição c associada a ct então
      para cada entrada et relacionada à condição c faça
        gere dados para a entrada et baseada na interpretação da condição c
        se a condição c está relacionada a dados persistentes então
          gere dados persistentes a partir da condição c
        fim se
        se a condição c está relacionada a um mensagem ao usuário então
          gere um resultado esperado relacionado à mensagem
        fim se
      fim para
    fim se
    para cada entrada et não relacionada à condição c faça
      gere dados para a entrada et
      se condição c está relacionada a dados persistentes então
        gere dados persistentes a partir da condição c
      fim se
    fim para
    para cada entrada et gerada para o caso de teste ct faça
      para cada partição de equivalência p relacionada à entrada et faça
        outro ← cópia do caso de teste ct
        atualize a entrada et com um valor cobrindo a partição de equivalência p
      fim para
      para cada valor-limite v relacionada à entrada et faça
        outro ← cópia do caso de teste ct
        atualize a entrada et com um valor cobrindo a análise do valor limite v
      fim para
    fim para
    para cada dado persistente d gerado para o caso de teste ct faça
      outro ← cópia do caso de teste ct
      gere um novo dado persistente atendendo aos critérios definidos por (Andrews et al.,
      2003) e pelo MODEST.
    fim para
  fim para
  se se existe pós-condição associada aos comandos a serem executados então
    gere os resultados esperados baseados nas pós-condições existentes
  fim se
```

O Algoritmo 5 descreve a geração de dados de teste no nível lógico. A principal diferença desse algoritmo para sua versão no nível conceitual é o uso dos elementos de modelagem UML e estereótipos do PU, assim como a implementação atendendo aos critérios prescritos pelo método. A geração de valores válidos para os campos de uma classe de fronteira pode ser realizada por meio da análise das propriedades desses campos. Conforme visto anteriormente, esses campos devem indicar a sua origem e seu destino. O campo *login* da *MainWindow* do nosso exemplo está relacionado ao atributo *login* da entidade *User*. Esse campo é um campo chave para recuperação de usuários, tendo tamanho mínimo de dois caracteres e tamanho máximo oito caracteres, composto apenas por letras.

Dependendo da condição analisada, o MODEST prescreve a especificação dos dados de povoamento do mecanismo de armazenamento a serem utilizados no teste. No exemplo citado anteriormente, é exigida a existência de um usuário específico no mecanismo de armazenamento, como por exemplo a criação de um usuário com *login* "user"e *senha* "pass". Nesse caso, os dados gerados como entrada para o caso de teste poderiam ser, por exemplo, *login* "user"e *password* "other". Isso garantiria o exercício da condição (*"User.read(login).getPassword() != password"*), uma vez que os valores a serem utilizados na interface de usuário ("user", "other") não conferem com os valores armazenados no mecanismo de persistência ("user", "pass").

3.2.10 Execução dos Testes

No modelo lógico, o MODEST prescreve a utilização da informação da visão de gerenciamento de modelos da UML para a execução automática dos casos de teste. Utilizando os dados dessa visão é possível inferir o nome completo das classes envolvidas, incluindo o nome do pacote. Além disso, cada classe pode ser analisada para identificar todos seus atributos e operações.

O MODEST prescreve que o veredicto de um teste siga a especificação contida na *UML Testing Profile* (OMG, 2003b). Existem quatro valores válidos: *reprovado*, *inconcludente*, *aprovado* e *erro*. O veredicto *aprovado* indica que o caso de teste executou com sucesso e que o SST funcionou conforme o esperado. O veredicto *reprovado*, por outro lado, indica que o SST não funcionou conforme suas especificações. Um veredicto *inconcludente* significa que a execução do caso de teste não conseguiu determinar se o SST funcionou apropriadamente. O veredicto *erro* indica que o a ferramenta de teste falhou durante a execução do caso de teste. No caso do veredicto reprovado, o MODEST prescreve que a falha seja de alguma forma registrada para conhecimento dos responsáveis.

O Algoritmo 6 descreve a execução dos testes no MODEST. Conforme podemos observar, o algoritmo é muito próximo ao algoritmo do nível conceitual. A determinação de um veredicto para o teste segue à risca as prescrições do MODEST: as operações de persistência invocadas são analisadas e os campos da tela, juntamente com o mecanismo de persistência são verificados utilizando as regras de determinação do estado final do sistema após o teste.

Algorítmo 6 Execução dos testes (nível lógico).

```
Procedimento ExecutorDeTestes
  para cada caso de teste ct faça
    para cada entidade ent exigida para execução do caso de teste ct faça
      crie uma instância da classe de entidade ent
      para cada valor associado aos atributos de ent faça
        atribua o valor ao atributo
      fim para
      complete o restante dos atributos da entidade ent
      execute o método CREATE
    fim para
    crie uma instância t da classe de fronteira associada a ct
    coloque t no estado inicial especificado em ct
    para cada entrada especificada em ct faça
      atualize o campo de t com o valor especificado
    fim para
    acione o comando associado ao procedimento de teste pt de ct
    veredicto ← aprovado
    se estado resultante er não é o estado resultante de ct então
      veredicto ← reprovado
    fim se
    para cada resultado esperado re especificada em ct faça
      obtenha o valor v do campo
      se v ≠ re então
        veredicto ← reprovado
      fim se
    fim para
    para cada operação de persistência invocada em pt faça
      verifique o resultado das operação de persistência
      se existe algum resultado inconsistente então
        veredicto ← reprovado
      fim se
    fim para
    se houve erro no MODEST então
      veredicto ← erro
    senão
      se não foi possível determinar o veredicto então
        veredicto ← inconcludente
      fim se
    fim se
    grave o veredicto do teste
  fim para
```

3.3 O Modelo Físico do MODEST

Desenvolvemos um protótipo para funcionar como uma ferramenta de apoio ao MODEST. Conforme já mencionado, essa ferramenta é chamada de MODESTоо e seus componentes são apresentados na Figura 3.14. Nesta seção detalharemos cada um dos componentes da ferramenta.

A MODESTоо não possui uma especificação de requisitos formal. Isso não foi feito por que na época da sua construção não sabíamos exatamente o que fazer, em virtude de estarmos explorando uma área de pesquisa relativamente nova e sem uma referência bibliográfica apresentando as principais questões relacionadas. Resolvemos então utilizar a prototipação para gerar uma versão funcional da ferramenta que nos auxiliasse a entender melhor os problemas relacionados. Isso nos ajudou a levantar alguns requisitos apresentados na Seção 2.11, que se propõe a ser um catálogo de requisitos para métodos de TBM e que, indiretamente indica os requisitos associados à ferramentas de apoio a métodos de TBM.

A MODESTоо foi implementada em Java[1], utilizando um banco de dados gratuito denominado HSQLDB[2], sendo acessado através de uma camada de persistência implementada utilizando a tecnologia Hibernate[3]. Utilizamos também o Java CUP (Hudson, 2005) para geração de analisadores (parsers) para a interpretação das condições especificadas na linguagem selecionada para o MODEST.

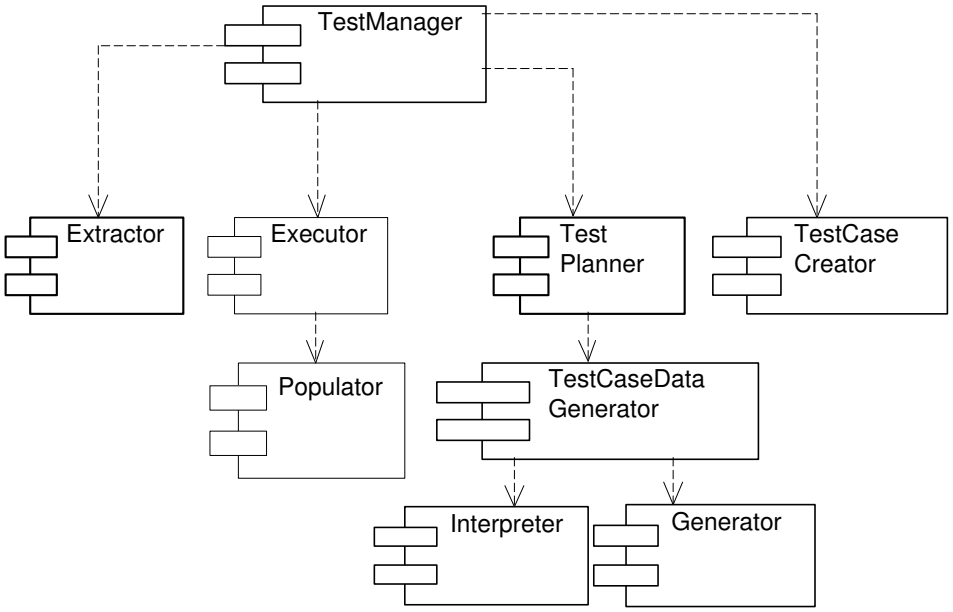

Figura 3.14: Uma ferramenta de apoio ao MODEST.

[1] http://java.sun.com/
[2] http://www.hsqldb.org/
[3] http://www.hibernate.org/

3.3.1 Extractor

O Extractor é o componente da MODESTоо responsável pela extração de dados do modelo descrevendo o SST para o formato utilizado pela MODESTоо. A MODESTоо foi originalmente planejada para ler uma especificação XMI (OMG, 1998) representando modelos UML descrevendo sistemas. O XMI é um formato para intercâmbio de metadados para ferramentas UML. Esse formato permite uma interoperabilidade entre ferramentas de modelagem UML, desde que elas sejam capazes de gerar um arquivo XMI representando o modelo. Ao tentarmos utilizar esse formato com o Rational Rose, notamos que nem todas as informações contidas nos modelos eram exportadas para arquivos XMI. Além disso a interoperabilidade entre ferramentas não funcionava, visto que o Rose utiliza uma versão do XMI diferente de algumas versões utilizadas em outras ferramentas. Isso nos fez abandonar esse formato.

Como essa opção não se mostrou satisfatória, utilizamos scripts Rose para gerar um arquivo texto com os dados do modelo, para então utilizarmos um analisador nesse arquivo e subseqüentemente povoar o mecanismo de persistência da MODESTоо com os dados extraídos. Por causa disso, a ferramenta atualmente funciona exclusivamente com modelos UML criados no Rational Rose.

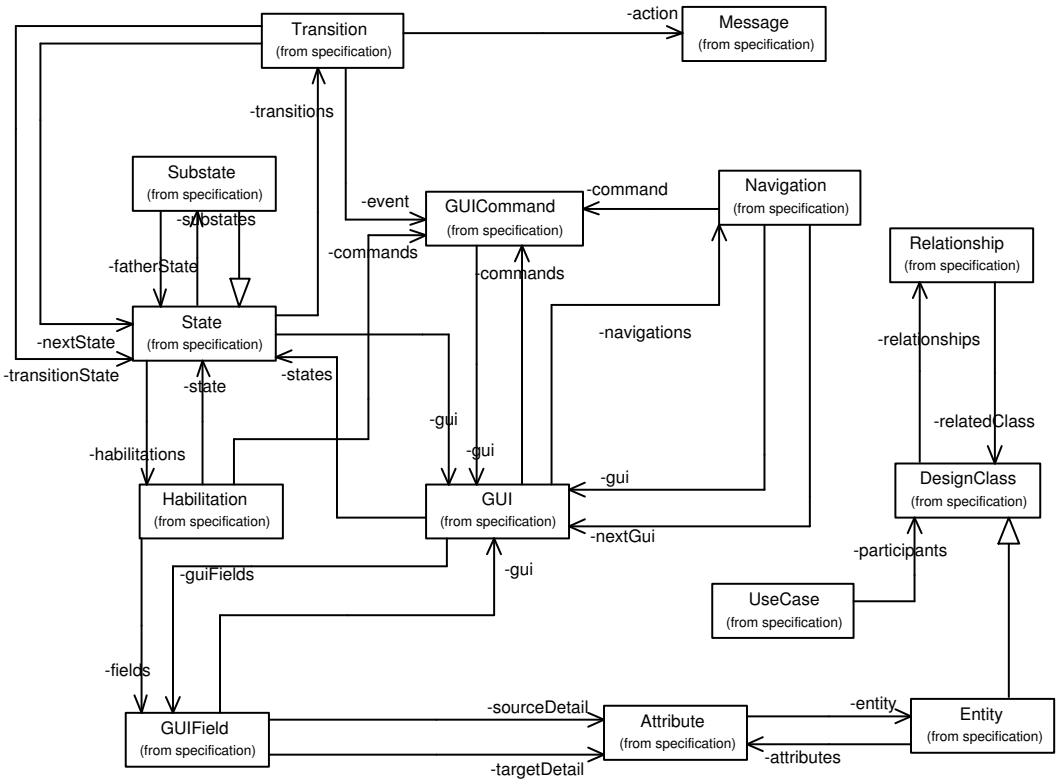

Figura 3.15: Pacote com a especificação do SST mantida pela MODESTоо.

O *Extractor* é composto basicamente pelo analisador acima mencionado. Ele é o com-

ponente da MODESTtoo para a extração de dados de modelos e povoamento do mecanismo de armazenamento da MODESTtoo com os dados referentes ao comportamento do SST. A Figura 3.15 exibe o diagrama de classes do pacote contendo a especificação do SST, que é povoado pelo Extractor. Durante esse povoamento é realizado um teste de consistência nas especificações, além da geração dos procedimentos de teste, que são baseados na descrição dos roteiros. Essas verificações de consistências são simples e descritas a seguir:

- Descrição da persistência. A descrição das operações de persistência deve conter um diagrama para cada método CRUD;
- Descrição de atributos. Todas as propriedades obrigatórias dos atributos das classes devem estar completamente descritos;
- Descrição das classes de entidade. A camada de entidade deve ter diagramas de classe mostrando todas as classes de entidade e seus relacionamentos;
- Descrição dos casos de uso. Cada caso de uso deve ter uma colaboração associada, com uma parte estática mostrando as classes participantes, e uma parte dinâmica, exibida na forma de diagramas de interação, mostrando os roteiros dos casos de uso;
- Descrição dos roteiros. As classes participantes dos roteiros devem estar presentes no diagrama de classes participantes. A primeira operação dos roteiros deve ser um comando de uma interface de usuário e seus eventuais parâmetros devem ser campos também dessa interface de usuário. Todas as condições excepcionais do roteiro devem possuir a especificação de uma condição que a faz ser acionada, juntamente com uma mensagem associada à mesma;
- Descrição das classes de fronteira. Todos os elementos de modelagem representando os campos das classes de fronteira devem ter as propriedades descrevendo sua fonte e seu destino preenchidas e consistentes com os atributos das classes de entidade eventualmente associados;
- Descrição de navegação. Deve haver um diagrama mostrando as transferências de controle entre as classes de fronteira, indicando para cada possível transferência o comando ativador da navegação;
- Descrição da habilitação. Cada estado em uma classe de fronteira deve especificar a habilitação dos campos e comandos;
- Descrição das restrições. As restrições utilizadas no diagrama de estados devem também estar modeladas nos roteiros dos casos de uso. Assim, se há a especificação da exibição de uma mensagem no diagrama de estados, deve haver o mesmo no roteiro. Além disso, nas especificações das restrições no diagrama de estados devem existir apenas campos da classe de fronteira associada ou expressões envolvendo classes de entidade;

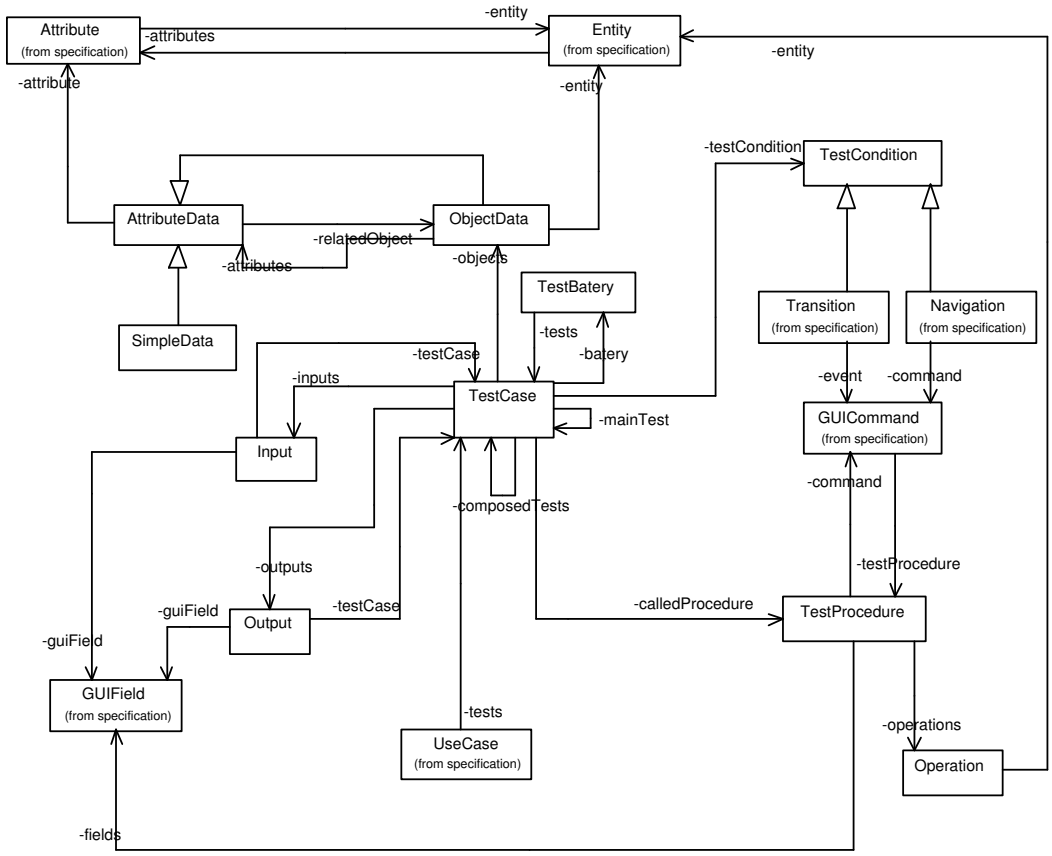

Figura 3.16: O pacote de testes da MODESToo.

- Descrição das mensagens. As mensagens utilizadas nos diagramas de estado e nos roteiros devem ser especificadas em uma classe de sistema estereotipada com estereótipo «message».

As consistências descritas acima são realizadas durante a extração de dados (*parsing*) do arquivo texto contendo as informações do modelo. Essas verificações auxiliam a criação de um modelo consistente com as exigências da MODESToo.

3.3.2 Test Planner

O *Test Planner* é o componente responsável pelo planejamento dos casos de testes para o SST. Cada caso de teste deve conter o procedimento de teste a ser utilizado, os estados alcançados durante a execução do procedimento de teste, as condições a serem exercitadas e as condições a serem avaliadas após a execução do teste.

O Algoritmo 4, do nível lógico, resume o comportamento do Test Planner. Como exibido no algoritmo, ele planeja a criação de um caso de teste a partir do caso de uso, utilizando as interfaces de usuário relacionadas.

```
NTEntityExpression ::=
  NTBegin NTEntity POINT NTRetrieve OB NTField CB NTEqComp NTNull NTEnd
  {:
    Generator.generateNullEntity(test, field);
  :}
  |
  NTBegin NTEntity POINT NTRetrieve OB NTField CB POINT NTGet OB CB NTComp NTValue NTEnd
  {:
    Generator.generateTest(test, ent, fields, attr, operation, isField, value);
  :}
  |
  NTBegin NTInvalid OB NTField CB NTEnd
  {:
    Generator.generateInvalid(test, field);
  :}
  |
  NTBegin NTField NTComp NTValue NTEnd
  {:
    Generator.generateTestForFieldExpression(test, fields, operation, isField, value);
  :}
;
```

Tabela 3.1: O analisador criado utilizando Java Cup para o MODEST.

Após a execução do Test Planner, todos os testes estarão criados e serão representados conforme apresentado na Figura 3.16. Nela podemos notar que cada caso de teste está relacionado a uma condição de teste e possui entradas e saídas, que são associados a elementos das interfaces de usuários. Além disso, podemos ver que um caso de teste possui dados persistentes exigidos para sua execução, representado na forma de um objeto de dados no diagrama.

3.3.3 Test Case Data Generator

O *Test Case Data Generator* é responsável pela geração dos dados de entrada. Ele também é responsável pela geração dos dados persistentes exigidos para a execução de um teste. Esse componente deve ser capaz de interpretar a linguagem utilizada para especificar as restrições no modelo; por isso o atual componente é baseado na linguagem utilizada neste trabalho e brevemente descrita na Subseção 3.2.4.

A Tabela 3.1 apresenta parte do código Java Cup (Hudson, 2005) representando a linguagem utilizada na especificação de restrições. Esse analisador é utilizado para separar os elementos da restrição, permitindo a interpretação automática e subseqüente geração de dados para o teste. Após a análise de uma restrição, um método específico para a geração dos dados de teste é invocado. Algumas vezes isso pode significar a criação de vários testes e não de apenas um, uma vez que os critérios utilizados podem exigir mais de um teste para cobrir uma determinada restrição. Isso é representado pela existência do conceito de testes compostos no diagrama de classes de teste da MODESToo (Figura 3.16). Um caso de teste pode ter vários testes compostos. Nesse caso a execução do caso de teste só termina quando a execução de todos seus testes compostos também terminar. Isso é utilizado, por exemplo, quando temos que exercitar todas as partições de equivalência de uma determinada condição.

