Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Informática

Dissertação de Mestrado

Uma Abordagem Dirigida por Modelos para a Geração Automática de Casos de Teste de Integração Usando Padrões de Teste

Camila de Luna Maciel

Campina Grande, Paraíba, Brasil

Agosto de 2010

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

Uma Abordagem Dirigida por Modelos para a Geração Automática de Casos de Teste de Integração Usando Padrões de Teste

Camila de Luna Maciel

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Patrícia Duarte de Lima Machado
Franklin de Souza Ramalho
(Orientadores)

Campina Grande, Paraíba, Brasil
©Camila de Luna Maciel, 06/08/2010

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

M152a Maciel, Camila de Luna.

Uma abordagem dirigida por modelos para a geração automática de casos de teste de integração usando padrões de teste /Camila de Luna Maciel. – Campina Grande, 2010.

163f. : il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande. Centro de Engenharia Elétrica e Informática.

Orientadores: Prof^{a}. PhD Patrícia Duarte de Lima Machado, Prof^{o}. Dr. Franklin de Souza Ramalho.

Referências.

1. Qualidade de Software. 2. Teste Dirigido por Modelo. 3. Padrões de Teste. 4. Teste de Integração. I. Título.

CDU 004.05(043)

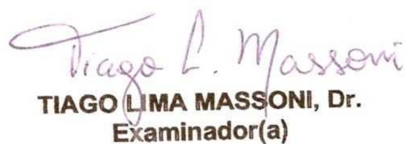
"UMA ABORDAGEM DIRIGIDA POR MODELOS PARA A GERAÇÃO AUTOMÁTICA DE CASOS DE TESTE DE INTEGRAÇÃO USANDO PADRÕES DE TESTE"

CAMILA DE LUNA MACIEL

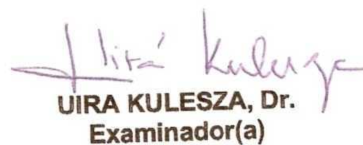
DISSERTAÇÃO APROVADA EM 06.08.2010

PATRICIA DUARTE DE LIMA MACHADO, Ph.D
Orientador(a)

FRANKLIN DE SOUZA RAMALHO, Dr.
Orientador(a)



TIAGO LIMA MASSONI, Dr.
Examinador(a)



UIRA KULESZA, Dr.
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Dentro da Engenharia de Software, novos paradigmas de desenvolvimento vêm surgindo no intuito de oferecer uma maior produtividade sem perda de qualidade aos softwares desenvolvidos. Um desses paradigmas é o MDD (*Model-Driven Development*), cuja principal finalidade é a introdução de modelos rigorosos durante todo o processo de desenvolvimento de software oferecendo, dentre outras vantagens, a geração automática de código a partir dos modelos. Contudo, mesmo em processos de desenvolvimento que seguem este paradigma, a atividade de teste de software ainda é fundamental, principalmente teste de integração, cujo objetivo é verificar que os componentes do software, implementados e testados individualmente, provêem a funcionalidade pretendida quando colocados para interagir uns com os outros. Embora classes individuais possam funcionar corretamente, várias novas faltas podem surgir quando os componentes são integrados. No entanto, em teste de integração, dependendo da complexidade do sistema, o número de casos de teste pode ser muito grande. Nesse contexto, o uso de padrões de teste, ou seja, estratégias que já foram utilizadas e se mostraram efetivas em teste de software, pode guiar a escolha de casos de teste mais efetivos e adequados dentre um número muito grande de possíveis casos de teste. Este trabalho tem como objetivo principal fornecer uma nova abordagem de teste de integração, definida dentro de um processo integrado de desenvolvimento e teste dirigidos por modelos (MDD/MDT - *Model-Driven Testing*), para a geração automática de casos de teste a partir de modelos, utilizando padrões de teste como base para o processo de geração. Para automatizar este processo, foi desenvolvida uma ferramenta baseada em transformações entre modelos segundo práticas da MDA (*Model-Driven Architecture*). Além disso, a abordagem proposta utiliza o perfil de teste da UML para a documentação de todos os artefatos de teste gerados. Adicionalmente, estudos experimentais preliminares foram realizados no intuito de avaliar a abordagem e, consequentemente, a ferramenta de suporte desenvolvida.