A gramática atualmente utilizada para análise das restrições contidas nos modelos foi projetada para cobrir um conjunto limitado de casos. Conforme veremos no Capítulo 5 um dos trabalhos futuros é estender a MODESToo permitindo sua avaliação em outros contextos. Essa extensão implica na extensão da gramática para análise de restrições.

3.3.4 Populator

O *Populator* é o componente responsável pelo povoamento do mecanismo de armazenamento, baseado nos requisitos do caso de teste. Isso pode ser uma tarefa complexa, uma vez que uma entidade pode ter muitas associações e muitos atributos obrigatórios não especificados pelo *Test Case Data Generator*. O *Test Case Data Generator* especifica apenas a criação dos dados essenciais para o teste, ou seja, os dados que estão diretamente envolvidos com as restrições sendo verificadas. Por exemplo, se o *Test Case Data Generator* especifica a criação de um usuário com senha "aaaaaa", o *Populator* deve gerar o restante dos atributos para essa classe, como o *login* (Figura 3.5), o nome e o conjunto de grupos ao qual ele pertence, uma vez que esses atributos são obrigatórios para criação de um usuário.

O Algoritmo 7 resume o funcionamento do Populator. Ele é dividido em quatro componentes. O PovoadorDeTeste é responsável por invocar a geração de todos os objetos exigidos para a execução do caso de teste. Alguns testes podem exigir a criação de muitos dados, como por exemplo, testes utilizando a cardinalidade máxima de uma entidade. O procedimento GeradorDeDados é responsável por criar as instâncias dos objetos, armazenando valores para todos os atributos obrigatórios. O procedimento GerarValor é responsável por gerar um valor para um atributo simples, por exemplo, atributos associados a tipos primitivos.

Conforme já mencionado, durante a geração de dados para os casos de teste, o *Test Case Data Generator* gera apenas os dados essenciais ao teste. O método CompletarObjetos é responsável por gerar os dados restantes.

Algorítmo 7 O algoritmo do Populator.

```
Procedimento PovoadorDeTeste(caso de teste ct)
  para cada objeto obj exigido para execução de ct faça
    GeradorDeDados(obj)
  fim para

Procedimento GeradorDeDados(obj)
  instancia ← CriarEntidade(obj);
  para cada atributo attr de obj faça
    se attr é um objeto então
      objInterno ← GerarDados(attr);
    senão
      objInterno ← GerarValor(attr);
    fim se
    ArmazenarAtributo(objInter, instancia);
  fim para
  CompletarObjeto(obj)

Procedimento GerarValor(attr)
  result ← null;
  se o tipo de attr é lógico então
    retorne novo valor logico;
  senão
    se o tipo de attr é data então
      retorne uma novo data;
    senão
      ....
      se tipo de attr é texto então
        retorne um texto;
      fim se
    fim se
  fim se

Procedimento CompletarObjeto(obj)
  para cada atributo attr de obj faça
    se attr é obrigatório então
      se não existe dado já armazenado no atributo então
        se attr é um atributo simples então
          valor ← GerarValorValidoParaAtributo(attr);
        senão
          valor ← CompletarObjeto(attr);
        fim se
        armazene o valor ao atributo de obj
      fim se
    fim se
  fim para
```

3.3.5 Executor

O *Executor* é o componente responsável pela carga do sistema, entrada de dados e análise dos resultados. Ele é responsável por iniciar uma transação e criar os dados persistentes exigidos

para a execução do testes, antes da sua execução efetiva. Após a execução do teste a transação é cancelada e todas as alterações feitas são descartadas.

Para tornar possível a execução automática, o MODEST prescreve ainda que algumas convenções na nomeação dos elementos visíveis aos usuários finais do sistema. Essas convenções visam facilitar a identificação dos campos e comandos. A imposição é simples: o nome das variáveis representando os elementos modelados na interface de usuário devem seguir um padrão de nomeação da forma "estilo + nome do elemento de modelagem". Assim, o campo login, da Tela de login, que tem estilo "TextField", deve ter nome textFieldLogin na implementação da interface de usuário.

Esse componente é dependente da tecnologia alvo: se o SST é uma aplicação Web, ele requer um executor baseado em Web. Todas as condições relacionadas ao caso de teste devem ser avaliadas por esse componente, visto que ele é responsável pela determinação do veredicto do testes.

O Algoritmo 8 exibe os passos exigidos para a execução de um teste. O método ObterTela retorna uma interface de usuário no estado apropriado para a execução do teste. O método EntrarDados atribui os dados de entrada nos respectivos campos da interface de usuário. O método ExecutarComando executa um comando em uma interface de usuário. Esses passos correspondem à execução do teste. Em seguida são realizadas algumas verificações para saber se os resultados obtidos confere com o esperado. Ao final da execução do teste, as alterações realizadas no mecanismo de armazenamento são desfeitas, para que dados utilizados nos testes anteriores não interfiram nas próximas execuções.

Algorítmo 8 O executor de testes da MODESTtoo.

```
Procedimento ExecutarTestes(Caso de Teste ct)
  IniciarTransacao();
  GerarDeObjetos(ct);
  Procedimento de Teste pt ← ct.obterProcedimento();
  GUI tela ← ObterTela(ct);
  EntrarDados(ct, tela);
  Comando c ← pt.obterComando();
  ExecutarComando(c, tela);
  MostrarTela(tela);
  VerificarPosCondicoes(ct, tela);
  VerificarEstadoAlcancado(ct, tela);
  VerificarOperacoesPersistentes(ct, tela);
  CancelarTransacao();
  se ct é um caso de teste composto então
    para cada caso de teste composto ctc faça
      ExecutarTestes(ctc);
    fim para
  fim se
```

3.3.6 Test Case Creator

O *Test Case Creator* é uma funcionalidade da MODESTtoo que permite a geração de programas de teste em diversas tecnologias. Atualmente ele é capaz de gerar os testes na forma de programas Java com JUnit [4] e na forma de programas aceitos para execução pela ferramenta de mutação MuJava (Ma et al., 2005). Essa facilidade permite a execução dos testes sem

[4] http://www.junit.org/

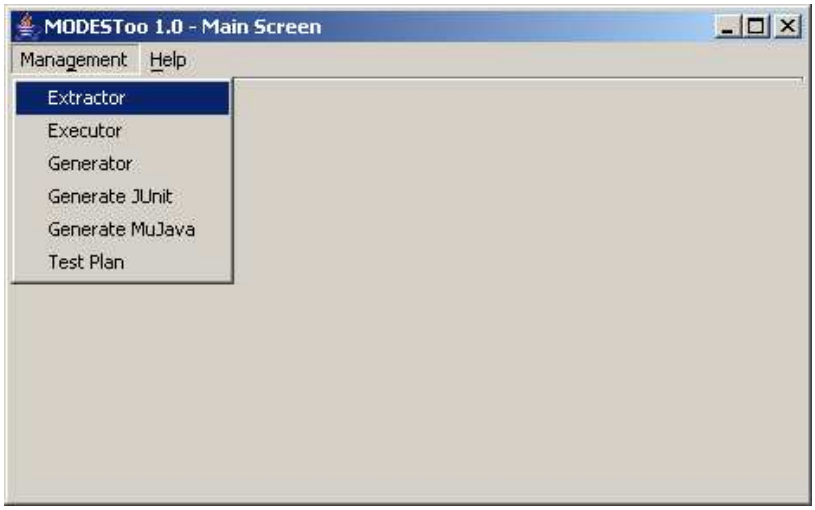

Figura 3.17: Visão da tela principal da MODESToo.

a execução do MODESToo, além de permitir a avaliação da qualidade dos testes gerados, através da integração com uma ferramenta de mutação.

3.3.7 Test Manager

O *Test Manager* é uma interface gráfica para administração da *MODESToo*. A Figura 3.17 apresenta a tela principal da ferramenta, que é o próprio Test Manager. Esse componente é utilizado para invocar os demais componentes da ferramenta, incluindo o gerador parcial do plano de teste. Esse componente deveria oferecer suporte para a criação de testes com apoio da especificação, visualização da especificação do sistema e agrupamento e agendamento dos testes, mas tais funcionalidades ainda não foram implementadas.

3.4 Personalizando o PRAXIS para o MODEST

Nesta seção descrevemos a personalização do PRAXIS para funcionamento com o MODEST. Conforme discutido neste capítulo, o MODEST é dividido em três níveis. No nível lógico é necessário mapear as prescrições do nível conceitual em termos do processo e da linguagem de modelagem utilizada. Nós definimos o nível lógico do MODEST utilizando a UML e o PU. Como existem diversos processos concretos, baseados no PU, que é um processo abstrato, resolvemos apresentar como os conceitos do MODEST poderiam ser mapeados para um processo baseado no PU.

Assim, nesta seção apresentamos a personalização do PRAXIS para o MODEST, detalhando os passos adicionais exigidos nas atividades do PRAXIS. Focalizamos em dois dos fluxos técnicos: Desenho (Figura 2.1) e Teste (Figura 2.2), uma vez que apenas esses fluxos são diretamente impactados com o uso do MODEST. Utilizamos como exemplo o caso de uso de Login e o caso de uso Gestão de Usuários do Merci. A realização desses casos de uso possui três interfaces de usuário: *MainWindow*, utilizada para a autenticação de usuários,

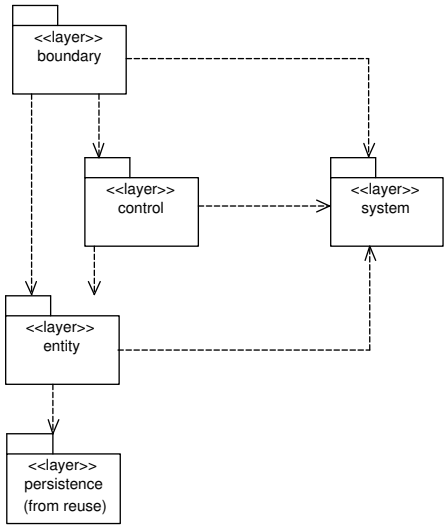

Figura 3.18: As camadas do Merci.

UsersWindow, utilizada para gerenciar usuários (*create*, *read*, *update*, e *delete*), e *SearchWindow*, utilizada para pesquisar um elemento em uma coleção.

3.4.1 Mudanças no Fluxo de Desenho

3.4.1.1 Desenho Arquitetônico

O Desenho Arquitetônico trata dos aspectos estratégicos do desenho. Ele define a divisão do produto em subsistemas, escolhendo as tecnologias mais apropriadas. As decisões sobre tecnologia devem viabilizar o atendimento aos requisitos não-funcionais e a execução do projeto dentro do prazo e custos estimados.

Jacobson et al. (1999) discute como determinar os elementos de modelo significantes arquiteturalmente em um sistema, permitindo assim definir a arquitetura do software. Em UML, subsistemas e outros componentes são representados por pacotes lógicos de desenho. São comuns arquiteturas descritas na forma de camadas representadas por pacotes. A Figura 3.18 mostra uma organização arquitetural recomendada pelo PRAXIS. Esta recomendação é compatível com as prescrições do MODEST, não havendo necessidade de atividades adicionais. Conforme discutido anteriormente, o MODEST é adequado para sistemas que possuam uma arquitetura composta por uma camada de apresentação, uma ou mais camadas contendo as regras de negócio e um mecanismo de armazenamento abstraído por uma camada de persistência (Figura 3.2).

Embora no Fluxo de Implementação exista uma atividade responsável pelo detalhamento do desenho, o Desenho Arquitetônico deve descrever uma configuração executável do sistema no seu ambiente real de execução. Isso pode ser feito por meio de um diagrama de desdobramento mostrando o modelo dinâmico do sistema, tal como o diagrama de desdobramento do Merci, exibido na Figura 3.19. Nesse caso, os clientes Merci enviam requisições para o servidor

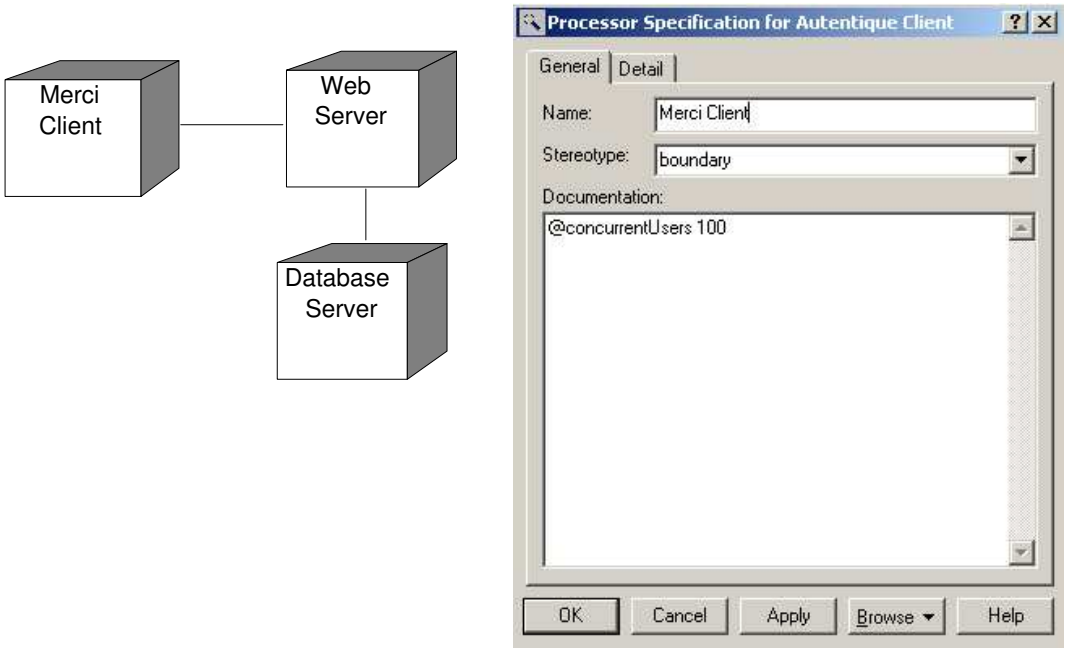

Figura 3.19: Diagrama de desdobramento com dados prescritos pelo MODEST.

Web que é responsável por toda a comunicação com o SGDB. Nesse caso, o uso do MODEST implica na especificação adicional das informações sobre quantidade de usuários concorrentes. Essa especificação é mostrada na Figura 3.19. O nodo representando a camada de apresentação deve ser estereotipado com estereótipo ≪*boundary*≫ e na documentação desse nodo deve existir a especificação da propriedade "@concurrentUsers".

3.4.1.2 Desenho das Interfaces

O Desenho das Interfaces trata do desenho real das interfaces de usuário, utilizando o ambiente escolhido para implementação. É comum produzir parte do código fonte das interfaces de usuários nesta atividade, uma vez que é mais fácil desenhar os elementos gráficos utilizando uma ferramenta de desenvolvimento integrada, para depois obter seu modelo via engenharia reversa.

Os atributos, operações e relacionamentos com outras interfaces de usuário são detalhados nesta atividade do PRAXIS. Os campos das interfaces de usuário devem detalhar algumas propriedades adicionais prescritas pelo MODEST. Essas propriedades se referem ao destino e à origem do campo e seu estilo na linguagem de programação utilizada. Na Figura 3.20 apresentamos essas propriedades para o campo *login* da *MainWindow*. Esse campo está relacionado ao atributo *login* da entidade *User*, detalhado na Figura 3.21 e discutida posteriormente. O estilo desse campo na linguagem alvo será *TextField*. É necessário enfatizar que a especificação de atributos e operações das classes de fronteira é uma tarefa prescrita pelo PRAXIS. O MODEST apenas requer alguns dados adicionais nessa descrição, exibidos na Figura 3.20.

Além da informação sobre campos e comandos, os desenvolvedores devem detalhar todos os estados e possíveis transições de uma interface de usuário. Um estado modela uma apre-

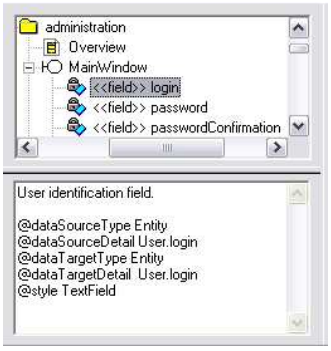

Figura 3.20: Propriedades relacionadas ao atributo *login* da *MainWindow*.

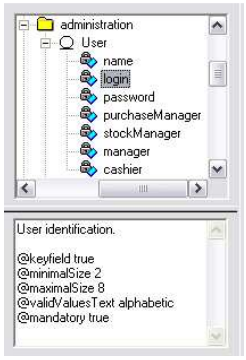

Figura 3.21: Propriedades relacionadas ao atributo *login* da entidade *User*.

sentação de uma interface de usuário, detalhando um conjunto de habilitações para campos e comandos. Esses dados já são prescritos pelo PRAXIS. O MODEST prescreve algumas convenções para a criação do diagrama de estados. Cada classe de fronteira deve ter um diagrama que mostre o funcionamento normal e os casos excepcionais, modelados utilizando restrições nas condições de guarda. Essa é a mudança mais significativa na adaptação, pois exige que cada transição possua uma condição formal que especifique as restrições para que a transição seja tomada.

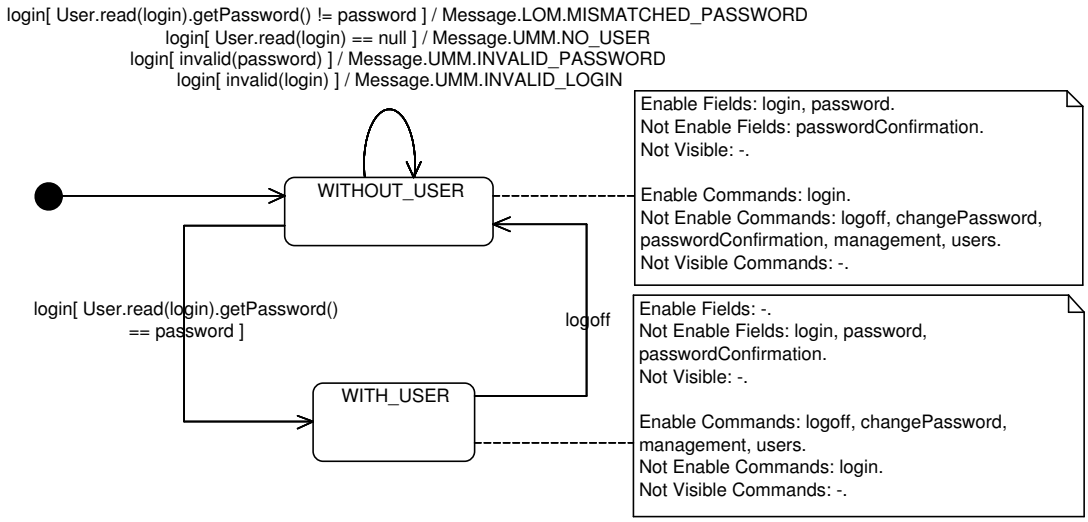

Figura 3.22: Trecho do diagrama de estados da *MainWindow*.

A Figura 3.22 mostra um exemplo dessas descrições para a *MainWindow*. Essa tela é utilizada para autenticar um usuário. De acordo com a figura, o estado inicial da tela é WITHOUT_USER. Esse estado descreve uma apresentação da tela, indicando os campos e comandos ativos, inativos e invisíveis. O MODEST exige que a identificação do status dos campos e comandos seja registrado no próprio diagrama de estados. Ele também prescreve

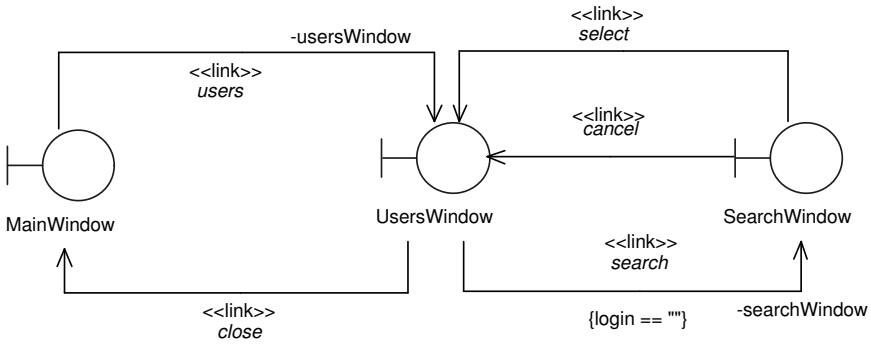

Figura 3.23: Exemplo de transferências de controle entre telas.

Precondições	1. A *MainWindow* está no estado WITHOUT_USER. 2. O UserActor digitou um texto no campo *login*. 3. O UserActor digitou um texto no campo *password*. 4. O UserActor clicou no botão *login*.
Passos	1. O**Merci** recupera o usuário com o *login* informado. 2. O **Merci** compara a *password* do usuário recuperado com a senha contida no campo *password*. 3. Se as senhas são iguais: 3.1 O **Merci** coloca a *MainWindow* no estado WITH_USER. 3.2 O **Merci** ativa os cardápios do sistema de acordo com as permissões do usuário.

Tabela 3.2: Um exemplo de fluxo de caso de uso.

que todas as transições devem especificar suas restrições por meio de uma linguagem formal. Na Figura 3.22 utilizamos uma linguagem simples baseada na camada de persistência utilizada para implementação do exemplo, criada com o intuito de simplificar essa tarefa.

Finalizando esta atividade, é necessário especificar o relacionamento entre as interfaces de usuários. Isso é uma tarefa usual no PRAXIS. O MODEST prescreve a criação de um diagrama detalhando as condições associadas às transferências de controle, caso existam, utilizando uma linguagem formal. A Figura 3.23 mostra uma parte das transferências de controle entre as interfaces de usuários do Merci. Nessa figura podemos notar que a navegação da *UsersWindow* para a *SearchWindow* é controlada pela condição *login == ""*.

3.4.1.3 Detalhamento dos Casos de Uso

O Detalhamento dos Casos de Uso resolve os detalhes relacionados ao desenho dos casos de uso, considerando os componentes reais das interfaces de usuários. Todos os fluxos alternativos devem ser detalhados, inclusive os que consistem de emissão de mensagens de exceção ou erro. Isso inclui o detalhamento do mecanismo de acesso, o fluxo principal, os subfluxos e fluxos alternativos. A Tabela 3.2 mostra um exemplo da descrição de fluxo no PRAXIS.

Durante esta atividade as exceções visíveis para os usuários do sistema devem ser descritas. Um tratador de exceções pode exigir apenas a exibição de uma mensagem ao usuário ou outros procedimento, como a execução de um fluxo alternativo. O PRAXIS recomenda a descrição de exceções e das ações e mensagens associadas. A Tabela 3.3 detalha algumas mensagens relacionadas ao caso de uso login. A única mudança relacionada ao uso do MODEST é a necessidade de especificação das mensagens exibidas aos usuários no modelo de desenho, em

Identificação	Categoria	Texto
MISMATCHED_PASSWORD	Informativa	A senha informada não corresponde com a senha armazenada.
UNCONFIRMED_PASSWORD	Informativa	A senha informada não confere com a senha contida no campo de confirmação de senha.

Tabela 3.3: Exemplo de descrição de mensagens relacionadas a um caso de uso do PRAXIS.

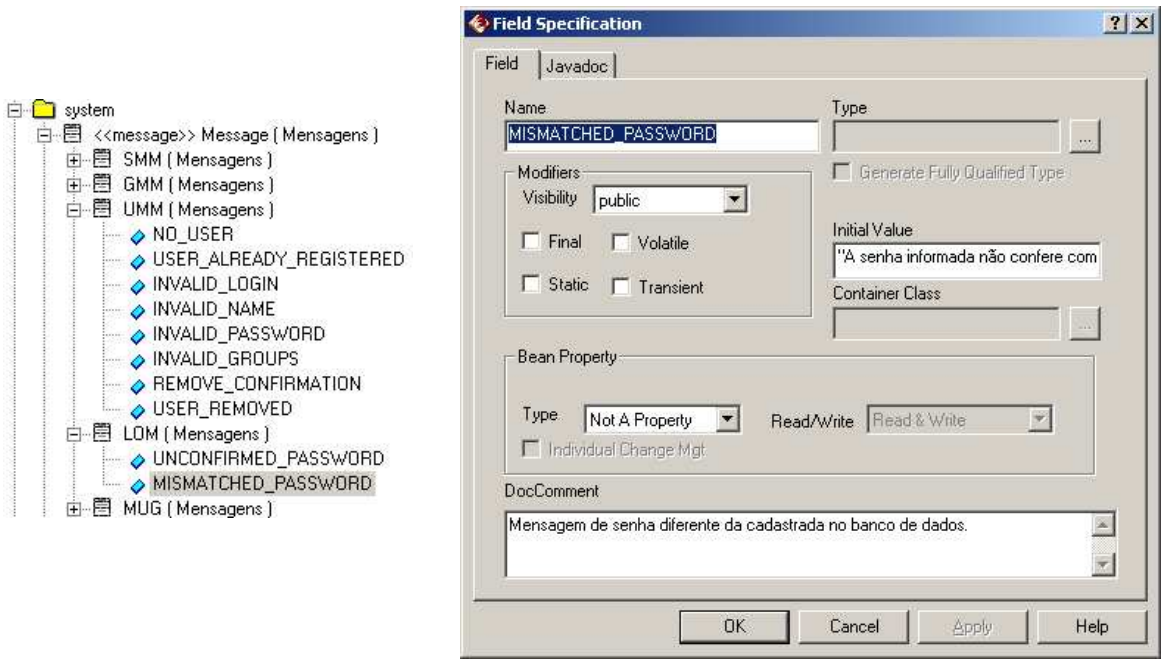

Figura 3.24: Classe do sistema contendo as mensagens exibidas aos usuários.

Propriedade	Descrição
keyField	Indica se o atributo pode ser utilizada para identificar elementos de forma única
minimalSize	Indica o tamanho mínimo dos dados suportado por esse atributo
maximalSize	Indica o tamanho máximo dos dados suportados por esse atributo
validValuesText	Indica o tipo de valores texto permitidos para um atributo texto (numérico, alfabético, ...)
mandatory	Indica se o atributo é obrigatório

Tabela 3.4: Propriedades especificadas pelo MODEST para atributos de entidades.

uma classe de sistema estereotipada com estereótipo $\ll boundary \gg$. Essa classe pode ser vista na Figura 3.24.

3.4.1.4 Desenho das Entidades

Esta atividade transforma as classes de entidade do modelo de análise nas classes correspondentes do modelo de desenho. Muitos detalhes devem ser determinados, como a visibilidade e navegabilidade dos atributos. Outra importante decisão é a definição da representação dos relacionamentos com cardinalidade maior que um.

O MODEST prescreve uma descrição mais específica dos atributos, incluindo as propriedades apresentadas na Tabela 3.4. Na atual implementação da ferramenta, essas propriedades

são especificadas utilizando a programação orientada por atributos (Wada et al., 2005), como mostrado nas propriedades referentes ao atributo *login* na Figura 3.21.

O MODEST também especifica algumas propriedades relacionadas às classes de entidade. Essas propriedades podem ser utilizadas para alguns testes não-funcionais. Detalhando a cardinalidade média de uma entidade e tempo médio para consultas, para cada uma das classes ou genericamente para todas, o MODEST pode povoar o mecanismo de armazenamento usando esses parâmetros, permitindo assim a realização de testes de desempenho e de estresse. Essas propriedades também são especificadas utilizando um formato semelhante à programação orientada a atributos. As propriedades "@averageCardinality"e "@queryProcessingTime"devem ser especificadas no campo para documentação da classe.

3.4.1.5 Desenho da Persistência

Esta atividade trata da definição das estruturas externas para armazenamento persistente, como arquivos e banco de dados. Dois problemas devem ser resolvidos: a definição física das estruturas persistentes externas, como esquemas de banco de dados, e a realização de uma ponte entre o modelo de desenho orientado a objetos e os paradigmas das estruturas de armazenamento, que geralmente são de natureza bastante diferente. Esta ponte é feita pela camada de persistência. As classes desta camada são desenhadas nesta atividade; entretanto, como uma camada de persistência bem desenhada é altamente reutilizável, muitas vezes a atividades se reduz a adaptações de componentes já existentes.

O PRAXIS recomenda a especificação do mecanismo de persistência utilizado. O MODEST prescreve algumas convenções durante essa definição: no Merci criamos para cada uma das operações CRUD um diagrama de interação que deixa claro como as operações de persistência podem ser invocadas pelos objetos persistentes. Além disso, tais diagramas foram criados dentro de uma pacote estereotipado com ≪*persistence*≫. A Figura 3.25 apresenta como exemplo o detalhamento da operação *Update*.

3.4.1.6 Realização dos Casos de Uso

Esta atividade determina como os objetos das classes de desenho colaborarão para realizar os casos de uso. As classes da camada de controle conterão o código que amarra essas colaborações, de maneira a obter-se o comportamento requerido pelos casos de uso de desenho. Diagramas de interação são usados para descrever as colaborações.

O PRAXIS prescreve a criação de diagramas de classes contendo as classes cujas instâncias participam nos diagramas de interação que ilustram o caso de uso. O MODEST prescreve o uso esse diagrama para identificar as interfaces de usuário relacionados ao caso de uso.

A Figura 3.26 mostra as classes que possuem instâncias participando da realização do caso de uso Login. O Merci divide seu funcionamento entre os elementos da arquitetura de cinco camadas. O funcionamento de *usuário* independente da aplicação é modelada na entidade *User*. A *MainWindow* é responsável pela entrada e saída de dados. A classe de controle *LoginHandler* implementa as operações de negócio relacionadas ao caso de uso. A partir

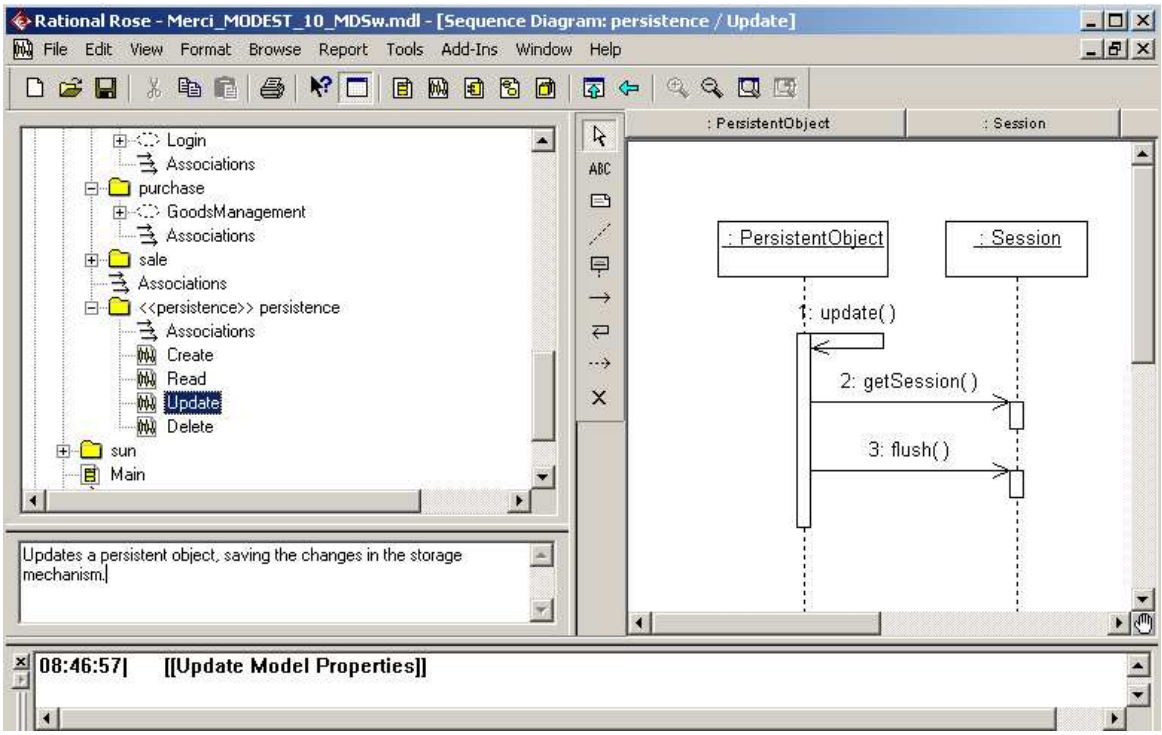

Figura 3.25: A operação *Update*.

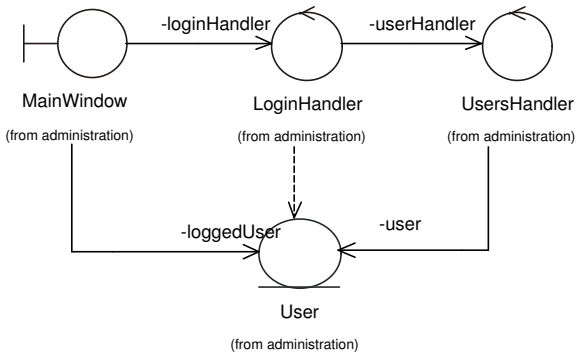

Figura 3.26: Diagrama mostrando os participantes do roteiro Login.

desse diagrama é possível identificar as interfaces de usuário relacionados ao caso de uso. Não existe nenhuma prescrição especial da MODEST na criação desse diagrama.

A Figura 3.27 mostra a divisão de comportamento entre as classes participantes do roteiro Login. A *MainWindow* apenas invoca as operações das classes de controle responsáveis pela execução de uma determinada função associada a um comando, mostrando em seguida os resultados e ativando os campos e comandos da tela de acordo com as permissões do usuário. O *LoginHandler* verifica os dados e implementa as regras do caso de uso Login. A entidade *User* modela os dados persistentes relacionados a um usuário.

O MODEST prescreve algumas convenções na descrição dos roteiros. Mensagens entre

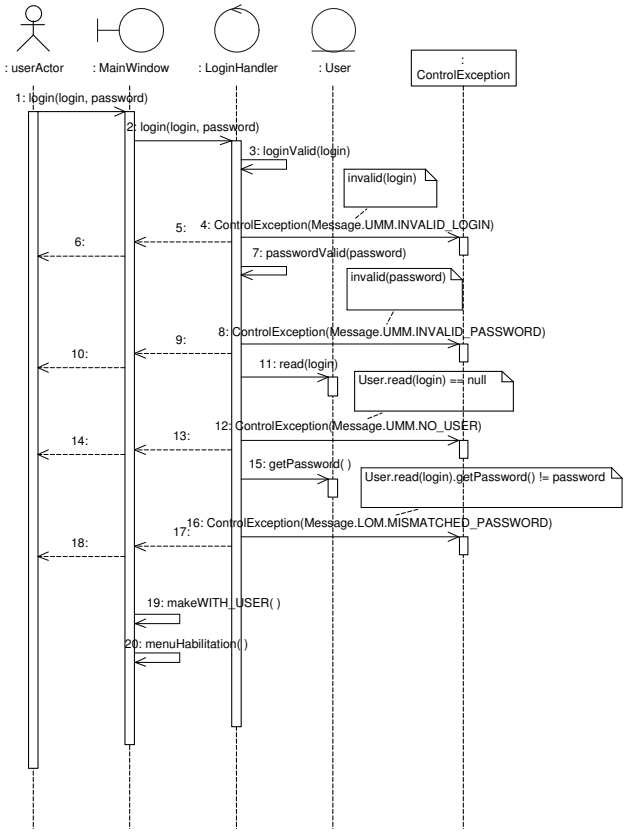

Figura 3.27: O roteiro Login.

atores e interfaces de usuários devem identificar o comando acionado e os dados necessários para a execução da funcionalidade selecionada. Isso é representado pela operação login(login, password), onde o primeiro *login* representa o comando acionado e o segundo *login* e *password* representam os dados associados à execução desse comando. Além disso, na descrição do diagrama devem ficar claras as operações de persistência invocadas, representada no diagrama pela operação *read(login)*, juntamente com todas as exceções associadas e suas condições de ativação, também apresentadas no diagrama.

3.4.1.7 Desenho das Liberações

Esta atividade determina como a construção do produto será dividida em liberações. Casos de uso e classes de desenho são repartidos entre as liberações, procurando-se mitigar primeiro os maiores riscos, obter realimentação crítica dos usuários a intervalos razoáveis, e possivelmente dividir as unidades de implementação de forma conveniente, entre a força de trabalho. Não existe qualquer prescrição do MODEST relacionado a esta atividade.

3.4.1.8 Revisão de Desenho

Esta atividade valida o esforço de Desenho, confrontando-o com os resultados dos Requisitos e da Análise. Dependendo da iteração, o PRAXIS padrão prevê revisões técnicas da Descrição do Desenho, ou inspeções, que focalizam principalmente o Modelo de Desenho. Esta atividade é usualmente utilizada como atividade de garantia da qualidade. Embora não seja proposto explicitamente pelo MODEST, em atividades como esta seria interessante a realização de uma revisão completa das descrições adicionais exigidas pelo uso do método. Isso poderia evitar problemas durante a geração e execução dos testes.

De forma geral, para que os testes para um caso de uso possam ser gerados automaticamente pelo MODEST é necessário a modelagem das seguintes informações:

- criação do caso de uso e da sua colaboração;
- criação de um diagrama representando a parte estática da colaboração, exibindo as classes participantes na realização do caso de uso (entidade, controle e fronteira).
- criação de diagramas de interação exibindo os principais roteiros do caso de uso; devem ser modelados os dados utilizados nos roteiros, assim como as operações de persistência associadas e possíveis exceções a serem levantadas, uma vez que esses roteiros serão utilizados como base para criação dos procedimentos de teste relacionados ao caso de uso;
- modelagem das classes de entidade associadas ao caso de uso, detalhando cada um dos seus atributos;
- modelagem das classes de fronteira associadas ao caso de uso, detalhando cada um dos seus atributos (campos) e operações (comandos), além de modelar o relacionamento de cada um dos campos com os atributos das classes de entidade;
- modelagem dos estados que as classes de fronteira associadas ao caso de uso podem assumir, detalhando os campos e comandos ativos, inativos e invisíveis em cada estado; além disso é necessário modelar as transições entre estados e possíveis mensagens exibidas ao usuário;
- modelagem de todas as mensagens exibidas ao usuário;
- modelagem das operações de persistência CRUD, permitindo assim a sua identificação nos roteiros dos casos de uso.

3.4.2 O Fluxo de Teste

3.4.2.1 Planejamento

O objetivo desta atividade é planejar o esforço de teste em uma iteração, descrevendo a estratégia de teste, além de estimar e agendar o esforço a ser empregado nas atividades do fluxo.

Identificação	Merci-Login-Login
Estado Inicial	WITHOUT_USER
Campos	login, password
Comandos	login
Operações de Persistência	User.read

Tabela 3.5: Exemplo da especificação de um procedimento de teste gerado pela MODESToo.

Input	login="aaaa", password="aaaa"
Persistent Data	login="aaaa", password="bbbb"
Test Procedure	Merci-Login-Login
Condition	User.read(login).getPassword() != password
Next State	WITHOUT_USER
Output	Message.LOM.MISMATCHED_PASSWORD

Tabela 3.6: Exemplo da especificação de um caso de teste gerado pela MODESToo.

O MODEST define como gerar um plano de teste parcial baseado na informação obtida nas atividades de desenho. O MODEST preenche diversas seções do documento oficial de planejamento, inserindo todos os dados relativos aos testes gerados de forma automática. Testadores utilizam esta atividade para completar o plano de teste e para planejar testes após uma avaliação dos testes gerados automaticamente pelo método.

3.4.2.2 Desenho

Durante esta atividade, os casos de teste e procedimentos de teste são gerados.

O MODEST prescreve como usar os roteiros criados no fluxo de desenho para gerar os procedimentos de teste. Como exemplo, consideraremos o roteiro da Figura 3.27. De acordo com as convenções do MODEST, mensagens entre atores e interfaces de usuários devem indicar os dados requeridos no roteiro e os comandos executados. Isso indica que o *login* e a *senha* (*password*) são os dados necessários para esse roteiro, além do comando *login* ser o comando associado ao roteiro. O procedimento de teste relacionado ao roteiro requer o estado WITHOUT_USER para execução. A Tabela 3.5 mostra uma parte do procedimento de teste obtido a partir do roteiro login. Ele foi gerado automaticamente pelo MODEST, que possui um componente responsável pela geração parcial do plano de teste do sistema, contendo os procedimentos e casos de teste. Esse plano de teste é formado basicamente pela composição dos procedimentos e casos de testes, aliado à identificação dos itens a testar.

A Tabela 3.6 mostra em detalhes a especificação de um caso de teste gerado pelo MODEST, que contém, dentre outros dados, a condição utilizada como base para criação do caso de teste (Condition). Nessa especificação existe o detalhamento dos dados de entrada (Input), os dados persistentes exigidos para execução do caso de testes (Persistent Data), o procedimento de teste a ser utilizado para execução (Test Procedure), a condição sendo avaliada (Condition), o próximo estado alcançado (Next State) e a saída esperada para esse teste (Output). Casos de teste são gerados a partir de uma análise das condições definidas durante o desenho, como por exemplo as condições de guarda. Isso é feito pelo interpretador para a linguagem utilizada para a especificação de condições.

Trans	Roteiro	Testes Planejados	Casos de Teste	Dados Persistentes
1	Login	login inválido	1. login alfabético menor que o mínimo, 2. login alfabético maior que o máximo, 3. login numérico com tamanho mínimo, 4. login com qualquer caractere com tamanho máximo, 5. login alfabético com espaço de tamanho médio	sem dados
2	Login	senha inválida	1. senha numérica menor que o mínimo, 2. senha alfabética maior que o máximo	sem dados
3	Login	usuário não cadastrado	1. login e senha válidos com tamanho mínimo, 2. login and senha válidos com tamanho máximo	1. sem dados, 2. cardinalidade máxima para a entidade User
4	Login	Senha incorreta	1. login e senha válidos com tamanho mínimo, 2. login e senha válidos com tamanho máximo	1. usuário com um login e senha válidos mas com senha diferente da usada como entrada, 2. cardinalidade máxima da entidade User, contendo um usuário com o login igual ao especificado na entrada do caso de teste, porém com senha diferente.
5	Login	login ok	1. login e senha válidos com tamanho mínimo, 2. login e senha válidos com tamanho máximo,	1. usuário com os mesmos valores do caso de teste, 2. cardinalidade máxima da entidade User, contendo um usuário com os mesmos valores utilizados na entrada do caso de uso.