Abstract

Within the Software Engineering, new development paradigms are emerging in order to offer greater productivity without sacrificing quality to the developed software. MDD (Model-Driven Development) is one of these paradigms, whose main purpose is to introduce rigorous models along all the software development process offering, among other advantages, automatic code generation from models. However, even in development processes that follow this paradigm, the software testing activity is still essential, especially integration testing, whose purpose is to verify that the software components, implemented and tested separately, provide the desired functionality when placed to interact with each other. While individual components may function correctly, several new faults can arise when the components are integrated. However, in integration testing, depending on the system complexity, the number of test cases can be very large. In this context, the use of test patterns, i. e., strategies that have been used and proved effective in software testing, can guide the user at choosing test cases more effective and appropriate among a very large number of possible cases test. The main objective of this work is to propose a new approach to integration testing, defined within an integrated model driven development and test process (MDD/MDT - Model-Driven Testing) for automatically generating test case from models adopting test patterns as basis for the generation process. To automate this process, we have developed a tool based on model transformations according to MDA (Model-Driven Architecture) practices. Furthermore, the proposal approach uses the UML testing profile to document all generated test artifacts. Additionally, preliminary experimental case studies were performed in order to evaluate the proposed approach and hence the developed tool support.

Agradecimentos

Primeiramente, agradeço a Deus, pela presença constante e pelas bênçãos derramadas sobre a minha vida.

Aos meus pais Aldo e Sonia, por todo amor, dedicação, compreensão, incentivo e, principalmente, por confiarem e acreditarem nesta vitória. À minha irmã Débora, que mesmo distante, sempre torceu por mim. Ao meu esposo Fabiano pela compreensão, carinho, companheirismo, atenção e amor, não medindo esforços para que eu conquistasse mais esta etapa de minha vida. Aos meus amigos que, mesmo diante das minhas ausências, mostraram a essência da verdadeira amizade.

Meus sinceros agradecimentos aos professores Patrícia e Franklin, meus orientadores, pela paciência e compreensão e por todo apoio dedicado.

Agradeço às minhas amigas de mestrado, Andreza, Anne Caroline, Sabrina e Rute pela amizade sincera e pela ajuda e conselhos dados quando mais necessitei. Agradeço também ao colega de mestrado Everton, pela colaboração com o trabalho.

Agradeço também à equipe do GMF pelos incontáveis debates, momentos de descontração e consultorias diversas. Em especial, agradeço a Lilian por ter sido tão prestativa e carinhosa. Agradeço também a Aninha por sempre ser tão atenciosa.

Agradeço a todos aqueles que apesar de já não fazerem parte da minha vida, tiveram importância e influência fundamentais na minha caminhada.

Em especial, agradeço à minha querida avó Severina, que apesar de não estar mais aqui conosco, tenho certeza está feliz por mim.

Por fim, agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), pelo apoio financeiro.

Conteúdo

1 Introdução **1**

1.1 Contextualização e Motivação 1

1.2 Objetivos do Trabalho 4

1.3 Resultados e Relevância do Trabalho 5

1.4 Organização do Documento 6

2 Fundamentação Teórica **8**

2.1 Teste de Software 8

2.1.1 Níveis de Teste de Software 9

2.1.2 Teste de Software Orientado a Objeto 9

2.1.3 *Model-Based Testing* 12

2.1.4 Padrões de Teste 14

2.2 *Model-Driven Architecture* 18

2.2.1 Meta-modelos 19

2.2.2 Transformações 19

2.3 *Model-Driven Testing* 21

2.3.1 Perfil de Teste da UML 2.0 (U2TP) 23

2.4 Considerações Finais 25

3 Abordagem para Geração de Casos de Teste de Integração baseada em Padrões de Teste **27**

3.1 Visão Geral da Abordagem 27

3.2 Adaptação dos Modelos de Desenvolvimento para Teste de Integração . . . 31

3.3 Identificação da Aplicação de Padrões de Teste 35

3.3.1 Padrões de Teste Documentados 37
3.4 Geração de Casos de Teste de Integração 42
3.4.1 Documentando e Gerando os Casos de Teste de Acordo com U2TP 42
3.5 Geração Automática de Casos de Teste de Integração baseada em Padrões de Teste 46
3.6 Considerações Finais 48