6	Logoff	logoff ok	1. logoff ok	-

Tabela 3.7: Testes gerados pela MODESToo.

A Tabela 3.7 apresenta a descrição de alguns dos casos de teste gerados a partir das especificações do caso de uso *Login* do Merci. A coluna **Trans** representa a transição do diagrama de estado utilizado para geração do teste. A coluna **Roteiro** indica o roteiro a ser utilizado como base para a construção do procedimento de teste do caso de teste. A coluna **Testes Planejados** indica o que deve ser testado. As colunas **Casos de Teste** e **Dados Persistentes** descrevem as entradas para o caso de teste e os dados persistentes utilizados na execução do caso de teste.

De forma similar à atividade de planejamento, os testadores que utilizam o PRAXIS adaptado para o MODEST utilizam esta atividade apenas para projetar casos de teste adicionais.

3.4.2.3 Implementação

Esta atividade é responsável pela automação dos procedimentos de teste por meio da criação dos componentes de teste.

O MODEST não requer a implementação dos testes como prescrito pelo PRAXIS, uma vez que ela prevê como os casos de teste devem ser gerados e executados. No nível físico, o MODEST possui um código genérico capaz de executar os casos de teste gerados de forma manual, utilizando as facilidades do *Test Manager*, ou os testes gerados automaticamente.

3.4.2.4 Execução

Nesta atividade os testes são executados, produzindo os relatórios relacionados à sua execução.

Testes podem ser executados utilizando o *Test Manager* da MODESToo. Testadores podem executar todos os testes ou podem selecionar apenas parte dos testes existentes, tais

como os testes associados um caso de uso especifico. O MODEST prescreve a criação de um relatório relacionado a execução dos testes para análise por parte da equipe de teste.

3.4.2.5 Verificação do Término

Esta atividade determina se as condições para completeza e sucesso dos testes foram alcançadas. Essa é uma atividade importante, uma vez que ela é responsável pela avaliação dos testes gerados pelo MODEST. Algumas métricas são preparadas para determinar o nível de qualidade do software e para determinar quantos testes adicionais precisam ser feitos.

Esta atividade é normalmente executada no PRAXIS a partir da análise dos documentos de teste, tais como o relatório de incidentes e a especificação de procedimentos e casos de teste. O MODEST pode automatizar a geração desses documentos, facilitando a decisão sobre a necessidade de testes adicionais.

3.4.2.6 Balanço Final

Esta atividade faz um balanço final da bateria de teste. As lições aprendidas são registradas e alguns documentos relacionados à bateria de teste são preenchidos. Não existe uma prescrição especial do MODEST relativo a esta atividade.

3.5 Análise do MODEST em Relação aos Requisitos para TBM

Nesta seção analisamos o MODEST utilizando os requisitos associados a métodos de TBM discutidos no capítulo anterior. Finalizando esta seção fazemos uma análise comparativa entre o TOTEM, MODEST e AGEDIS, visto que esses dois outros métodos também foram analisados sob essa mesma perspectiva.

3.5.1 Requisitos do Engenheiro de Teste

3.5.1.1 Verificação do relacionamento entre o software e o armazenamento++

O MODEST trata, de forma explícita, o relacionamento entre o Software e o Armazenamento, assumindo a existência de uma camada de persistência. A descrição dos dados persistentes é similar a descrição utilizada no TOTEM, contudo, o MODEST prescreve a especificação de informação adicional, e utiliza essa informação para verificar o relacionamento entre o Software e o Armazenamento, além de facilitar a geração do oráculo.

3.5.1.2 Geração de testes para requisitos funcionais+++

O MODEST prescreve o uso de diversos critérios para a geração de testes, implementando alguns critérios definidos por Andrews et al. (2003) e Offutt e Abdurazik (1999), além de definir outros critérios.

O MODEST possui três atividades relacionadas à modelagem de GUI. Nessas atividades o MODEST prescreve a descrição das GUI, incluindo os campos e comandos acessíveis aos usuários finais, a especificação das possíveis transferências de controle entre instâncias da GUI, e a descrição de suas diferentes formas de exibição. Essa informação é utilizada para geração de testes, simulando o uso do sistema por seus usuários finais.

3.5.1.3 Geração de testes para requisitos não-funcionais+++

O MODEST é restrito a uma arquitetura pré-definida, mas existe uma atividade prescrevendo a especificação de alguns parâmetros relacionados à arquitetura. Esses parâmetros são utilizados para a geração de testes de desempenho e estresse. No entanto, essa parte do MODEST não foi implementada.

3.5.1.4 Geração de oráculo+++

O MODEST prescreve os requisitos que permitem a geração automática do oráculo. A viabilidade do uso desses requisitos é demonstrada na MODESTtoo. A solução adotada pelo método foi restringir as arquiteturas suportadas e usar algumas convenções de modelagem do comportamento do SST, assim como o uso de uma linguagem formal para detalhar restrições nesse comportamento. Essa abordagem é adequada para SI, mas sua extensão pode não ser trivial para outros tipos de aplicações.

3.5.1.5 Documentação de teste++

O MODEST prescreve a geração de um plano de teste parcial, contendo a especificação dos casos de teste e dos procedimentos de teste. O resultado da execução dos testes gera um relatório semelhante ao Relatório de incidentes.

3.5.1.6 Avaliação do teste+++

O MODEST não prescreve mecanismos para a avaliação dos testes, embora ele prescreva o uso de critérios para a geração de teste, assegurando certos níveis de cobertura.

3.5.1.7 Planejamento de testes++

Assim como os outros métodos, o MODEST também não possui mecanismos para auxiliar o planejamento dos testes.

3.5.2 Requisitos do Engenheiro de Componentes

3.5.2.1 Suporte para manutenção e regressão+++

Assim como os outros métodos, o MODEST também não prescreve mecanismos para manutenção e mudanças no sistema.

3.5.2.2 Interoperabilidade entre métodos++

O MODEST utiliza parte da UML Testing Profile em sua definição, representando os testes e o SST em um formato independente.

3.5.2.3 Geração de teste manual++

O MODEST possui um componente específico, denominado Test Manager, para dar apoio à criação de testes adicionais. Essa parte do MODEST ainda não está completamente implementada.

3.5.3 Requisitos para o Testador do Sistema

3.5.3.1 Execução de teste automática+++

O MODEST prescreve mecanismos que permitem à MODESToo executar automaticamente os testes existentes.

3.5.3.2 Acompanhamento de falhas+

O MODEST prescreve uma integração com alguma ferramenta de acompanhamento de falhas, mas não possui isso desenvolvido na versão atual.

3.5.4 Requisitos do Engenheiro de Processo

3.5.4.1 Aumento da qualidade e produtividade+++

O MODEST descreve um estudo experimental que indica que o uso do método no desenvolvimento de SI pode reduzir o esforço gasta nas atividades de teste, ao mesmo tempo em que aumenta a qualidade dos testes gerados, quando comparados a teste manualmente criados.

3.5.5 Comparação do MODEST x TOTEM x AGEDIS

A Tabela 3.8 apresenta um breve resumo do atendimento aos requisitos identificados por parte parte dos métodos analisados. Utilizamos três símbolos para representar o atendimento completo ao requisitos (↑), atendimento parcial (→) e o não atendimento (↓).

O TOTEM atende a poucos requisitos. Isso ocorre em virtude de tal método não possuir uma ferramenta de apoio. Muitas questões associadas ao TBM somente são identificadas durante a implementação de tais ferramentas. A geração de oráculos, por exemplo, é mencionada no TOTEM, porém, isso é feito de forma muito teórica e de difícil implementação. Por esse motivo, utilizamos a simbologia que o método atende parcialmente esse requisito. Da mesma forma, a teste do relacionamento entre o sistema e o armazenamento, assim como a utilização de critérios são apenas mencionados no trabalho. Apenas o requisito relacionado à geração de testes funcionais é atendido de forma satisfatório pelo método. Os demais requisitos não são sequer comentados no método.

	Requisito / Método	TOTEM	MODEST	AGEDIS
1	Armazenamento	→	↑	↑
2	Requisitos Funcionais	↑	↑	↑
3	Critérios	→	↑	↑
4	Arquitetura	↓	→	→
5	Requisitos Não-Funcionais	↓	→	↑
6	Oráculo	→	↑	↑
7	Artefatos de Teste	↓	↑	↑
8	Avaliação	↓	→	→
9	Planejamento	↓	↓	↓
10	Manutenção e Mudanças	↓	↓	↓
11	Interoperabilidade	↓	→	→
12	Testes Manuais	↓	→	↑
13	Execução Automática	↓	↑	↑
14	Acompanhamento de Falhas	↓	→	↑
15	Aumento da Qualidade e Produtividade	↓	↑	→

Tabela 3.8: Resumo da análise do atendimento aos requisitos pelos métodos considerados.

O MODEST atende a vários dos requisitos identificados. No entanto, alguns são atendidos de forma parcial. O teste da arquitetura do sistema é limitado por meio da restrição nos sistemas suportados. O teste de requisitos não-funcionais é mencionado no método mas não é efetivamente implementado. O método possui um mecanismo para avaliação dos testes por meio da integração com uma ferramenta de análise de mutantes mas essa integração não se mostrou bem sucedido na prática, devido a algumas limitações da própria ferramenta de mutação. O MODEST prescreve e implementa um mecanismo para garantir a interoperabilidade por meio do uso de um modelo independente de tecnologia e contendo ainda geradores de teste para tecnologias específicas. Embora isso não garanta a interoperabilidade de forma genérica é um passo nessa direção. O MODEST prescreve ainda o auxílio para a geração de testes manuais, assim como o acompanhamento automático de falhas, mas não possui nada implementado nessa direção.

O AGEDIS, embora seja um projeto grande, envolvendo diversos parceiros, assim como o MODEST, não atende de forma completa todos os requisitos identificados. O teste da arquitetura está limitado a configuração de alguns parâmetros para a execução de testes envolvendo sistemas distribuídos. A avaliação dos testes é limitada à verificação da cobertura de instruções alcançada. Embora o AGEDIS tenha definido uma linguagem para especificação de modelos de teste e do próprio SST, essas definições ainda estão longe de garantir a interoperabilidade entre métodos e ferramentas. Uma grande lacuna deixada pelo AGEDIS é a avaliação efetiva do método em um experimento. A documentação do método descreve de forma muito superficial algumas aplicações do método, tornando impossível avaliar se o seu uso pode trazer ganhos efetivos no desenvolvimento de software.

Uma interessante constatação obtida a partir da elaboração do catálogo de requisitos é que nenhum dos métodos considerados focalizou nos aspectos gerenciais relacionados ao testes, aspectos esses diretamente relacionados ao planejamento dos testes, assim como nos aspectos relacionados à constante necessidade por alterações em sistemas em desenvolvimento.

Conforme representado na Tabela 3.8 nenhum dos métodos considerou esses requisitos.

3.6 Considerações finais

Neste capítulo apresentamos o método de TBM MODEST. Apresentamos o método utilizando três níveis: conceitual, com as prescrições do método em alto nível; lógico, com um mapeamento dessas prescrições no contexto do PU e da UML; e físico, exibindo as tecnologias utilizadas para criação de uma ferramenta de apoio ao método.

No capítulo exibimos como criar um modelo UML com todos os requisitos de testabilidade exigidos pelo MODEST. Utilizando um modelo com tais características, o MODEST planeja, cria e executa testes, podendo com isso reduzir o esforço necessário nessas atividades. Utilizando o MODEST foi possível comprovar que a reutilização de artefatos prescritos em um processo de software pode contribuir significativamente para a automação das atividades de teste, podendo ainda aumentar, não somente a qualidade do produto final, como também a própria qualidade da especificação do software.

Neste capítulo também apresentamos alguns critérios de testes baseados no uso de modelos, e em particular, no uso de modelos UML. Conforme descrito, o MODEST segue alguns critérios definidos em outros trabalhos, além de definir novos critérios, como o Critério de estresse, Critério de composição, Critério de desempenho e Critério de navegação.

Uma importante diferença do MODEST para alguns trabalhos discutidos no capítulo anterior é a sua especialização para uma arquitetura específica. A tentativa de ser muito genérico impõe altos custos no seu uso. A especialização do MODEST o torna mais barato, visto que a geração do oráculo, que geralmente é o aspecto que demanda mais custo e esforço nesse tipo de método, não requer um esforço tão grande quanto o exigido na maioria dos outros métodos e ferramentas aqui apresentados.

A personalização do MODEST para o PRAXIS não é complexa, visto que a extensão das atividades de desenho não gera um impacto significativo na construção de software. Além disso, a redução no esforço que pode ser obtida com o uso do método tende a ser significativa, uma vez que o MODEST pode automatizar a maioria das atividades de teste, que, de acordo com o que foi discutido na Introdução, geralmente é responsável pelo consumo de boa parte dos recursos de um projeto.

No próximo capítulo apresentamos detalhes sobre um estudo experimental realizado com o intuito de caracterizar o método. Conforme apresentado no estudo, o MODEST pode reduzir os custos relacionados ao desenvolvimento de software, ao mesmo tempo em que pode aumentar a qualidade dos testes gerados.

Capítulo 4

Um Estudo Experimental do MODEST

Nosso estudo da literatura mostrou que a maioria dos métodos e ferramentas para automação de testes apresenta avaliações mostrando o impacto das atividades propostas, em termos de qualidade, sem demonstrar o impacto dessas atividades, em termos de custos. Isso motivou nossa investigação sobre como avaliar nosso método levando em consideração não só os possíveis ganhos em qualidade, mas também os custos associados ao seu uso. Neste capítulo apresentamos um estudo experimental para caracterizar o uso do MODEST, seguindo com rigor as prescrições existentes na Engenharia de Software Experimental (Wohlin et al., 2000). O estudo mostra que o uso do método pode trazer tanto redução de custos, através da redução do esforço gasto nas atividades de testes, quanto aumento na capacidade de detecção de falhas.

Nas próximas seções detalhamos este estudo usando a seguinte organização: a Seção 4.1 apresenta o estudo experimental realizado; a Seção 4.1.11 apresenta algumas tentativas de estudos realizadas anteriormente; e a Seção 4.1.12 apresenta as considerações finais do capítulo.

4.1 Estudo experimental

Nesta seção descrevemos o estudo experimental realizado com o intuito de caracterizar o uso do MODEST no desenvolvimento de parte de um SI.

4.1.1 Definição dos objetivos

O propósito deste estudo experimental é caracterizar o uso do MODEST no desenvolvimento de SI, verificando se seu uso pode, por exemplo, aumentar a capacidade de detecção de falhas dos testes gerados, e ao mesmo tempo reduzir os custos do desenvolvimento, em função da diminuição do esforço necessário na execução das atividades de teste. O foco de qualidade deste estudo está descrito na Subseção 4.1.2, onde foram escolhidos o foco no esforço (tempo) e capacidade de detecção de falhas.

Conforme apresentado na Seção 2.10, os diversos trabalhos analisados apresentam métodos e ferramentas de automação de teste, contudo sem apresentar estudos experimentais que possam indicar ganhos relacionados ao seu uso, tanto em termos de qualidade dos testes quanto em redução de custos associados. A maioria desses métodos avalia apenas aspectos de qualidade, ignorando os aspectos relacionados aos custos.

Dessa forma, este trabalho descreve um estudo cujo objeto é o método para automação de testes MODEST, juntamente com sua ferramenta de suporte, chamada MODESTtoo. O estudo foi realizado sob a perspectiva dos pesquisadores, com o intuito de saber se o uso do método pode trazer ou não ganhos ao desenvolvimento, em termos de esforço e capacidade de detecção de falhas, conforme comentado anteriormente. O estudo experimental pode ser classificado com um estudo de caso, conduzido dentro de um ambiente controlado (in vitro), no contexto da disciplina de Engenharia de Software para alunos de graduação em Ciência da Computação do Departamento de Ciência da Computação (DCC) da Universidade Federal de Minas Gerais (UFMG).

Nessa disciplina foi apresentado o processo PRAXIS. No estudo desenvolvemos parte do Merci. Esse sistema é o sistema exemplo disponível no sítio WWW com o material de apoio ao processo PRAXIS. Selecionamos dois casos de uso para o estudo: Login, estimado em 20 pontos de função não ajustados e Gestão de Mercadorias, estimado em 29 pontos de função não ajustados. Comparamos o desenvolvimento desses dois casos de uso utilizando o processo PRAXIS instanciado com e sem o MODEST, sendo aqui referenciados, respectivamente, como MEP (MODEST Enhanced Process) e NEP (Non Enhanced Process). Os estudantes executaram o fluxo de desenho e teste prescrito pelo PRAXIS para cada um dos casos de uso.

O caso de uso Gestão de Mercadorias é praticamente 50% maior que o caso de uso Login, contudo, ele é muito mais complexo, uma vez que envolve associação entre elementos (Mercadorias e Fornecedores), navegação entre objetos, e diversas regras de validação, junto com todas as operações CRUD. Essa complexidade é evidenciada em termos do seu tamanho contado em linhas de código (LOC): o caso de uso login corresponde a 885 LOC, enquanto o caso de uso Gestão de Mercadorias corresponde a 1940 LOC. Esses dois casos de uso foram selecionados devido às suas diferenças, reduzindo algumas ameaças à validade, conforme discutido a seguir.

Nós selecionamos como participantes deste estudo os alunos que se inscreveram para a disciplina de Engenharia de Software, ocasionando a ausência de um importante ingrediente em um estudo totalmente controlado: falta de aleatoriedade. Este estudo é, portanto, classificado como um quasi-experimento.

4.1.1.1 Resumo da definição

Analisar *a utilização de um processo de software instanciado com e sem o MODEST*, com o propósito de *caracterizar*, com respeito ao *esforço e capacidade de detecção de falhas*, do ponto de vista do *pesquisador*, no contexto de *alunos de graduação em Ciência da Computação desenvolvendo sistemas de informação.*

4.1.2 Questões e métricas

O estudo experimental foi projetado com o objetivo de responder às seguintes questões:

- O esforço para a criação das especificações de desenho seguindo as prescrições do processo instanciado com o MODEST é maior que o esforço exigido para a criação das especificações de desenho seguindo as prescrições do processo sem o MODEST? Utilizamos a medida de tempo em horas como métrica para a criação das especificações de desenho.

- O esforço para criar um modelo de desenho com os requisitos de testabilidade do MODEST e subseqüente geração de testes é menor que o esforço para executar as atividades de teste sem o suporte do MODEST? Utilizamos a medida de tempo em horas como métrica para a realização das atividades de desenho e teste. Observe que no item anterior a principal questão é: o esforço de modelagem utilizando MODEST é maior que o esforço sem utilizá-lo? Neste item a questão é avaliar se o esforço para testes utilizando MODEST, que corresponde basicamente à modelagem do SST, é diferente do esforço para geração de testes sem o apoio do MODEST, que nesse caso corresponde à realização manual das atividades de teste.

- A capacidade de detecção de falhas dos testes gerados pelo uso do MODEST é maior que a capacidade de detecção de falhas dos testes gerados manualmente pelos participantes? Utilizamos a quantidade de falhas detectadas por bateria de testes como métrica para essa avaliação.

4.1.3 Hipóteses

- Hipótese nula, $H_{0\ desenho}$: Não existe diferença no esforço, medido em termos de tempo em horas, para criar as especificações de desenho com e sem o uso do MODEST. $H_{0\ desenho}$: TempoDesenho(MEP) = TempoDesenho(NEP). Hipótese alternativa, $H_{1\ desenho}$: TempoDesenho(MEP) $\neq$ TempoDesenho(NEP).

- Hipótese nula, $H_{0\ teste}$: O esforço empregado no desenvolvimento (desenho + testes) utilizando o MODEST, medido também em termos de tempo em horas, é igual ao esforço empregado na execução das atividades de teste sem utilizar o MODEST. $H_{0\ teste}$: EsforçoTeste(MEP) = EsforçoTeste(NEP). Hipótese alternativa, $H_{1\ teste}$: EsforçoTeste(MEP) $\neq$ EsforçoTeste(NEP).

- Hipótese nula, $H_{0\ qualidade}$: A capacidade de detecção de falhas dos testes gerados manualmente pelos participantes NEP, medida em termos do número de falhas encontradas com a semeadura de erros, é igual à capacidade de detecção de falhas dos testes gerados automaticamente pelos participantes MEP. $H_{0\ qualidade}$: Detecção(MEP) = Detecção(NEP). Hipótese alternativa, $H_{1\ qualidade}$: Detecção(MEP) $\neq$ Detecção(NEP).

Pergunta	Possíveis Respostas	Sua Resposta	Questões Adicionais	Detalhamento
Voce já trabalhou com o desenvolvimento de software?	0 - Nunca trabalhei ou fiz estágio nesta área, 1 - Trabalhei (ou fiz estágio) mas não me envolvi com desenvolvimento de software, 2 - Trabalhei (ou fiz estágio) e era codificador, mas não sabia se existia um processo de software, 3 - Já trabalhei com dese		Se 3, detalhe qual(is) o(s) processo(s) utilizado(s), por quanto tempo e qual sua atividade na empresa.	
Você já implementou testes de unidade para algum código que voce desenvolveu? (Pode ser incluído aqui trabalhos de disciplinas.)	0 - Nunca escrevi testes antes, apenas executava os programas criados com algumas entradas para verificar se estava funcionando, 1 - Já fiz testes mas não faço com constância, 2 - Já fiz bastante, continuo fazendo para quase tudo que desenvolvo.		Se 1 ou 2, o que foi usado e qual seu tempo de experiência no assunto.	