4 Suporte Ferramental **50**
4.1 Visão Geral 50
4.2 Módulo de Filtragem dos Modelos 54
4.3 Módulo de Identificação da Aplicação de Padrões de Teste 57
4.4 Módulo de Geração de Casos de Teste 60
4.5 Considerações Finais 64

5 Avaliação Experimental **67**
5.1 Método de Avaliação Experimental Utilizado 67
5.2 Estudo Experimental para Avaliação da Abordagem 68
5.2.1 Definição 68
5.2.2 Planejamento 73
5.2.3 Execução e Resultados 76
5.2.4 Interpretação 81
5.3 Procedimento do Estudo Experimental para Avaliação da Ferramenta 87
5.3.1 Definição 87
5.3.2 Planejamento 89
5.3.3 Execução e Resultados 90
5.3.4 Interpretação 92
5.4 Considerações Finais 94

6 Trabalhos Relacionados **96**
6.1 Teste de Integração Baseado em Modelos 96
6.2 Teste Dirigido por Modelo: Teoria e Prática 98
6.3 Teste Dirigido por Modelo e Padrões de Teste 99

6.4 Considerações Finais 100

7 Conclusões **102**

7.1 Trabalhos Futuros 104

A Catálogo de Padrões de Teste **111**

A.1 *Round-trip Scenario Test* 111

A.2 *Abstract Class Test* 111

A.3 *Error Simulator* 114

A.4 *Test Case With Time Specification* 118

B Especificação do *ATM System* **122**

B.1 Descrição do Domínio 122

B.2 Modelo de Casos de Uso: Diagrama de Casos de Uso 122

B.3 Modelo Estrutural: Diagrama de Classes 123

B.4 Modelos Comportamentais: Diagramas de Sequência 125

B.5 Cenários de Integração 130

C Especificação de Outros Sistemas **133**

C.1 *Object Manager System* 133

C.2 *Simplified ATM System* 134

C.3 *File Manager* 137

D Resultados do Primeiro Estudo de Caso **138**

D.1 Resultados do Primeiro Participante 138

D.2 Resultados do Segundo Participante 144

D.3 Resultados do Terceiro Participante 148

E Resultados do Segundo Estudo de Caso **152**

E.1 Precisão 152

E.2 Escalabilidade 161

F Questionários de Avaliação **162**

Lista de Símbolos

ATL - *ATLAS Transformation Language*

CIM - *Computational Independent Model*

CITM - *Computational Independent Testing Model*

GPL - *General Public License*

GQM - *Goal/Question/Metric*

IOP - *Integration Order Profile*

MBT - *Model-Based Testing*

MDA - *Model-Driven Architecture*

MDD - *Model-Driven Development*

MDT - *Model-Driven Testing*

MOF - *Meta Object Facility*

OCL - *Object Constraint Language*

OMG - *Object Management Group*

OO - *Orientado a Objeto*

PIM - *Platform Independent Model*

PITM - *Platform Independent Testing Model*

PSM - *Platform Specific Model*

PSTM - *Platform Specific Testing Model*

QVT - *Query/View/Transformation*

SUT - *System Under Test*

UML - *Unified Modeling Language*

U2TP - *UML 2 Testing Profile*

XMI - *XML Metadata Interchang*

Lista de Figuras

2.1 Diagrama de classes extraído de [29] para a análise de dependências. 12
2.2 Atividades de teste baseado em modelo. 13
2.3 Transformações entre modelos em MDA. 19
2.4 Modelos de desenvolvimento (MDD) vs modelos de teste (MDT) [3]. . . . 22