Voce já usou alguma ferramenta de modelagem, tais como Rose, Poseidon, Together, MagicDraw, ...	0 - Nunca usei, 1 - Já usei mas nem sabia direito porque estava utilizando, 2 - Já usei muito mas só quando sou obrigado, 3 - Já usei bastante e continuo usando na maioria dos projetos que participo.		Se 1, 2 ou 3, descreva qual(is) ferramenta(s) foram utilizadas, por quanto tempo e descreva o que foi modelado com tais ferramentas.	
Voce já usou alguma ferramenta para automação de testes?	0 - Nunca usei, 1 - Já usei mas nem sabia direito porque estava utilizando, 2 - Já usei muito mas só quando sou obrigado, 3 - Já usei bastante e continuo usando na maioria dos projetos que participo.		Se 1, 2 ou 3, descreva qual(is) ferramenta(s) foram utilizadas e por quanto tempo.	

Figura 4.1: Questionário de caracterização dos participantes do estudo.

4.1.4 Seleção de variáveis

As variáveis independentes são: o processo de software e o método de teste MODEST. As variáveis dependentes são: o tempo de criação das especificações de desenho, o tempo para a realização das atividades de desenho e testes, e a capacidade de detecção de falhas dos testes gerados.

4.1.5 Seleção dos participantes

Utilizamos como participantes do estudo os alunos da disciplina Engenharia de Software do DCC/UFMG, ofertada no segundo período de 2005. Essa disciplina é oferecida para alunos do último ano de graduação em Ciência da Computação, curso esse previsto para ser completado em quatro anos.

Os alunos dessa disciplina geralmente possuem bons conhecimentos em programação e pouca ou nenhuma experiência no uso de processos de software, ferramentas e métodos de automação de testes. Ainda assim, solicitamos aos participantes que respondessem um questionário para caracterizar sua formação e experiência em Engenharia de Software, notadamente no uso de processos de software, ferramentas e métodos de teste, no intuito de evitar algum viés que pudesse perturbar o estudo. Esse questionário é apresentado na Figura 4.1.

O estudo iniciou com 34 participantes, no entanto, conforme descrito nas próximas seções, nem todos concluíram todas as etapas. A maioria dos participantes (60%) tinha até dois anos de experiência profissional trabalhando com codificação. O restante (40%) não possuía experiência profissional. Dentre todos os participantes, apenas 8% já havia trabalhado com processos de software, porém, eles utilizaram processos ágeis, totalmente diferentes do processo utilizado no estudo. Esse é o mesmo percentual de participantes que afirmaram já ter utilizado uma ferramenta de modelagem, embora diferente da ferramenta utilizada no estudo. Com relação a testes, apenas 8% afirmaram já ter trabalhado com testes, mas isso se resumiu à realização de testes de unidade.

4.1.6 Treinamento

O questionário de caracterização dos participantes mostrou que os participantes não possuíam conhecimentos relacionados ao estudo. A maioria dos conhecimentos em Engenharia de Software foi adquirido durante a própria disciplina. Assim, os grupos tiveram duas horas de aula sobre geração de testes, abordando critérios de teste como o Particionamento em Classes de Equivalência e Análise de Valor-Limite, entre outros, para que seus testes pudessem atingir bons níveis de cobertura. Nessa aula também foi apresentado o gabarito a ser utilizado para a especificação de teste. Além disso, houve mais quatro horas de aula explicando como implementar testes utilizando a ferramenta Abbot[1], selecionada para ser a ferramenta para implementação de testes manuais utilizada no estudo. Essa ferramenta é utilizada para implementação dos testes, por meio da criação de um script que pode re-executada sempre que necessário. A ferramenta possui um mecanismo de captura de ações executadas em um sistema, permitindo sua re-execução de forma automatizada. Além disso, é possível programar certas ações utilizando a linguagem utilizada para representação dos scripts.

Com relação às atividades de desenho, houve quatro horas de aula explicando o padrão e as convenções utilizadas para o desenho dos casos de uso para cada um dos grupos (MEP e NEP). As quatro horas de treinamento foram apresentadas durante as duas etapas do estudo, conforme descrito a seguir e apresentado na Figura 4.2.

Após a conclusão de cada um dos treinamentos, os participantes tinham que realizar um exercício sobre os assuntos abordados no treinamento. Os participantes só podiam iniciar o estudo após a conclusão do exercício, antecipando assim a familiarização com os métodos e ferramentas utilizados no estudo.

4.1.7 Projeto do estudo

O estudo foi realizado em duas etapas, conforme apresentado na Figura 4.2. Na primeira etapa os alunos com número de ordem ímpar na caderneta da disciplina utilizaram o MEP e os alunos com número de ordem par utilizam o NEP para realizar o mesmo conjunto de tarefas: desenhar e testar o caso de uso Login. Na segunda etapa do estudo a atribuição de participantes aos tratamentos (*treatment*) foi invertida, no intuito de termos um maior controle sobre o estudo. Nessa etapa os participantes MEP e NEP (agora invertidos) tiveram que desenhar e testar o caso de uso Gestão de Mercadorias.

Decidimos não realizar o agrupamento (*blocking*) tendo em vista que a caracterização dos participantes nos mostrou que, embora alguns participantes trabalhassem como desenvolvedores na indústria, eles praticamente não possuíam conhecimentos dos objetos utilizados no estudo. Utilizamos um fator (*factor*) com dois tratamentos, assim, cada participante utilizou tanto o MEP quanto o NEP, porém em dois casos de uso com características bem diferentes. A atribuição desses elementos aos tratamentos foi feita utilizando o número de ordem da caderneta da disciplina. Na primeira etapa os alunos com número de ordem ímpar utilizaram o MEP e os pares o NEP. Na segunda etapa isso foi invertido.

[1] http://abbot.sourceforge.net

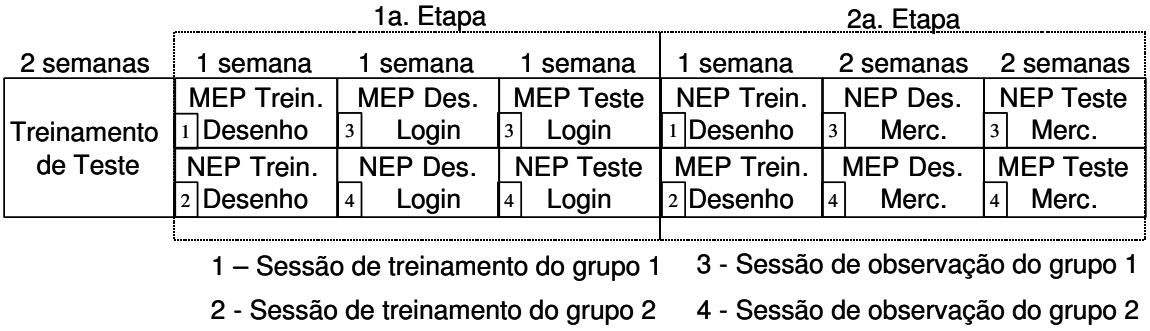

Figura 4.2: Agenda de atividades do estudo.

Data	Fluxo	Descrição	Duração	Inicio	Fim
13/09/2004	DS	Modelagem da TelaPrincipal do merci, com campos e comandos, diagrama de estados.	1:00	21:00	22:00
14/09/2004	DS	Finalização da modelagem da TelaPrincipal.	1:00	18:10	19:10
14/09/2004	DS	Descrição do CUD Login.	1:00	19:10	20:10
14/09/2004	TS	Criação do plano de testes para o CUD Login.	0:45	20:15	21:00
15/09/2004	TS	Criação do plano de testes para o CUD Login.	0:55	0:15	1:10

Figura 4.3: Excerto de um registro de horas utilizado no estudo.

Para a coleta de dados referentes ao esforço, utilizamos uma planilha em que os participantes deveriam registrar o tempo gasto e descrever sucintamente a atividade executada. A Figura 4.3 apresenta um excerto do registro de horas utilizado no estudo. Cada linha contém a data de realização da atividade, o fluxo (ES = Estudo do processo, TS = Teste, DS = Desenho), e duração, calculada automaticamente a partir do horário de início e fim da atividade. Utilizando esse formulário foi possível calcular o tempo gasto pelos participantes para cumprir as atividades do estudo.

4.1.8 Operação

Antes de iniciarmos o estudo fizemos uma breve apresentação, aos participantes, sobre os tratamentos envolvidos e as atividades associadas ao estudo, sem deixar claro quais eram exatamente as hipóteses envolvidas. Garantimos que seria preservado o anonimato dos estudantes, com relação aos dados do estudo, explicando ainda como eles seriam utilizados. Todos concordaram com o estudo, no entanto, não achamos necessário documentar essa aprovação por meio um termo de consentimento.

O estudo experimental foi divido em duas etapas, uma para cada caso de uso, Login e Gestão de Mercadorias. Cada etapa continha dois passos: 1 - desenho do caso de uso, 2 - teste do caso de uso. Em cada um dos passos foi solicitado que os participantes preenchessem um formulário com o registro das atividades realizadas e o tempo total dessas atividades. Cada passo foi realizado em uma semana. Dentro dessa semana os participantes tinham que enviar diariamente os resultados produzidos e os registros de horas gastos em cada atividade.

O desenho dos casos de uso se ateve às atividades *Desenho das interfaces*, *Detalhamento*

dos casos de uso, *Desenho das entidades* e *Realização dos casos de uso*, uma vez que foi fornecido um modelo de desenho já contendo algumas definições de desenho, como por exemplo a definição da arquitetura e do mecanismo de persistência utilizado, restringindo as tarefas de desenho a essas quatro atividades. Ao final de cada passo os resultados eram entregues. Durante o desenho dos casos de uso, os resultados eram submetidos a revisão, para garantir que todos fizessem o desenho por completo, evitando assim termos uma falsa medida do tempo gasto durante essa etapa. Os autores deste trabalho foram os responsáveis pela revisão dos modelos.

Com relação às atividades de teste, os participantes tiveram que *desenhar* os testes, produzindo uma especificação de testes para o caso de uso, juntamente com a *implementação* dessa especificação, gerando uma bateria de testes, e a sua *execução*. A análise da capacidade de detecção de falhas dos testes foi feita utilizando a semeadura de erros. A execução dos testes nesse mecanismo gerava um resumo contendo a indicação das falhas detectadas por cada bateria de testes executada. Os participantes não sabiam quais falhas existiam, eles apenas tinham que utilizar o mecanismo de execução para gerar o arquivo com o resultado. Somente após a finalização de uma etapa do estudo era divulgada a descrição das faltas inseridas na implementação dos casos de uso.

No mecanismo de avaliação da capacidade de detecção de falhas, os testes dos participantes eram executados para o sistema original, sem falhas, e para cada uma das versões do sistema com falha.

Nós planejamos o uso da semeadura de erros para avaliar a qualidade dos testes, mais especificamente, a capacidade de detecção de falhas. Semeamos 29 faltas para o caso de uso Login e 45 faltas para o caso de uso Gestão de Mercadorias. Essas faltas foram criadas com base no histórico de falhas comumente encontradas nos trabalhos das disciplinas de Engenharia de Software e com o auxílio de um testador experiente. As figuras 4.4 e 4.5 exibem um resumo das faltas inseridas no estudo. Existem sete tipos de falhas diretamente relacionadas ao esquema de classificação do IEEE (IEEE, 1993), conforme detalhado na Tabela 4.1. Nessa tabela apresentamos a classificação de falhas proposta pelo IEEE, nossa classificação, e a descrição resumida da falha. Conforme descrito, as faltas inseridas são faltas associadas a falhas comuns no desenvolvimento de software.

Não foi solicitado para os participantes a codificação dos casos de uso selecionados para o estudo. Os participantes utilizaram, nas execuções dos testes, um mesmo código base, produzido pelos autores do estudo. Isso viabilizou a semeadura de erros, além de garantir que os resultados não foram perturbados por eventuais diferenças nas codificações dos participantes.

Os participantes receberam como insumo para execução das atividades do estudo a especificação de requisitos dos casos de uso selecionados, assim como o modelo de desenho para uso (MEP e NEP, dependendo da etapa do estudo) e o gabarito para especificação e implementação dos testes.

Dos 34 participantes, três foram eliminados por não terem completado todos os passos, tendo eles desistido não só do estudo mas também da disciplina. Dois participantes abandonaram o estudo durante sua execução e outros quatro participantes foram descartados por

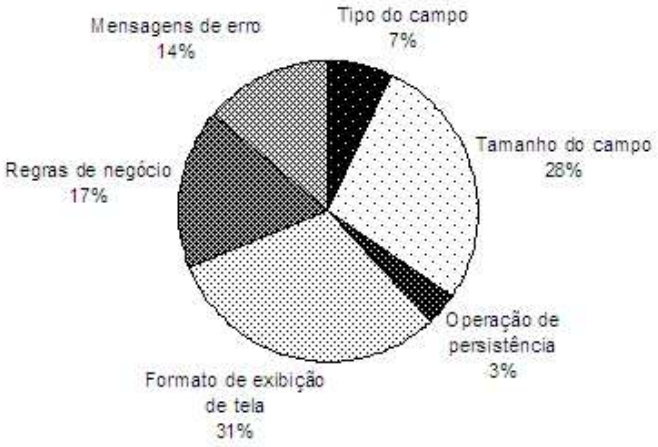

Figura 4.4: Resumos das faltas semeadas no caso de uso Login.

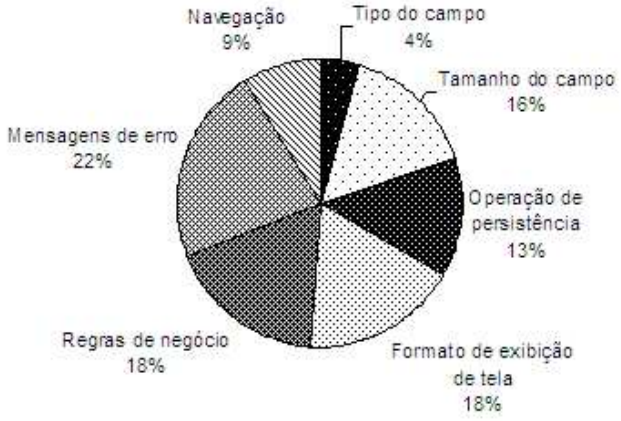

Figura 4.5: Resumos das faltas semeadas no caso de uso Gestão de Mercadorias.

ter sido constatado, no acompanhamento do estudo, que seus dados não eram apropriados. Isso se deu por que alguns participantes utilizaram modelos desenvolvidos por outros participantes para servir de base para o seu modelo. Os dados dos 25 participantes que restaram constituíram os dados utilizados na análise estatística descrita a seguir.

4.1.9 Análise e interpretação

4.1.9.1 Análise quantitativa

Utilizamos o diagrama de caixa (*box-plot*) para visualizar a dispersão e variação dos dados coletados. Esse diagrama pode ser elaborado de diferentes formas. Selecionamos uma abordagem seguida por Warmer e Pfleeger (1996). Utilizamos um valor que é o tamanho da caixa (*box*) multiplicado por 1.5 e somado e subtraído do maior e menor quartil respectivamente. As

Classificação IEEE	Categoria	Descrição da falha
Problema de entrada - Entrada errada aceita	Tipo do campo	Essa categoria de falhas está relacionada a faltas que consistem em permitir o uso de um tipo de dados errado em um campo, como por exemplo, o uso de um dado não alfabético em um campo exigindo apenas caracteres alfabéticos.
Problema de entrada - Entrada errada aceita	Tamanho do campo	Essa categoria de falha está associada a faltas que consistem em permitir o uso de um dado de tamanho incorreto em um campo, como por exemplo, utilizando uma senha com tamanho menor que o mínimo exigido.
Problema de saída - Resultado incorreto	Operação de persistência	Essa categoria de falha está associado a faltas que alteram uma operação de persistência, causando um funcionamento incorreto, como por exemplo, alterar a operação "Delete"para não remover o elemento solicitado.
Problema de saída - Resultado incorreto	Forma de exibição da tela incorreto	Essa categoria de falha está associada a faltas que consistem em alterar a aparência de uma tela, mostrando, por exemplo, como ativos, comandos que deveriam estar inativos.
Problema de saída - Resultado incorreto	Regras de negócio	Essa categoria de falha está associada a faltas que consistem em alterar a implementação de regras de negócio, como por exemplo, permitir a remoção de mercadorias contendo pedidos pendentes, quando a regra de negócio não permite isso.
Erro em mensagem do sistema	Mensagem de erro	Essa categoria de falha está associada a faltas que consistem em alterar as mensagens de erro exibidas pelo sistema.
Outros	Navegação	Essa categoria de falha está associada a faltas que consistem em alterar a navegação dentro de um caso de uso ou entre telas dando acesso a diferentes casos de uso.

Tabela 4.1: Sumário das falhas utilizadas no estudo com classificação do IEEE.

figuras 4.6, 4.7 e 4.8 mostram os diagramas de caixa para os dados do estudo. Não obtivemos participantes com resultados acima ou abaixo do esperado (outliers). Com isso, realizamos a análise quantitativa sem ter que nos preocupar com a eliminação de dados de participantes.

A Figura 4.6 evidencia que o tempo para desenho utilizando o MODEST é maior que o tempo para desenho sem utilizá-lo. Isso era esperado, uma vez que o uso do MODEST requer a modelagem de mais informação, o que gera um maior esforço durante o desenho. A Figura 4.7 evidencia que o tempo para desenho e teste utilizando o MODEST é menor que o tempo para execução das atividades de teste sem o MODEST. O uso do MODEST possibilita a automação de parte das atividades de teste, reduzindo o esforço para sua execução. A Figura 4.8 evidencia que a capacidade de detecção de falhas dos testes automáticos (MEP) é maior que a capacidade de detecção de falhas dos testes gerados manualmente pelos participantes (NEP). Os testes gerados automaticamente pelo MODEST foram mais efetivos para a detecção das falhas associadas às faltas inseridas no sistema.

As figuras 4.9 e 4.10 mostram um resumo dos dados obtidos no estudo experimental. Analisando esses dados podemos notar que os participantes utilizando o MEP dedicaram um esforço total menor que os participantes que utilizaram o NEP. Ainda se compararmos apenas o tempo de desenho MEP com o tempo de teste NEP observaremos que o esforço MEP foi menor. Observe que nessas tabelas não existe o tempo relativo às atividades de teste para os participantes MEP. Isso acontece por que o uso do MEP e da MODESToo desloca parte do esforço de teste para o desenho. Essa parte do esforço deslocada é reduzida em função do uso da MODESToo, que automatiza tarefas do MEP que no NEP são manuais. O tempo para geração e execução dos testes para os participantes MEP foi contabilizado dentro do tempo gasto no desenho. O esforço que não é deslocado para o desenho está relacionado principalmente ao planejamento e análise dos resultados da execução dos testes, atividades essas não presentes no estudo.

Durante o estudo os participantes submeteram, para revisão dos autores, os desenhos.

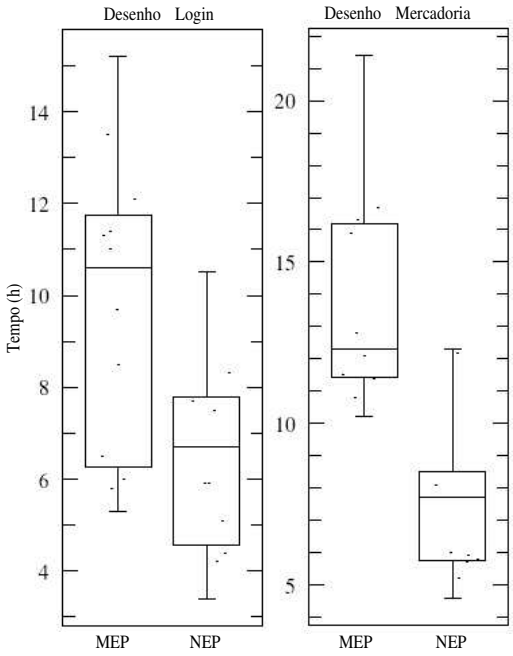

Figura 4.6: Diagrama de caixa (*box-plot*) do esforço para o desenho dos casos de uso.

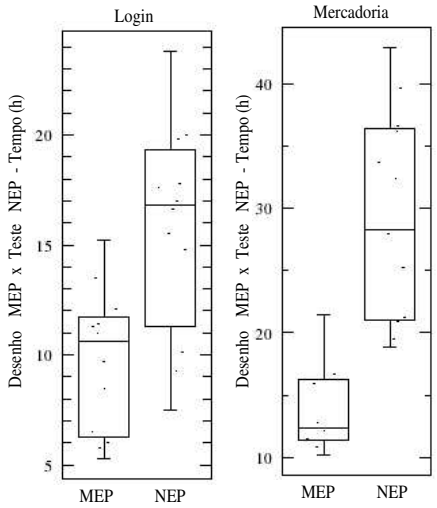

Figura 4.7: Diagrama de caixa (*box-plot*) do esforço para o desenho MEP e teste NEP.

Após a revisão, os desenho que não estivessem completos, tinham que ser refeitos e submetidos novamente. Os desenhos dos participantes só eram considerados completos quando atendessem a todos os itens das respectivas listas de conferência, NEP (Apêndice A) e MEP (Apêndice B). Por causa disso, a cada nova submissão os registros de horas dos participantes tinham que ser atualizados, com o tempo adicional para correção dos defeitos apontados. O MODEST e a MODESToo foram construídos observando critérios relacionados às faltas que foram inseridas, e portanto não é surpreendente que, uma vez considerado completo o desenho, todas as falhas fossem detectadas. Isso é mostrado na Figura 4.8, onde os participantes MEP colapsam em

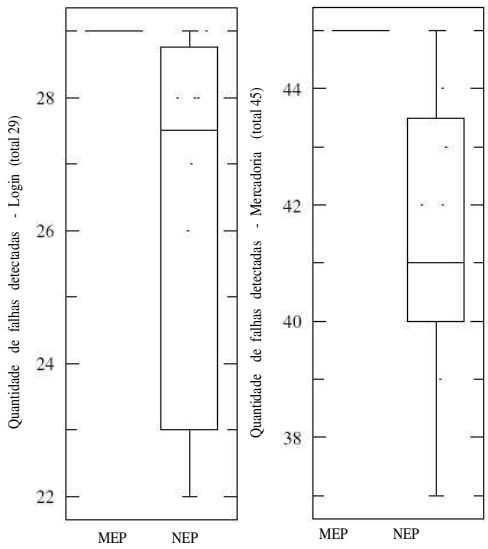

Figura 4.8: Diagrama de caixa (*box-plot*) do número de falhas detectadas.

Participante MEP	Tempo de Desenho do Login (horas)	Falhas Detectadas (29 falhas)	Participante NEP	Tempo de Desenho do Login (horas)	Tempo de Teste do Login (horas)	Total (horas)	Falhas Detectadas (29 falhas)
1	5,8	29	1	4,2	19,8	24,0	28
2	5,3	29	2	5,1	23,8	28,9	28
3	12,1	29	3	3,4	10,1	13,5	22
4	6,5	29	4	7,8	20,0	27,8	23
5	15,2	29	5	5,9	17,6	23,5	26
6	13,5	29	6	7,8	17,8	25,6	27
7	9,7	29	7	4,4	15,5	20,0	29
8	11,0	29	8	8,3	17,0	25,3	23
9	6,0	29	9	7,7	9,3	17,0	29
10	11,4	29	10	5,9	7,5	13,4	29
11	8,5	29	11	10,5	14,8	25,3	22
12	11,3	29	12	7,5	16,6	24,1	28
13	10,6	29					

Figura 4.9: Resumo dos dados coletados para o caso de uso Login.

Participante MEP	Tempo de Desenho de Gestão de Mercadorias (horas)	Falhas Detectadas (45 falhas)	Participante NEP	Tempo de Desenho de Gestão de Mercadorias (horas)	Tempo de Teste de Gestão de Mercadorias (horas)	Total (horas)	Falhas Detectadas (45 falhas)
1	10,8	45	1	4,6	28,0	32,5	39
2	12,3	45	2	5,2	21,2	26,4	42
3	10,2	45	3	8,5	33,7	42,2	41
4	11,5	45	4	8,1	18,8	26,9	42
5	16,3	45	5	12,3	28,3	40,6	45
6	21,4	45	6	7,7	39,6	47,3	44
7	11,4	45	7	5,7	36,2	41,9	40
8	16,7	45	8	7,7	19,5	27,2	45
9	12,1	45	9	12,2	20,9	33,1	41
10	12,3	45	10	5,9	32,4	38,3	43
11	15,9	45	11	5,8	36,6	42,3	40
12	12,8	45	12	8,5	42,9	51,4	41
			13	6,0	25,2	31,2	37

Figura 4.10: Resumo dos dados coletados para o caso de uso de Mercadorias.

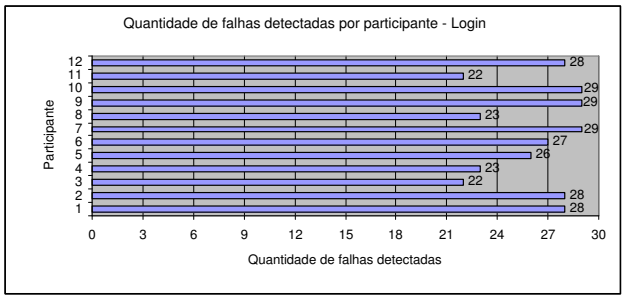

Figura 4.11: Quantidade de falhas detectadas por participante NEP - Login.

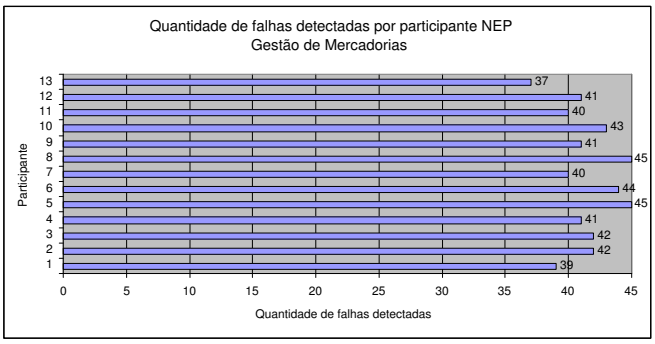

Figura 4.12: Quantidade de falhas detectadas por participante NEP - Mercadorias.

uma linha indicando que eles detectaram todos as falhas.

Caso não revisássemos os desenhos entregues, provavelmente os participantes MEP não teriam descobertos todas as falhas associadas às faltas inseridas, visto que nem todos os aspectos exigidos pelo MODEST seriam modelados, no entanto, o tempo para criação das especificações de desenho seriam reduzidos. Dessa forma, estaríamos avaliando muito mais a capacidade dos participantes em modelar seguindo as prescrições MODEST que propriamente a capacidade de detecção de falhas dos testes gerados. Esse foi o principal fator que nos levou a realizar o estudo dessa forma. Sabemos que isso pode ter gerado um viés no estudo, mas não encontramos voluntários para realização dessas tarefas, em virtude do grande esforço necessário para sua realização.

Embora a quantidade de falhas detectadas pelos participantes utilizando NEP esteja próxima do número total de falhas, os participantes utilizando MEP descobriram mais falhas. As quantidades de falhas detectadas pelos participantes NEP são exibidas graficamente nas figuras 4.11 e 4.12. Os participantes utilizando MEP conseguiram detectar todas as falhas associadas a faltas inseridas na implementação dos casos de uso utilizados no estudo.

É importante ressaltar que as faltas inseridas no estudo eram faltas simples, associadas a falhas comuns no desenvolvimento de software, podendo ser detectadas por testes gerados utilizando os critérios de particionamento em classes de equivalência, análise de valor limite e os critérios definidos por Andrews et al. (2003). Isso foi evidenciado pelo fato de alguns participantes NEP terem encontrado todas as falhas. No entanto, uma hipótese muito explo-

	User			LoginHandler			MainWindow			Util		
Participante	Vivos	Mortos	Escore	Vivos	Mortos	Escore	Vivos	Mortos	Escore	Vivos	Mortos	Escore
Participante 1	0	58	100%	0	5	100%	6	118	95%	0	72	100%
Participante 2	0	58	100%	0	5	100%	10	114	92%	1	71	99%
Participante 3	2	56	97%	2	3	60%	14	110	89%	11	61	85%
Participante 4	1	57	98%	1	4	80%	8	116	94%	1	71	99%
Participante 5	6	52	90%	0	5	100%	13	111	90%	2	70	97%
Participante 6	2	56	97%	1	4	80%	13	111	90%	2	70	97%
Participante 7	0	58	100%	0	5	100%	12	112	90%	5	67	93%
Participante 8	0	58	100%	1	4	80%	13	111	90%	1	71	99%
Participante 9	0	58	100%	0	5	100%	14	110	89%	11	61	85%
Participante 10	0	58	100%	1	4	80%	15	109	88%	1	71	99%
Participante 11	1	57	98%	0	5	100%	4	120	97%	9	63	88%
Participante 12	0	58	100%	0	5	100%	5	119	96%	1	71	99%
Testes gerados via MODESToo	18	40	69%	0	5	100%	4	120	97%	0	72	100%

Figura 4.13: Resultado da Análise de Mutantes para o caso de uso Login.

rada na Análise de Mutantes, o efeito de acoplamento (coupling effect) (DeMillo et al., 1978) diz que faltas complexas estão relacionadas a faltas simples, conforme confirmado por alguns estudos empíricos (Offutt, 1989; Budd et al., 1980).

Para reduzir as ameaças à validade do estudo tentamos utilizar uma outra medida para avaliar a qualidade dos testes: o escore de mutação, obtido a partir da Análise de Mutantes. Selecionamos duas ferramentas de mutação em Java que pareciam viáveis de serem utilizadas no estudo: MuJava (Ma et al., 2005) e Jester [2]. A ferramenta Jester implementa apenas dois operadores de mutação: (i) a troca de 0s por 1s e (ii) a troca de predicados com VERDADEIRO e FALSO. Fizemos algumas avaliações do uso do Jester para medir a qualidade de testes e notamos que, em função da simplicidade dos operadores, todos mutantes eram mortos pelos testes.

A MuJava implementa um bom conjunto de operadores de mutação e por esse motivo resolvemos utilizá-la. No entanto, tivemos que converter todos os testes para um formato específico da ferramenta. Os testes de login executaram sem maiores problemas, porém, não conseguimos executar todos os testes de mercadoria. A MuJava não finaliza a execução de diversos testes. Esse problema ocorre por causa de algumas mutações realizadas ou, mais comumente, devido ao alto consumo de memória imposto pelo seu mecanismo de funcionamento. Como a quantidade de testes de mercadoria, assim como a quantidade de mutantes gerados para esse caso de uso, foi bem maior do que no Login, ainda não foi possível executá-los por completo. Ainda estamos estudando possíveis soluções para esse problema.

A Figura 4.13 apresenta os resultados parciais obtidos durante a Análise de Mutantes para o caso de uso Login. A figura apresenta a quantidade de mutantes não equivalentes, mutantes vivos e mutantes mortos em cada uma das classes relacionadas ao caso de uso Login, juntamente com o escore de mutação obtido pelo teste na respectiva classe. Não existe uma diferença significativa entre os resultados dos testes gerados manualmente pelos participantes do estudo quando comparados com os testes gerados pela MODESToo. No entanto, podemos notar que os testes gerados pela MODESToo deixaram vários mutantes da classe User vivos. Isso aconteceu por que existem diversas mutações associadas aos construtores das classes que utilizam parâmetros. A classe User possui o construtor sem parâmetros e um construtor que recebe todos os parâmetros para instanciar os atributos da classe. Como a MODESToo

[2] http://jester.sourceforge.net/

Variável	H_0:TempoDesenv(MEP) = TempoDesenv(NEP)		H_0:TempoDesenho(MEP) = TempoDesenho(NEP)		H_0:Qualidade(MEP) = Qualidade(NEP)	
	Login	Gestão de Mercadorias	Login	Gestão de Mercadorias	Login (29)	Gestão de Mercadorias (45)
Teste t	-7,38	-9,31	3,00	5,29	3,58	5,13
Desvio padrão	4,27	6,27	2,68	3	1,97	1,68
Média MEP	9,76	13,6	9,76	13,6	29	45
Média NEP	22,4	37	6,54	7,55	26,2	41,5
Mediana MEP	10,6	12,3	10,6	12,3	29	45
Mediana NEP	24,1	38,3	6,7	7,7	27,5	41
Desvio padrão MEP	3,14	3,27	3,14	3,27	0	0
Desvio padrão NEP	5,22	8,1	2,07	2,45	2,86	2,33

Figura 4.14: Resumo do teste estatístico para os dados coletados.

utiliza apenas o construtor sem parâmetros para inicialização de objetos de uma classe, ela não foi capaz de matar tais mutantes. A atribuição de valores aos atributos pela MODESToo é sempre feita via métodos get e set da própria classe. Analisando a figura, podemos notar que nas outras classes associadas ao caso de uso Login, o escore de mutação dos testes gerados pela MODESToo foi superior, embora estatisticamente não haja uma diferença significativa.

4.1.9.2 Teste de hipóteses

Utilizamos o teste t de Student para realizar o teste de hipóteses. Nesse teste, rejeitamos H_0, se $|t| > t_{\alpha/2,f}$, onde $f = n + m - 2$ é número de graus de liberdade, n é o número de participantes MEP e m o número de participantes NEP. A variável α é o nível de significância e $t_{\alpha/2,f}$ é um valor encontrado em tabelas estatísticas (Wohlin et al., 2000). Utilizamos um nível de significância de 5%, com 23 graus de liberdade (f = 13 + 12 - 2). Assim, $t_{\alpha/2,f} = 2,069$.

A Figura 4.14 apresenta um resumo dos testes estatísticos para cada uma das hipóteses consideradas, para cada um dos casos de uso. A linha denominada *Teste t* apresenta o valor resultante da aplicação do testes estatístico. Conforme mencionado anteriormente, o módulo do resultado do teste t ($|t|$) foi maior que $t_{\alpha/2,f}$ $(2,069)$, assim, a hipótese nula é rejeitada em todos os casos. As demais linhas são auto explicativas.

O tempo para desenho utilizando o MODEST é significativamente maior que o tempo para desenho sem utilizar o MODEST, o que era esperado, uma vez que o desenho utilizando as prescrições MODEST requer maior detalhe dos itens descritos. Por outro lado, o tempo para desenho e teste dos casos de uso é significativamente reduzido utilizando o MODEST.

Com relação à capacidade de detecção de falhas, medida a partir do número de falhas detectadas por participante, constatamos que embora os testes manualmente gerados tivessem alcançado bons níveis de detecção, a capacidade de detecção de falhas dos testes automáticos é significativamente melhor. As especificações de desenho produzidas pelos participantes foram utilizadas para gerar testes automáticos capazes de detectar todas as falhas associadas às faltas inseridas. Conforme discutido anteriormente, isso não é surpreendente, visto que o MODEST foi construído para a geração de teste utilizando os critérios de particionamento em classes de equivalência, análise de valor limite e os critérios definidos por Andrews et al. (2003).

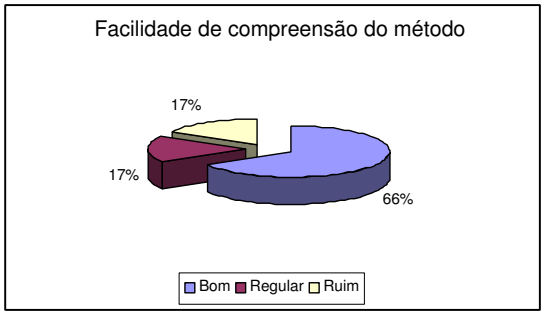

Figura 4.15: Avaliação da compreensibilidade do MODEST.

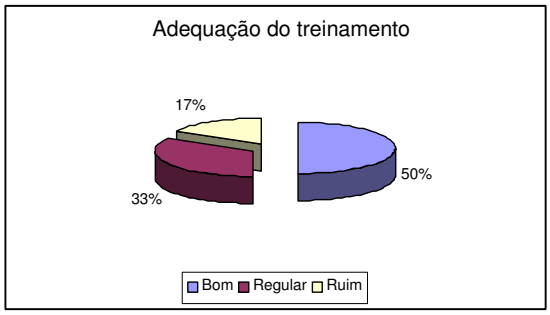

Figura 4.16: Avaliação do treinamento para uso do MODEST.

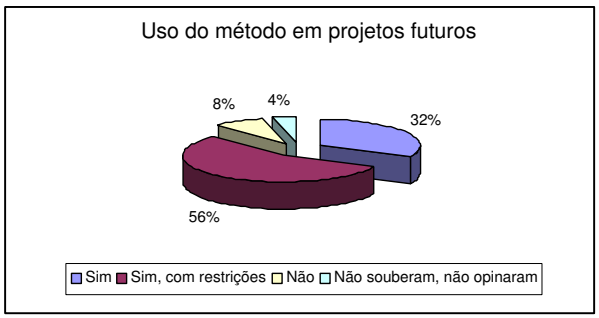

Figura 4.17: Avaliação do uso do MODEST.

4.1.9.3 Análise qualitativa

Após a conclusão do estudo, avaliamos as impressões dos participantes com relação ao estudo realizado, utilizando um questionário exibido no Apêndice C. A seguir detalhamos as respostas dos participantes.

Avaliamos o nível de compreensibilidade do método, com o intuito de saber o grau de dificuldade na utilização do MODEST. Conforme apresentado na Figura 4.15, para 66% dos participantes o método tem um bom nível de compreensibilidade, não sendo difícil aplicá-lo. Para 17% o nível de compreensibilidade é regular, enquanto que outros 17% afirmaram que o método é difícil de usar.

Avaliamos também a adequação dos treinamentos para o estudo, conforme apresentado

na Figura 4.16. Para 83% dos participantes o treinamento ministrado foi adequado (bom ou regular). O restante (17%) achou o treinamento inadequado. Alguns participantes sugeriram melhorias para o treinamento, indicando mecanismos que podem torná-lo mais efetivo. Essas sugestões foram registradas para uso na replicação do estudo.

Avaliamos também a possibilidade de uso do método em projetos futuros em que os participantes estivessem envolvidos para estimarmos o nível de receptividade do método. Conforme mostrado na Figura 4.17, cerca de 32% deles utilizariam o método sem qualquer restrição, enquanto 56% o utilizariam, mas com algumas restrições, como por exemplo o tamanho do projeto, melhor entendimento do funcionamento do método, análise dos resultados em outros contextos e adaptação a certas tecnologias. Cerca de 8% dos participantes afirmaram que não usariam o método, em virtude dele prescrever muita informação sobre o funcionamento do sistema sob teste. Alguns participantes afirmaram não serem capazes de responder a questão, ou não manifestaram sua opinião (4%).

4.1.10 Validade

4.1.10.1 Validade interna

Para evitar efeitos negativos relativos a história (*history*), combinamos os prazos para entrega dos resultados do estudo com antecedência, para evitar falsas expectativas dos participantes. Isso também foi feito para tentar reduzir problemas relacionados à carga de tarefas dos estudantes. Embora isso tenha sido estimado, durante a execução do estudo parte dos estudantes queixou-se sobre a dificuldade de cumprimento dos prazos. Para evitar distorções nos resultados, as atividades de uma das etapas foram deslocadas para um período menos intenso de provas e trabalhos dos estudantes.

Para evitarmos o problema da maturação, utilizamos dois casos de uso com características bem distintas um do outro. Com isso, procuramos evitar uma influência positiva na utilização dos tratamentos, visto que eles tinham pouco em comum. Acreditamos que isso também ajudou a evitar influências negativas, como cansaço e tédio, na repetição de uma atividade muito similar.

Para análise da capacidade de detecção de falhas dos testes utilizamos o mecanismo de semeadura de erros e um arcabouço para execução, que gerava automaticamente o resultado. Os participantes não sabiam quais falhas tinham sido geradas, eles apenas executavam seus testes nesse arcabouço. Apenas após a conclusão do estudo divulgamos a descrição das faltas inseridas, na tentativa de evitar aprendizado indesejado. Da mesma forma, as faltas inseridas no estudo foram selecionadas com a ajuda de um testador experiente. O MODEST e a MODESToo não foram adaptados para atacar especificamente as faltas selecionadas: eles foram desenvolvidos utilizando critérios que prescrevem a geração de testes para a maioria das falhas comuns no desenvolvimento de software. Isso explica a capacidade de detecção do método.

Os artefatos utilizados para coleta de dados já tinham sido utilizados em dois outros estudos, conforme apresentado na Seção 4.1.11, sendo aperfeiçoados para este estudo.

Com relação à mortalidade, a Subseção 4.1.8 apresenta os números relativos aos estudantes que participaram do estudo. Dentre os participantes que abandonaram o estudo não encontramos nenhuma caracterização específica, que por ventura apontasse uma classe representativa de participantes.

Para evitarmos ameaças à validade do estudo relacionadas a um único grupo, separamos os participantes em dois grupos, aplicando os dois tratamentos a cada um dos grupos. A distribuição dos alunos com conhecimentos prévios em alguns conceitos de teste foi equivalente e, além disso, tais alunos não tiveram um comportamento diferenciado nos seus grupos. A ordem de atribuição dos participantes aos tratamentos foi aleatória, conforme descrito anteriormente. Fizemos isso para evitar o equalização compensatória dos tratamentos (*compensatory equalization of treatments*), rivalidade compensatória (*compensatory rivalry*) e desmoralização por ressentimento (*resentful demoralization*) (Wohlin et al., 2000).

4.1.10.2 Validade externa

Planejamos o estudo envolvendo alunos de Engenharia de Software do último ano do curso. Conforme mencionado na Subseção 4.1.5, alguns alunos já trabalhavam em empresas de desenvolvimento de software, porém, esses alunos estavam trabalhando com atividades relacionadas à implementação. Assim, acreditamos que eles têm o perfil típico de um desenvolvedor atuando na indústria, porém com pouca experiência.

Já foram realizados, por duas vezes, estudos experimentais semelhantes ao que aqui descrevemos, porém sem o rigor necessário para reduzir as ameaças à sua validade (Seção 4.1.11). Ainda assim, os resultados obtidos foram similares. Os dados obtidos nos estudos anteriores seguem o mesmo padrão de comportamento.