3.1 Processo integrado de desenvolvimento e testes dirigidos por modelos usando padrões de teste. 28
3.2 Abordagem dirigida por modelos para a geração de casos de teste de integração. 29
3.3 Etapas da abordagem a serem realizadas pelo testador. 30
3.4 *Integration Order Profile* (IOP). 33
3.5 Exemplo de aplicação do perfil IOP. 34
3.6 Extensão do meta-modelo de UML para padrões de teste. 36
3.7 (a) Diagrama de sequência *Remove Object*, e (b) seu grafo de fluxo correspondente. 40
3.8 Caminhos para o diagrama de sequência *Remove Object*, sem loops ou condições, gerados a partir do grafo de fluxo. 41
3.9 Casos de teste gerados para o diagrama de sequência *Remove Object*, documentados de acordo com U2TP e aplicando o padrão. 41
3.10 Exemplos de mensagens de retorno do veredito. 44
3.11 Diagrama de sequência mostrando um caso de teste de acordo com U2TP. . 45
3.12 Diagrama de classes mostrando a arquitetura de teste de acordo com U2TP. 45
3.13 Arquitetura da abordagem de acordo com os princípios de MDA. 46

4.1 Visão geral do funcionamento da ferramenta. 51
4.2 Arquitetura geral da ferramenta. 53

4.3 Abordagem MDA do módulo de filtragem dos modelos. 55
4.4 Meta-modelo *ApplicableTestPatterns* para os padrões de teste identificados. 58
4.5 Abordagem MDA do módulo de identificação da aplicação de padrões de teste. 58
4.6 Abordagem MDA do módulo de geração de casos de teste. 60
4.7 Configurações ATL para a execução terceiro módulo para o padrão *Round-trip Scenario Test*. 65

5.1 O paradigma *Goal/Question/Metric* - GQM (adaptado de [9]). 68
5.2 Atividades realizadas pelos participantes durante o estudo experimental. . . 78
5.3 Comparativo do esforço com relação ao tempo. 82
5.4 Comparativo do esforço com relação ao número de casos de teste. 82
5.5 Comparativo do esforço para a geração de um caso de teste. 83
5.6 Comparativo da porcentagem de casos de teste válidos gerados. 84
5.7 Comparativo da capacidade de detecção de defeitos dos casos de teste. . . . 85
5.8 Comparativo da escalabilidade da ferramenta para os quatro sistemas utilizados. 93

A.1 (a) Fragmento do diagrama de classes mostrando o «*TestComponent*» desenvolvido. (b) Um caso de teste exemplo para o diagrama de sequência *Remove Object*, documentado de acordo com U2TP e aplicando o padrão *Abstract Class Test*. 114
A.2 Fragmento do diagrama de classes do *Object Manager System*. 116
A.3 Diagrama de sequência *openFile* do *Object Manager System* mostrando o tratamento de exceções. 117
A.4 Diagrama de sequência mostrando os casos de teste para o tratamento de exceções, documentados de acordo com U2TP e aplicando o padrão. 118
A.5 Diagrama de sequência para o caso de uso *Deposit Funds*. 120
A.6 Diagrama de sequência mostrando o caso de teste, documentado de acordo com U2TP e aplicando o padrão *Test Case With Time Specification*. 121

B.1 Casos de uso do *ATM System*. 123
B.2 Diagrama de classes do *ATM System*. 124
B.3 Diagrama de sequência representando o caso de uso *Query Account*. 125

B.4 Diagrama de sequência representando o caso de uso *Validate PIN*. 126
B.5 Diagrama de sequência representando o caso de uso *Withdraw Funds*. . . . 127
B.6 Diagrama de sequência representando o caso de uso *Transfer Funds*. 128
B.7 Diagrama de sequência representando o caso de uso *Deposit Funds*. 129
B.8 Diagrama de sequência representando o comportamento do ATM *System* com relação às exceções. 130