O exemplo utilizado, dois casos de uso de um sistema para controle de mercearias, embora seja pequeno, possui as características comumente existentes nos sistemas de informação categorizados como Sistemas de Processamento de Transações (Cook, 1996b). Assim, acreditamos que as conclusões obtidas no estudo possam ser estendidas para SPTs com tamanho usual. Mesmo sendo um estudo envolvendo apenas dois casos de uso, o tempo total empregado pelos participantes foi acima de 1.000 horas, o que tornou o estudo muito oneroso no cenário aqui descrito e o tornaria praticamente inviável em um cenário industrial, sem o patrocínio adequado.

Durante nosso estudo notamos que o número de testes gerados automaticamente pela MODESToo foi comparável ao número de testes gerado manualmente pelos voluntários. Estamos planejando outros estudos envolvendo softwares maiores e mais complexos para avaliar com maior precisão a escalabilidade do MODEST, mas a principio acreditamos que isso seja mantido, uma vez que a MODESToo gera testes utilizando critérios de teste comumente utilizada por testadores atuando na indústria de software.

4.1.10.3 Validade de conclusão

A verificação das hipóteses foi feita utilizando o teste t de Student. A potência desse teste estatístico é considerada alta (Wohlin et al., 2000).

Conforme apresentado na Subseção 4.1.5, os participantes do estudo praticamente não possuíam conhecimentos prévios nos objetos utilizados. Em função dessa homogeneidade, acreditamos que os resultados observados se deram, em sua maior parte, devido aos tratamentos utilizados, e não por causa de outros fatores não considerados.

No intuito de minimizarmos os efeitos associados à imprecisão das medidas relacionadas ao esforço empregado nos passos do estudo, solicitamos o envio diário dos resultados intermediários e do formulário com o registro de horas relativos a esses resultados. Esses formulários eram analisados para verificar uma eventual imprecisão dos dados coletados. Além disso, acompanhamos cada passo do estudo (follow-up), e, em função desse acompanhamento, descartamos participantes cuja qualidade dos registros não foi considerada adequada.

Foram utilizados os números de ordem dos estudantes na caderneta da disciplina para selecionar a atribuição de tratamentos. Além disso, realizamos o estudo em duas etapas invertendo os tratamentos utilizados por cada grupo. Os resultados obtidos foram próximos, conforme apresentado na Subseção 4.1.9. Conforme mencionado anteriormente, os participantes que já haviam trabalhado com testes não tiveram um comportamento diferenciado no estudo.

O uso do MODEST é mais adequado quando integrado a processos de desenvolvimento que enfatizam o uso de modelos, visto que tais processos já prescrevem boa parte da informação exigida pelo método. As conclusões relacionadas a este estudo, principalmente as que envolvem análise de esforço, estão relacionadas ao uso de um processo com essas características. Não acreditamos que seja válido estender nossas conclusões para processos onde não haja ênfase no uso de modelos, como por exemplo, nos processos ágeis.

No estudo, solicitamos que os participantes fizessem a especificação dos testes (desenho) antes da sua implementação, conforme recomendado por alguns padrões (IEEE, 1998) e pelo processo utilizado. Além disso, a especificação dos testes antes da implementação contribui para uma melhor qualidade dos testes gerados. No entanto, sabemos que essa abordagem nem sempre é utilizada na prática (Neto et al., 2006).

Conforme mencionado anteriormente, as faltas inseridas para avaliação da capacidade de detecção dos testes gerados foram criadas com base no histórico de falhas comumente encontradas nos trabalhos da disciplina, com o auxílio de um testador experiente, que estava realizando mestrado na UFMG e pertencia ao nosso grupo de pesquisa. A maioria dessas falhas são falhas simples de serem descobertas, sendo detectadas, por exemplo, por testes gerados utilizando os critérios de particionamento em classes de equivalência, análise de valor limite e os critérios definidos por Andrews et al. (2003). A realização do desenho dos casos de uso contemplando todas as prescrições MODEST, possibilita a geração de testes com a capacidade de detectar todas essas falhas. Fizemos uma revisão dos artefatos gerados pelos participantes MEP para garantir que todas as prescrições tinham sido atendidas. Fizemos

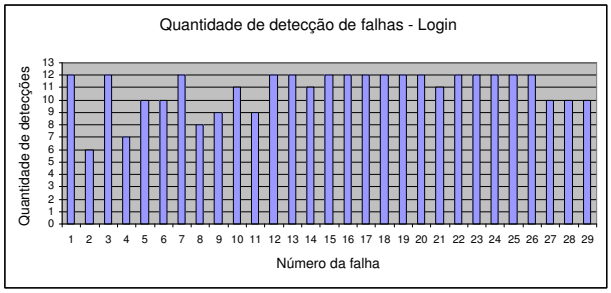

Figura 4.18: Quantidade de detecção por falha no caso de uso Login.

isso para garantir que estávamos avaliando a capacidade de detecção de falhas com o uso do método ao invés de avaliar a capacidade dos participantes em criar as especificações de desenho no formato exigido pelo MODEST.

Tentamos realizar uma avaliação da qualidade dos testes gerados utilizando a Análise de Mutantes, uma vez que tal medida é menos subjetiva que a semeadura de erros. Isso nos auxiliaria a reduzir a ameaça à validade das conclusões, porém, conforme mencionado no final da Subseção 4.1.9.1, não obtivemos sucesso nessa tentativa, em virtude de diversos problemas relacionados ao uso de uma ferramenta de mutação em Java.

4.1.10.4 Validade de construção

As idéias utilizadas no estudo já tinham sido utilizados em nossos estudos anteriores (Seção 4.1.11), sendo aperfeiçoadas para o estudo atual. Assim acreditamos que elas foram suficientemente definidas antes de serem traduzidas em tratamentos e medidas, conforme apresentado na Subseção 4.1.2. Essas medidas são bastante claras e amplamente conhecidas no contexto de Engenharia de Software e, além disso, desde os primeiros estudos se mostraram adequadas.

Utilizamos dois casos de uso para reduzirmos possíveis problemas relacionados à aplicação dos tratamentos em um único objeto. Os casos de uso selecionados eram bem diferentes, forçando a utilização de muitos conceitos associados aos tratamentos, porém com pouca interseção durante as etapas.

Planejamos o estudo de forma que os participantes não tivessem conhecimento das hipóteses a serem verificadas, de forma a evitar comportamentos positivos e negativos, pois sabíamos da presença de dois participantes defensores de processos ágeis. Esses alunos participaram do estudo em grupos separados (pois um tinha número par e outro número ímpar), mas não foi notado qualquer diferença significativa nos seus dados. Garantimos o anonimato da divulgação dos dados dos participantes para evitar apreensões que poderiam levar a desvios de resultados.

O uso somente da semeadura de erros para avaliar a capacidade de detecção de falhas dos testes pode gerar desvios nos resultados. No entanto, acreditamos que isso não ocorreu no estudo, uma vez que, analisando os resultados obtidos, podemos notar que as faltas inseridas não continham artifícios, uma vez que as falhas associadas foram descobertas por várias

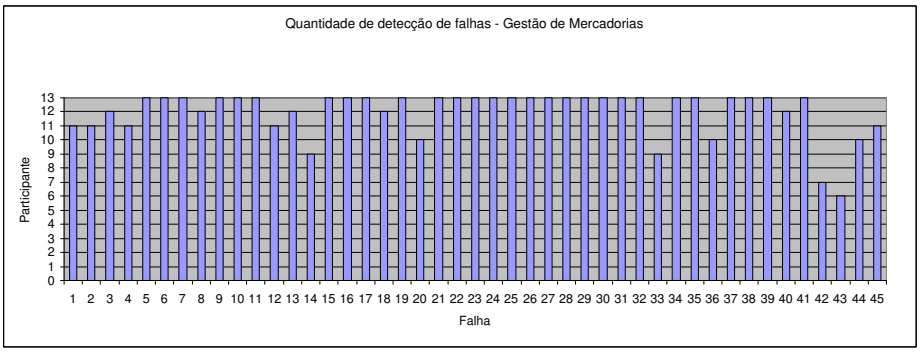

Figura 4.19: Quantidade de detecção por falha no caso de uso de Mercadorias.

baterias de testes manuais produzidas pelos participantes, conforme apresentado nas figuras 4.18 e 4.19. Podemos notar que a falha menos detectada no caso de uso Login foi a falha de número dois, que foi detectada por 50% dos participantes utilizando NEP. Para o caso de uso Gestão de Mercadorias, a falha de número 43 foi detectada por seis dos 13 participantes NEP (46%). Além disso, as faltas inseridas eram bastante representativas, visto que elas foram extraídas da base de dados histórica associada à disciplina.

4.1.11 Estudos anteriores

Antes da execução do estudo experimental descrito neste capítulo já havíamos realizado dois outros estudos, porém sem seguir o rigor exigido para tal tipo de estudo. Esses estudos serviram como forma de nos preparar para o estudo mais formal. Nesta seção apresentamos uma parte desses estudos anteriores, mencionando os principais erros cometidos nessas tentativas.

Os estudos tinham o mesmo objetivo: caracterizar o uso do MODEST no desenvolvimento de parte de um SI, analisando o esforço relacionado ao seu uso, e qualidade dos testes gerados, medidos em termos de capacidade de detecção de falhas utilizando a semeadura de erros.

Na primeira iniciativa, realizada no primeiro semestre de 2004, utilizamos 16 alunos da disciplina de Engenharia de Software. Ao longo do estudo, 10 alunos abandonaram o estudo e apenas seis concluíram todas as atividades. Três alunos utilizaram o MEP e três utilizaram o NEP para desenhar e testar o caso de uso Login. Os dados relativos ao esforço empregado pelos participantes são apresentados na Tabela 4.2. A Figura 4.20 apresenta os dados relativos à detecção de falhas. Inserimos 30 faltas no caso de uso login.

Diversos problemas no estudo o tornaram inválido: a mortalidade foi acima dos limites toleráveis, não foi realizado uma análise no perfil dos participantes, o acompanhamento utilizado foi muito mal idealizado, além de não ter sido avaliada a dispersão dos dados, que foi muito grande.

Na segunda iniciativa, realizada no segundo semestre de 2004, utilizamos 16 alunos da disciplina de Engenharia de Software, e todos concluíram as atividades do estudo. Metade utilizou o MEP e metade utilizou o NEP para desenhar e testar o caso de uso Login, de forma muito parecida com o primeiro estudo. Os dados relativos ao esforço empregado pelos

MEP			NEP		
Part1	**Part2**	**Part3**	**Part4**	**Part5**	**Part6**
3,50	5,32	3,72	9,75	18,47	20,31

Tabela 4.2: Esforço empregado na primeira tentativa de estudo experimental (tempo em horas).

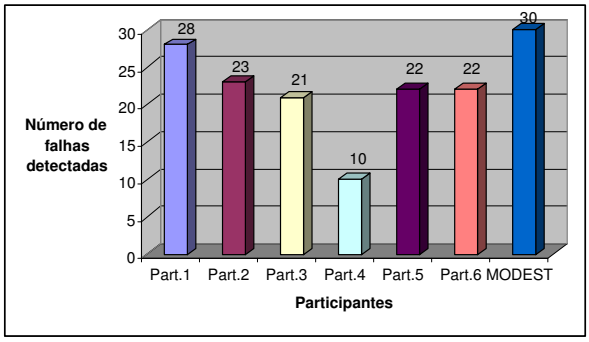

Figura 4.20: Falhas detectadas por participante na primeira tentativa de estudo experimental.

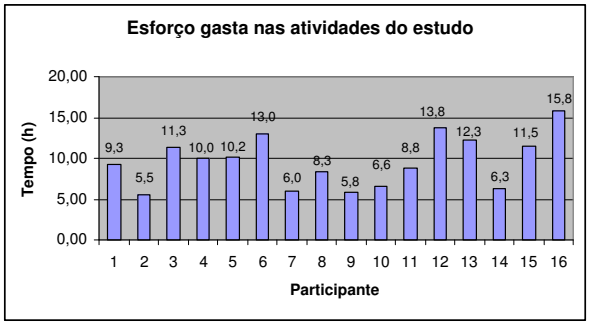

Figura 4.21: Esforço empregado por participante na segunda tentativa de estudo experimental (tempo em horas).

participantes são apresentados na Figura 4.21. A Figura 4.22 apresenta os dados relativos à detecção de falhas. Também neste estudo foi utilizada a inserção de 30 faltas no caso de uso login.

Alguns erros cometidos neste estudo foram diferentes dos erros cometidos no estudo anterior. Dentre esses erros o principal foi o fato de dedicarmos mais horas de treinamento para os participantes utilizando o MODEST. Além disso, mais uma vez não realizamos uma análise no perfil dos participantes e o acompanhamento utilizado não permitiu termos uma adequada noção de como as atividades estavam sendo realizadas. Outro ponto discutível no estudo foi o fato de penalizar alguns estudantes, visto que metade deles utilizou o MODEST e outra metade não. Isso pode ter servido como um desmotivador e desta forma ter gerado as diferenças registradas.

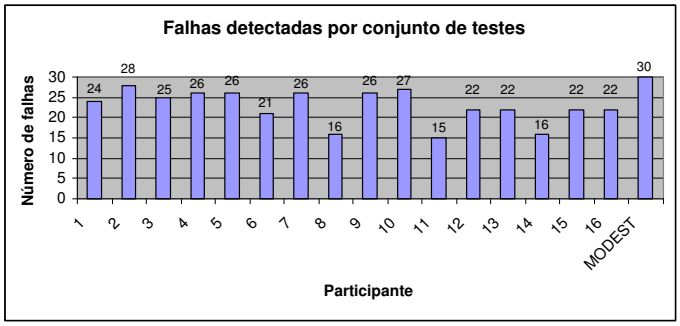

Figura 4.22: Falhas detectadas por participante na segunda tentativa de estudo experimental.

4.1.12 Considerações finais

Neste capítulo apresentamos um estudo experimental realizado com o intuito de caracterizar o MODEST, focalizando em aspectos relacionados ao esforço do seu uso e possíveis ganhos de capacidade de detecção de falhas nos testes gerados. O estudo mostrou que o MODEST pode reduzir o custo do desenvolvimento, em função da redução do esforço exigido durante a construção do software. Embora o uso do método aumente o tempo no desenho, ele reduz o esforço exigido nas atividades de teste. Essa redução propicia uma diminuição geral no tempo de desenvolvimento, que normalmente acarreta em ganhos em termos de custos.

O uso do método pode trazer uma maior capacidade de detecção de falhas. Isso foi evidenciado no estudo, que mostrou que os testes gerados automaticamente, utilizando o MODEST, foram mais eficazes na detecção de falhas no sistema que os testes manualmente produzidos pelos participantes.

As falhas detectadas a partir do uso do MODEST não eram falhas triviais, visto que, embora alguns participantes tenham detectado todas as falhas, nem todos conseguiram detectá-las por completo. Adicionalmente, cada falta injetada foi detectada por pelo menos 46% dos participantes, mostrando que as faltas injetadas não foram introduzidas de maneira artificiosa, que impediriam a sua detecção. No entanto, esse aspecto será melhor investigado quando conseguirmos finalizar a Análise de Mutantes já iniciada, conforme discutido na seção Trabalhos futuros.

Capítulo 5

Conclusão

Muitas organizações não realizam o teste de software, dentre outras razões, por causa do alto custo associado a essa atividade. Além disso é comum que as atividades de teste sejam discutidas de forma desacoplada dos processos de desenvolvimento. Isso nos levou a pesquisar como melhorar as atividades de teste, de forma mais integrada aos processos de software. Resolvemos focalizar nossa pesquisa nos Testes Baseados em Modelos, particularmente, em modelos desenvolvidos seguindo as prescrições do Processo Unificado, descritos utilizando a UML. A escolha do PU e da UML como especificações de processo e linguagem para nossa pesquisa se deu principalmente pelo fato desse processo e linguagem serem bastante disseminados na indústria de software, e também por termos experiência no seu uso em projetos industriais e acadêmicos. As contribuições mais importantes deste trabalho são listadas a seguir:

- **Elaboração de um catálogo de requisitos para o TBM.** Desenvolvemos um catálogo de requisitos para o TBM baseado em trabalhos relacionados, nossa experiência acadêmica e profissional no desenvolvimento e aplicação de métodos de TBM e na realização de oficinas de requisitos com profissionais ligados ao desenvolvimento de software. Esse catálogo pode auxiliar usuários do TBM na seleção de métodos e ferramentas, assim como pode guiar o desenvolvimento de novos mecanismos relacionados ao TBM. O trabalho iniciou com uma quantidade menor de requisitos (Santos-Neto et al., 2005a), e foi sendo estendido (Santos-Neto et al., 2007). O catálogo é abstrato, uma vez que ele se propõe a ser útil em diversos domínios de aplicação, mesmo tendo sido bastante influenciado pela nossa experiência no desenvolvimento de SI.

- **Desenvolvimento de um novo método para o TBM.** Desenvolvemos o MODEST, um método de TBM, que no seu nível conceitual foi idealizado para ser independente de processo e de linguagem de modelagem, no entanto, para efeito de simplificação, segue uma arquitetura pré-definida. Apresentamos como os conceitos do MODEST, no seu nível lógico, podem ser mapeados na UML prescrita pelo PU. Além disso, apresentamos um exemplo da integração do MODEST com um processo baseado no PU. Um importante ganho gerado pelo uso do MODEST é justamente a criação de modelos que são úteis não somente para fins de documentação. Esses modelos podem ser utilizados para

auxiliar a geração parcial de código, como também podem ser utilizados para a geração de testes, aumentando ainda mais seu valor.

- **Avaliação do método.** Realizamos um estudo experimental com o objetivo de caracterizar o uso do método seguindo as recomendações da Engenharia de Software Experimental (Wohlin et al., 2000). Nesse estudo pudemos constatar que o o uso do MODEST no desenvolvimento de parte de um SI pode trazer redução de custos associados ao desenvolvimento, por meio da diminuição do esforço, e, ao mesmo tempo, aumento de qualidade dos testes, por meio da criação de testes com maior capacidade de detecção de falhas.

5.1 Limitações e dificuldades do trabalho

Durante o decorrer do trabalho algumas das dificuldades encontradas geraram limitações. A discussão abaixo apresenta algumas dessas dificuldades no intuito de auxiliar outros trabalhos.

- **Desenvolvimento da MODESTToo**. Diversos fatores dificultaram o desenvolvimento da MODESTToo. Dentre eles destacamos sua complexidade, que é alta, visto que é necessário criar interpretadores de expressões, utilizar objetos de forma indireta e gerar dados com diferentes formatos. O seu desenvolvimento exigiu o estudo aprofundado de algumas tecnologias complexas, estendendo o tempo para a construção do protótipo. Por causa disso a versão desenvolvida ainda é limitada e alguns dos seus componentes ainda não possuem uma versão inicial finalizada, como por exemplo o Test Manager e a parte responsável pela geração e execução de testes de desempenho e estresse. O componente Extractor foi implementado de duas formas diferentes, mas nenhuma delas foi totalmente satisfatória, uma vez que existem diversas limitações em cada uma das abordagens. Atualmente, acreditamos que o advento de tecnologias para transformações entre modelos, prescrita pela MDA (OMG, 2003a), seja a opção mais interessante para implementação da MODESTToo, uma vez que essa tecnologia oferece boa parte do suporte exigido para tal tarefa.

- **Avaliação experimental do MODEST**. Realizamos duas avaliações iniciais do uso do método até atingirmos um nível de conhecimento satisfatório para uma avaliação efetiva. Parte do problema esteve relacionada ao fato dessa área não ser dominada pelos autores deste trabalho. O aprendizado para esse estudo se estendeu por quase dois anos, exigindo muita dedicação de nossa parte e dos alunos envolvidos. Ao longo desse aprendizado identificamos novos estudos experimentais que pretendemos fazer como trabalhos futuros e que são discutidos à frente.

- **Análise de mutantes**. Iniciamos a avaliação da qualidade dos testes gerados pelos participantes do estudo experimental e pela MODESTToo utilizando a Análise de Mutantes. Esse ítem exigiu muito tempo e esforço por uma série de razões: (i) embora existam muitas ferramentas de mutação disponíveis, atualmente existem poucas as ferramentas

para o ambiente Java, e dessas poucas, não encontramos nenhuma para sistemas compostos por várias classes; (ii) tivemos que fazer algumas adaptações no sistema original para que ele pudesse executar juntamente com a ferramenta de mutação selecionada; um exemplo de adaptação foi a ferramenta exigir que nenhuma das classes do SST estivesse na variável de ambiente CLASSPATH, mas a camada de persistência utilizada inicialmente exigia justamente o contrário; (iii) a ferramenta utilizada possui escalabilidade muito limitada; a execução de mutantes na ferramenta causa um consumo excessivo de memória, por que ela instancia diversos objetos para execução dos mutantes mas não interage com o coletor de lixo, para que seja feita a remoção dos objetos; além disso, o número de mutantes gerados foi da ordem de milhares, potencializando ainda mais esse problema.

5.2 Trabalhos futuros

Durante a realização deste trabalho algumas atividades foram identificadas para serem realizadas no futuro, conforme detalhamento a seguir.

- **Extensão do MODEST**. Estamos planejando a extensão do MODEST para contemplar alguns aspectos identificados no catálogo de requisitos para o TBM e não contempladas atualmente. Nesse esforço inicial focalizamos mais os aspectos técnicos, mas certamente entendemos que devemos também focalizar os aspectos gerenciais. Além disso, estamos analisando o impacto da adaptação do MODEST para outros contextos, como por exemplo, sua adaptação para métodos ágeis.