C.1 Diagrama de classes do *Object Manager System*. 133
C.2 Diagrama de sequência do *Object Manager System* (*Add Object*). 134
C.3 Diagrama de sequência do *Object Manager System* (*Remove Object*). 134
C.4 Diagrama de classes do *Simplified ATM System*. 135
C.5 Diagrama de sequência do *Simplified ATM System* (*Authenticate Client*). . . 135
C.6 Diagrama de sequência do *Simplified ATM System* (*Withdraw Funds*). . . . 136
C.7 Diagrama de sequência do *Simplified ATM System*(*Deposit Funds*). 136
C.8 Diagrama de classes do *File Manager*. 137
C.9 Diagrama de sequência do *File Manager* (*Open File*). 137

D.1 Caso de teste inválido gerado pelo participante 1. 140
D.2 Caso de teste inválido gerado pelo participante 2. 146

Lista de Tabelas

2.1 Exemplos de padrões de teste. 17

2.2 Comparativo entre formatos de definição existentes para padrões de teste. . 18

5.1 Tipos de defeitos de integração que poderiam ser detectados pelos padrões de teste. 72

5.2 Perfil dos participantes do estudo de caso. 77

5.3 Informações sobre os artefatos do sistema *ATM System*. 79

5.4 Dados coletados por participante para o primeiro estudo de caso. 80

5.5 Tipos de defeitos de integração detectados pelos casos de teste agrupados por participante e pelo tipo de defeito. 81

5.6 Resumo dos dados coletados por participante para obtenção dos valores das métricas. 81

5.7 Informações sobre os artefatos dos quatro sistemas utilizados. 91

5.8 Dados coletados com a execução da ferramenta para os quatro sistemas utilizados. 92

6.1 Resumo dos trabalhos relacionados. 101

D.1 Dados coletados na execução do estudo (participante 1). 139

D.2 Casos de teste válidos por cenário de acordo com os padrões de teste (participante 1). 143

D.3 Tipos de defeitos de integração que poderiam ser detectados pelos casos de teste (participante 1). 144

D.4 Dados coletados na execução do estudo (participante 2). 145

D.5 Casos de teste válidos por cenário de acordo com os padrões de teste (participante 2). 147

D.6 Tipos de defeitos de integração que poderiam ser detectados pelos casos de teste (participante 2). 148

D.7 Dados coletados na execução do estudo (participante 3). 149

D.8 Casos de teste válidos por cenário de acordo com os padrões de teste (participante 3). 150

D.9 Tipos de defeitos de integração que poderiam ser detectados pelos casos de teste (participante 3). 151

E.1 Resumo dos dados coletados de todos os sistemas para o cálculo da precisão. 161

E.2 Dados coletados de todos os sistemas para o cálculo da escalabilidade. . . . 161

Lista de Códigos Fonte

2.1 Exemplo de regras ATL. 21

3.1 Elementos que identificam a aplicação do padrão *Round-trip Scenario Test* de acordo com o meta-modelo *TestPatternMetamodel*. 39

4.1 Regra ATL do módulo de filtragem dos modelos. 57

4.2 Regras ATL do módulo de identificação da aplicação de padrões de teste. . 59

4.3 Regras ATL do módulo de geração de casos de teste. 63

A.1 Elementos que identificam a aplicação do padrão *Abstract Class Test* de acordo com o meta-modelo *TestPatternMetamodel*. 112

A.2 Elementos que identificam a aplicação do padrão *Error Simulator* de acordo com o meta-modelo *TestPatternMetamodel*. 115

A.3 Elementos que identificam a aplicação do padrão *Test Case With Time Specification* de acordo com o meta-modelo *TestPatternMetamodel*. 119

E.1 XMI para o cenário inicial do *Object Manager System*. 153

E.2 XMI para o primeiro cenário de mutação do *Object Manager System*. . . . 154

E.3 XMI para o segundo cenário de mutação do *Object Manager System*. 154

E.4 XMI para o cenário inicial do *Simplified ATM System*. 155

E.5 XMI para o primeiro cenário de mutação do sistema *Simplified ATM System*. 156

E.6 XMI para o segundo cenário de mutação do sistema *Simplified ATM System*. 156

E.7 XMI para o cenário inicial do sistema *File Manager System*. 157

E.8 XMI para o primeiro cenário de mutação do sistema *File Manager System*. . 158

E.9 XMI para o cenário inicial do *ATM System*. 158

E.10 XMI para o primeiro cenário de mutação do *ATM System*. 160

E.11 XMI para o segundo cenário de mutação do *ATM System*. 160

Capítulo 1

Introdução

1.1 Contextualização e Motivação

O processo de desenvolvimento de software tem passado por intensas transformações nos últimos anos. Novos paradigmas de desenvolvimento vêm surgindo no intuito de aumentar a produtividade sem perda da qualidade dos softwares construídos. Uma desses paradigmas é o MDD (*Model-Driven Development*), cuja finalidade é a introdução de modelos rigorosos durante todo o processo de desenvolvimento oferecendo, dentre outras vantagens, a geração automática de código a partir dos modelos [6]. Em outras palavras, o foco de MDD é concentrar os esforços na construção de modelos, que representam diferentes visões do sistema, e, potencialmente, em diferentes níveis de abstração, e não mais em linguagens de programação. Dessa forma, os modelos deixam de ser somente documentação para tornarem-se elementos ativos no processo de desenvolvimento. Esta característica elimina muitos dos problemas dos processos de desenvolvimento tradicionais, tais como: inconsistência entre documentação e sistema final; pouca ou quase que total falta de portabilidade; e baixa confiabilidade [3].

Uma das realizações de MDD é uma iniciativa da OMG (*Object Management Group* [36]) conhecida por MDA (*Model-Driven Architecture*) [27]. Em MDA, modelos em vários níveis de abstração são os artefatos centrais do desenvolvimento do software. Sua ideia-chave é mudar a ênfase na implementação do software, para o foco na modelagem, meta-modelagem e transformação entre modelos. Além disso, MDA visa a separação entre a especificação de funcionalidades do sistema e a sua realização usando uma especificação

mais detalhada e específica de plataforma. A portabilidade é atestada em MDA, pois de posse dos modelos iniciais de planejamento, que são independentes de plataforma, e das regras de transformação adequadas para cada etapa, automaticamente tem-se a transição do sistema independente de plataforma para uma plataforma específica e desta para o código final do software. Da mesma forma, o caminho inverso pode ser seguido [3].

No entanto, um sistema não pode ser considerado confiável sem que seja realizado um processo de verificação e validação (V&V) adequado. Mesmo sistemas desenvolvidos seguindo o paradigma MDD, onde o código do software pode ser gerado automaticamente a partir de modelos, não estão livres de defeitos. Nesse contexto, teste de software, uma das técnicas de V&V mais usadas, torna-se ainda essencial. Principalmente teste de integração, cujo objetivo é verificar que os componentes do software, implementados e testados individualmente, provêem a funcionalidade pretendida quando colocados para interagir uns com os outros. Embora componentes individuais possam funcionar corretamente, várias novas faltas podem surgir quando eles são integrados [11]. Ainda, no contexto de desenvolvimento de software, pode haver a necessidade de fornecer rapidamente um conjunto limitado de funcionalidades do software aos clientes. Então, mesmo que o software não esteja completo, testes de integração devem ser realizados para certificar-se de que o software (ou parte dele) é de qualidade, ou seja, atende à sua especificação e às exigências do cliente [43].