- **Extensão da MODESTtoo**. Conforme mencionado na seção anterior, tivemos alguns problemas durante o desenvolvimento da MODESTtoo. Isso ocasionou um atraso na sua implementação e impossibilidade de concluir alguns componentes. Atualmente estamos estudando tecnologias novas, além de formalizar sua especificação de requisitos e seus detalhes arquiteturais. Isso inclui, dentre outros aspectos, utilizar OCL (Warmer e Kleppe, 2003) como a linguagem para modelagem de restrições, desenvolver a parte responsável pela geração e execução de testes de desempenho e estresse, e criar uma versão mais estável.

- **Avaliação experimental do MODEST**. No estudo experimental realizado com o intuito de caracterizar o uso do MODEST focalizamos nas questões relacionadas ao custo do seu uso e na qualidade dos testes gerados automaticamente, quando comparados com testes gerados de forma manual. A qualidade dos testes foi medida em termos da capacidade de detecção de falhas, utilizando a semeadura de erros. Conforme mencionado no experimento, tentamos utilizar uma outra medida, o escore de mutação. No entanto, as ferramentas de mutação para o ambiente Java não permitiram que essa avaliação fosse concluída. Ainda estamos trabalhando nessa avaliação tentando solucionar os problemas encontrados. Além disso, já estamos planejando um outro estudo experimental,

semelhante ao feito por Sinha e Smidts (2006), com o objetivo de responder às seguintes questões: (i) o MODEST é comparável em termo de usabilidade com outra técnica de TBM? (ii) os testes gerados utilizando MODEST são superiores em desempenho aos testes gerados utilizando uma outra técnica de TBM? Para execução desse estudo temos que selecionar uma outra técnica a ser avaliada, assim como estender a MODESToo. A própria técnica proposta por Sinha e Smidts (2006) é uma candidata. Além disso, conforme já mencionado, estamos trabalhando no empacotamento do experimento, incluindo uma evolução da MODESToo, de formar a facilitar a sua replicação em outros contextos.

- **Extensão do catálogo de requisitos**. Já iniciamos um trabalho cujo objetivo é a extensão do catálogo de requisitos, na forma de um estudo sobre necessidades associadas a ferramentas de auxílio às atividades de teste. Neste trabalho estamos utilizando o QFD (Seção 2.9) para agrupar e priorizar os requisitos identificados, utilizando uma equipe de desenvolvedores multidisciplinar para nos auxiliar nessa tarefa. Atualmente estamos concluindo o QFD de forma a gerar um arcabouço para avaliação de ferramentas com base nos dados identificados. Utilizando esse arcabouço, não somente será possível definir a qualidade projetada para uma ferramenta a ser desenvolvida por uma organização, de forma a atender plenamente as necessidades dos clientes, como também avaliar as ferramentas existentes.

- **Criação de uma ferramenta de mutação de especificações UML**. Durante a fase final deste trabalho concebemos as idéias iniciais para a criação de uma ferramenta de mutação das especificações UML. Utilizando essa ferramenta, poderemos realizar experimentos avaliando a qualidade das especificações, assim como a própria qualidade da MODESToo. Essa idéia embora pareça promissora requer uma versão da MODESToo mais estável, visto que a mutação das especificações tende a produzir modelos bem variados e, atualmente, a MODESToo funciona apenas para um conjunto restrito de elementos de modelagem.

Apêndice A

Lista de Conferência NEP

Neste apêndice apresentamos os passos constituindo a lista de conferência NEP. Os participantes do experimento tinham que conferir cada um dos itens nessa lista para enviar seu modelo de desenho. Após o recebimento de um modelo, conferíamos cada item para verificar se o desenho estava completo.

Casos de uso:

- Existe o caso de uso criado?
- Ele está associado ao ator apropriado?

Entidades:

- As classes de entidade foram criadas no seu pacote apropriado e com estereótipo certo?
- Todos os atributos possuem comentários que facilitem seu entendimento?
- A entidade foi incluída nos diagramas de herança e visão geral?

Interfaces de Usuário:

- Está no pacote correto?
- Todos os campos e comandos estão estereotipados?
- Todos os campos e comandos possuem comentários e documentação no formato correto?

Diagrama de estados:

- Os estados existentes refletem todas as habilitações da tela?
- Cada estado tem a indicação do que está ativo e inativo?
- Cada transição tem a especificação do evento e/ou da condição associada?

Descrição dos casos de uso:

- O mecanismo de acesso deixa claro como acessar a tela que está associada ao caso de uso?
- O fluxo principal está claro e está associado a funcionalidade mais freqüentemente utilizada no caso de uso?
- Os fluxos alternativos possuem pré-condições para sua ativação corretas?
- Todos os passos no caso de uso são executados por um ator ou pelo sistema?
- Os passos do caso de uso usam apenas estados, campos e comandos do caso de uso (os nomes devem ser compatíveis)?
- Existe alguma referência a alguma coisa de outro caso de uso? Se houver está errado!
- As exceções deixam claro quando podem ser lançadas?
- As mensagens de erro estão todas especificadas e usadas adequadamente no caso de uso?
- As ações realizadas nos passos são inteligíveis? (Não são usados termos vagos como processa, usa, ...)

Apêndice B

Lista de Conferência MEP

Neste apêndice apresentamos os passos constituindo a lista de conferência MEP. Os participantes do experimento tinham que conferir cada um dos itens nessa lista para enviar seu modelo de desenho. Após o recebimento de um modelo, conferíamos cada item para verificar se o desenho estava completo.

Casos de uso:

- Existe o caso de uso criado?
- Ele está associado ao ator apropriado?
- Existe uma colaboração associada ao caso de uso?
- Essa colaboração está em realizações no pacote apropriado?

Entidades:

- As classes de entidade foram criadas no seu pacote apropriado e com estereótipo certo?
- Todos os atributos possuem comentários que facilitem seu entendimento?
- Todos os atributos possuem a especificação das propriedades dos seus atributos? (key-field, size, ...)
- A entidade foi incluída nos diagramas de herança e visão geral?

Interface de Usuário:

- Está no pacote e local correto?
- Todos os campos e comandos estão estereotipados?
- Todos os campos e comandos possuem comentários e documentação no formato correto?

Diagrama de estados

- Os estados existentes refletem todas as habilitações da tela?
- Cada estado tem a indicação do que está ativo e inativo usando o formato correto?
- Cada transição tem a especificação do evento e/ou da condição associada?
- As condições especificadas seguem as regras da linguagem utilizada (especificada abaixo)?
- Os nomes dos comandos utilizados nos eventos estão de acordo com os campos da tela?
- Cada sigla de mensagem corresponde a uma mensagem na classe de mensagens apropriada?

Mensagens do sistema:

- Uma classe de mensagens para o caso de uso foi criada dentro da classe de mensagens?
- Todas as mensagens são públicas, estáticas e finais?
- O conteúdo das mensagens está de acordo com a especificação?

Descrição dos casos de uso (diagramas de seqüência):

- Todos os roteiros do caso de uso foram representados na forma de diagramas de seqüência?
- Todas as mensagens invocadas nos diagramas possuem comentários detalhando o que elas deveriam fazer?
- A primeira mensagem dos diagramas inicia no ator e chega na a tela, além de representar o comando acionado e os parâmetros enviados?
- Todas as possíveis exceções levantadas em um roteiro estão modeladas no diagrama?
- Todas as exceções possuem no seu comentário a indicação da condição que causa sua geração?
- Todos as chamadas a métodos do mecanismo de persistência foram detalhadas no diagrama (criar, alterar, recuperar ou excluir)?
- Todos os acessos aos atributos foram detalhados no diagrama (obterLogin(), atribuirSenha(), ...)?
- As chamadas aos métodos dos objetos participantes da realização do caso de uso parecem ser factíveis para a implementação do caso de uso?

Sintaxe completa da linguagem utilizada para descrição de condições MODEST:

- Entidade.recuperar(campoChave): Recupera um determinado objeto do tipo da entidade.
- Entidade.recuperar(nomeAttr, valAttr): Recupera um determinado objeto com o atributo "nomeAttr"contendo valor "valAttr".
- Entidade.obterPrimeiro() ou Entidade.obterUltimo(): Obtém o primeiro ou último objeto do tipo entidade.
- Entidade.recuperarTodos(): Recupera uma coleção de objetos com todos os elementos do tipo da entidade.
- Objeto.próximo() ou Objeto.anterior(): Retorna o objeto posterior ou anterior ao objeto usado para acionar o método anterior/proximo.
- Objeto.ehPrimeiro() ou Objeto.ehUltimo(): Retorna verdadeiro se o objeto é o primeiro ou último objeto existente.
- ColecaoDeObjetos.tamanho(): Retorna a quantidade de elementos na coleção de objetos.
- ColecaoDeObjetos.obter(num): Retorna o objeto na coleção na posição num.
- ColecaoDeObjetos.obter(attr, val): Retorna o objeto na coleção que possui o valor "val"no atributo "attr".
- Objeto.obterAtributo(): Obtém o valor de um determinado atributo de um objeto.
- Objeto.obterColecao(): Obtém uma coleção associada a um objeto.

Apêndice C

Questionário de Avaliação

Este questionário tem como objetivo entender melhor o estudo experimental realizado, no intuito de facilitar a análise dos dados envolvidos e possivelmente melhorá-lo para estudos futuros. Em hipótese alguma essas respostas serão utilizadas para avaliar o participante ou serão divulgados seus autores.

- Você o utilizaria o MODEST em projetos futuros (Sim/Não/Depende)? Comente se necessário.
- O que você achou o treinamento para execução do estudo (bom/regular/ruim)? Comente se necessário.
- Como você avalia o nível de compreensibilidade do método (bom/regular/ruim)? Comente se necessário.
- Poderia sugerir possíveis alterações que melhorariam o procedimento adotado no estudo experimental envolvendo o MODEST?
- Quais críticas e/ou sugestões você teria sobre os passos existentes no MODEST?

Apêndice D

Lista de Abreviaturas e Acrônimos

- AS: Arquitetura do Sistema
- CRUD: Create, Read, Update, Delete
- GUI: Graphical User Interface
- LOC: Lines Of Code
- MDA: Model Driven Architecture
- MEP: Modelo Específico de Plataforma
- MIP: Modelo Independente de Plataforma
- MODEST: **M**eth**OD** to h**E**lp **S**ystem **T**esting
- OCL: Object Constraint Language
- PRAXIS: **PR**ocesso para **A**plicativos e**X**tensíveis e **I**nterativo**S**
- PU: Processo Unificado
- QFD: Quality Function Deployment
- RNF: Requisito Não Funcional
- SI: Sistema de Informação
- SGDB: Sistema Gerenciador de Banco de Dados
- SST: Sistema Sob Teste
- TBM: Teste Baseado em Modelo
- UML: Unified Modeling Language
- V&V: Verificação e Validação

Referências Bibliográficas

Abdurazik, A. e Offutt, J. (2000). Using UML collaboration diagrams for static checking and test generation. In *Proceedings of the 3rd International Conference on the Unified Modeling Language (UML'00)*, pp. 383–395, York, UK.

Ambler, S. (1998). *Building Object Applications that Work*. Cambridge University Press and Sigs Books.

Andrews, A.; France, R.; Ghosh, S. e Craig, G. (2003). Test adequacy criteria for UML design models. *Jornal of Software Testing, Verification, and Reliability*, 13(2):95–127.

Aranha, E. e Borba, P. (2002). Testes e geração de código de sistemas Web. In *Anais do XVI Simpósio Brasileiro de Engenharia de Software*, pp. 114–128, Gramado, RS.

Barbosa, D.; Andrade, W.; Machado, P. e Figueiredo, J. (2004). Spaces - uma ferramenta para teste funcional de componentes. In *Anais do XVIII Simpósio Brasileiro de Engenharia de Software (SBES2004) - Sessão de Ferramentas*, pp. 55–61, Brasília, DF.

Basili, V.; Selby, R. e Hutchens, D. (1986). Experimentation in software engineering. *IEEE Transactions on Software Engineering*, 12(7):733–743.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

Binder, R. (2000). *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley.

Booch, G.; Rumbaugh, J. e Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley.

Briand, L. e Labiche, Y. (2001). A UML-based approach to system testing. In *Proceedings of the 4th Unified Modeling Language Conference (UML'01)*, pp. 194–208, Toronto, Canada.

Budd, T. (1981). *Mutation Analysis: Ideas, Examples, Problems and Prospects*. North-Holand Publishing Company.

Budd, T. A.; DeMillo, R. A.; Lipton, R. J. e Sayward, F. G. (1980). Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '80)*, pp. 220–233.

Campos, V. F. (1992). *TQC: controle da qualidade total (no estilo japonês).* Escola de Engenharia, UMFG, Fundação Cristiano Ottoni.

Cheng, L. C. (1995). *QFD: planejamento da qualidade.* Escola de Engenharia, UMFG, Fundação Cristiano Ottoni.

Cook, M. (1996a). *Building Enterprise Information Architecture: Reengineering Information Systems.* Prentice Hall.

Cook, M. (1996b). *Building Enterprise Information Architecture: Reengineering Information Systems.* Prentice Hall.

DeMillo, R.; Lipton, R. e Sayward, F. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41.

du Bousquet, L.; Martin, H. e Jézéquel, J.-M. (2001). Conformance testing from UML specifications. experience report. In *Workshop of the pUML-Group held together with the «UML»2001 on Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists*, pp. 43–55. GI.

Farias, C. e Machado, P. (2003). Um método de teste funcional para verificação de componentes. In *Anais do XVII Simpósio Brasileiro de Engenharia de Software (SBES2003)*, pp. 193–208, Manaus, AM.

Filho, W. P. (2003). *Engenharia de Software: Fundamentos, Métodos e Padrões.* LTC, 2a edição.

Ghosh, S.; France, R.; Conrad, B.; Kawane, N.; Andrews, A. e Pilskalns, O. (2003). Test adequacy assessment for UML design model testing. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE 2003)*, pp. 332–343, Denver, Colorado.

Goel, A. L. (1985). Software reliability models: Assumptions, limitations, and applicability. *IEEE Transactions on Software Engineering*, 11(12):1411–1423.

Goodenough, J. e Gerhart, S. (1975). Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173.

Haag, S.; Raja, M. K. e Schkade, L. L. (1996). Quality function deployment usage in software development. *Communications of ACM*, 39(1):41–49.

Hartman, A. e Nagin, K. (2004). The AGEDIS tools for model based testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, Boston, Massachusetts, USA.

Hoffmann, M.; Kuhn, N.; Weber, M. e Bittner, M. (2004). Requirements for requirements management tools. In *Proceedings of the IEEE International Requirements Engineering Conference (RE'04)*, pp. 301–308, Kyoto, Japan.

Hudson, S. (2005). Java Cup User's Manual. Available Online: http://www.cs.princeton.edu/ appel/modern/java/CUP/manual.html, Graphics Visualization and Usability Center, Georgia Institute of Technology. Modified by Frank Flannery, C. Scott Ananian, Dan Wang with advice from Andrew W. Appel, Last updated July 1999 (v0.10j).

IEEE (1990). *IEEE Standard Glossary of Software Engineering Terminology - IEEE Std.610.12-1990*. IEEE Computer Society.

IEEE (1993). *IEEE Standard Classification for Software Anomalies - IEEE Std 1044-1993*. IEEE Computer Society.

IEEE (1998). *IEEE Standard for Software Test Documentation - IEEE Std 829-1998*. IEEE Computer Society.

IEEE (2004). *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society.

Jacobson, I.; Booch, G. e Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley.

Kitchenham, B.; Pickard, L. e Pfleeger, S. L. (1995). Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62.

Kobryn, C. (1999). UML 2001: A standardization odyssey. *Communications of the ACM*, 42(10):29–37.

Kruchten, P. (1995). Architectural blueprints - the "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50.

Kruchten., P. (2003). *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 3rd edição.

Lee, D. e Yannakakis, M. (1996). Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pp. 1090–1126.

Lott, C. e Rombach, D. (1996). Repeatable software engineering experiments for comparing defect-detection techniques. volume 1, pp. 241–277.

Ma, Y.-S.; Offutt, J. e Kwon, Y. R. (2005). MuJava: an automated class mutation system. *Software Testing, Verification, and Reliability*, 15(2):97–133.

McQuillan, J. A. e Power, J. F. (2005). A survey of uml-based coverage criteria for software testing. Technical Report NUIM-CS-TR-2005-08, Department of Computer Science, NUI Maynooth, Co. Kildare, Ireland.

MCT (2004). *Programa Brasileiro da Qualidade e Produtividade de Software*. Ministério da Ciência e Tecnologia, 3a edição.

Mills, H. (1971). *Top Down Programming in Large Systems. Debugging Techniques in Large Systems*. Prentice Hall.

Nebut, C.; Fleurey, F.; Traon, Y. e Jézéquel, J. (2003). Requirements by contracts allow automated system testing. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'03)*, p. 85, Denver, Colorado, USA.

Neto, A. C. D.; Natali, A.; Rocha, A. e Travassos, G. (2006). Caracterização do estado da prática das atividades de teste em um cenário de desenvolvimento de software brasileiro. In *V Simpósio Brasileiro de Qualidade de Software*, Vila Velha, ES.

NIST (2002). Planning Report 02-3, National Institute of Standards and Technology, http://www.nist.gov/director/prog-ofc/report02-3.pdf.

Offutt, A. (1989). The coupling effect: fact or fiction. In *Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification (SIGSOFT '89)*, pp. 131–140.

Offutt, J. e Abdurazik, A. (1999). Generating tests from UML specifications. In *Proceedings of the 2nd Unified Modeling Language Conference (UML'99)*, Fort Collins, Colorado, USA.

OMG (1998). *XML Metadata Interchange*. Object Management Group, http://www.omg.org. last access on november 2004.

OMG (2003a). Mda guide version 1.0.1. Technical Report omg/2003-06-01, Object Management Group, http://www.omg.org/mda/.

OMG (2003b). *UML Testing Profile*. Object Management Group, http://www.omg.org. last access on november 2004.

Papadimitriou, C. (1993). *Computational Complexity*. Addison Wesley.

Pickin, S.; Jard, C.; Traon, Y. L.; Jéron, T.; Jézéquel, J.-M. e Guennec, A. L. (2002). System test synthesis from UML models of distributed software. In *Proceedings of the International Conference Houston on Formal Techniques for Networked and Distributed Systems (FORTE 2002)*, pp. 97–113, Houston, Texas, USA.

Pilskalns, O.; Andrews, A.; Ghosh, S. e France, R. (2003). Rigorous testing by merging structural and behavioral UML representations. In *Proceedings of 6th Unified Modeling Language Conference (UML 2003)*, pp. 234–248, San Francisco, California, USA.

Pressman, R. (2006). *Engenharia de Software*. McGraw-Hill, 6a edição.

Pretschner, A.; Prenninger, W.; Wagner, S.; Kühnel, C.; Baumgartner, M.; Sostawa, B.; Zülch, R. e Stauner, T. (2005). One evaluation of model-based testing and its automation. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pp. 392–401.

Royce, W. W. (1970). Managing the development of large-scale software: Concepts and techniques. In *Technical Papers of Western Electronic Show and Convention (WesCon)*, Los Angeles, USA.

Rumbaugh, J.; Jacobson, I. e Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison-Wesley.

Santos-Neto, P. e Resende, R. (2004). A method for system testing automation. In *First Experimental Software Engineering Latin American Workshop (ESELAW'04), realizado em conjunto com o XVIII Simpósio Brasileiro de Engenharia de Software (SBES 2004)*, Brasilia - DF, Brasil.

Santos-Neto, P.; Resende, R. e Pádua, C. (2005a). Requisitos para automação de testes de sistemas de informação. In *Anais do II Simpósio Brasileiro de Sistemas de Informação (SBSI)*, Florianópolis, SC.

Santos-Neto, P.; Resende, R. e Pádua, C. (2006). An evaluation of a model-based testing method for information systems. In *Submitted for evaluation*.

Santos-Neto, P.; Resende, R. e Pádua, C. (2007). Requirements for information systems model-based testing. In *Proceedings of the ACM Symposium on Applied Computing (SAC2007)*, Seoul, Korea. To appear.

Santos-Neto, P.; Resende, R. e Pádua, C. (2005b). A method for information system testing automation. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, Porto, Portugal.

Santos-Neto, P.; Resende, R. e Pádua, C. (2005c). System testing automation: A developer perspective. In *Proceedings of the 19th Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, Taipei, Taiwan, Republic of China.

Sinha, A. e Smidts, C. (2006). An experimental evaluation of a higher-ordered-typed-functional specification-based test-generation technique. *Empirical Software Engineering*, 11(2):173–202.

Wada, H.; Suzuki, J. e Oba, K. (2005). Modeling turnpike: a model-driven framework for domain-specific software development. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'05)*, pp. 128–129.

Warmer, J. e Kleppe, A. (2003). *The Object Constraint Language*. Addison-Wesley, 2nd edição.

Warmer, N. e Pfleeger, S. (1996). *Software Metrics: A Rigorous & Practical Approach*. International Thomson Computer Press, 2nd edição.

Williams, C. E. (2001). Towards a test-ready meta-model for use cases. In *Workshop of the pUML-Group held together with the «UML»2001 on Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists*, pp. 270–287. GI.

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