Em teste de integração, dependendo da complexidade do sistema, o número de combinações a serem testadas pode ser muito grande (infinitas combinações). Esta característica mostra a necessidade do uso de estratégias para minimizar os esforços gastos nesse tipo de teste. Já há algum tempo, metodologias atestaram que casos de teste de qualidade poderiam ser derivados a partir modelos derivados de especificações, surgindo a abordagem MBT (*Model-Based Testing*) [17]. No entanto, com relação à geração de casos de teste de integração, as abordagens MBT existentes costumam gerar casos de teste que, muitas vezes, são de difícil implementação por serem abstratos demais ou dependentes de linguagens específicas de plataforma ou por não conterem informação suficiente para derivar casos de teste concretos (por exemplo, o método SCOTEM proposto por Ali *et al.* [2]). Além disso, em estratégias de MBT, o enfoque na geração de casos de teste deixa ainda informal o planejamento e a construção da infra-estrutura de teste.

Outro conceito que visa facilitar a atividade de testes, e consequentemente do teste de integração, são os padrões de teste [11, 28, 46]. Um padrão de teste pode ser definido como uma solução, aplicável em várias situações, para um problema recorrente no contexto de teste de software [11, 28]. O princípio é reutilizar essências de soluções e experiências em teste, bem como idéias/estratégias que já foram aplicadas e se mostraram efetivas em contextos similares. Neste sentido, o uso de padrões de teste pode guiar a escolha de casos de teste mais efetivos e adequados dentre um número muito grande de possíveis casos de teste de integração. Além disso, uma característica muito importante dos padrões de teste é que muitos deles podem ser usados para a geração de casos de teste a partir de especificações. Logo, o uso de padrões, além de simplificar o desenvolvimento dos testes de integração possibilitando a geração de bons casos de teste, viabiliza o aumento do nível de automação.

A aplicação de padrões de teste para a geração automática de casos de teste a partir de modelos no contexto de teste de integração é bastante promissora. No entanto, devido aos problemas relacionados às estratégias MBT, se faz necessário o uso de uma estratégia onde casos de teste possam ser gerados de forma mais eficaz. Para isso, pode ser utilizado MDT (*Model-Driven Testing*) [6, 24]. O interesse principal de MDT é combinar MBT estratégias da MDA para geração completa dos artefatos de teste necessários para a especificação, execução e avaliação de casos de teste de forma automática através de regras de transformação entre modelos. Com MDT, casos de teste podem ser definidos de forma abstrata e refinados junto com a aplicação, ou seja, novos casos de teste podem ser incluídos ou detalhados para considerar decisões de projeto, e casos de teste podem ser gerados para testar aplicações em diferentes plataformas [3]. O principal diferencial de MDT com relação à MBT é que muitas abordagens baseadas em MBT não consideram a distinção entre modelos independentes e específicos de plataforma, os modelos são geralmente projetados de acordo com uma plataforma específica ou são genéricos demais e representam apenas informações independentes de plataforma.

Há uma carência de abordagens que fazem uso de padrões de teste para a geração automática de casos de teste, principalmente no contexto de teste de integração. Além disso, a utilização deles no processo de teste ainda é bastante manual, mesmo considerando o nível de automação que provêem. Isso acontece porque, da forma com que os padrões estão definidos atualmente, é necessário que a equipe de teste examine a implementação do

software para decidir se um dado padrão de teste pode ser utilizado ou não para a construção de casos de teste. No entanto, uma vez que padrões de teste são estratégias que podem minimizar os custos inerentes ao teste de integração e tendo em vista as vantagens que MDT proporciona com relação à geração automática de casos de teste independentes de plataforma a partir de modelos, com a combinação destes conceitos casos de teste de integração importantes e mais adequados para um contexto específico podem ser desenvolvidos mais efetivamente.

1.2 Objetivos do Trabalho

O trabalho aqui proposto tem como objetivo fornecer um nova abordagem de teste, a qual é definida dentro de um processo integrado de desenvolvimento e teste dirigidos por modelos (MDD/MDT). A ideia principal da abordagem é a utilização de padrões de teste para a geração automática de casos de teste a partir de modelos, no contexto de teste de integração. O intuito é fazer uso apenas de artefatos produzidos pelo processo de desenvolvimento utilizado, especificamente, dos modelos de desenvolvimento descritos por diagramas UML (*Unified Modeling Language*) [40]. Isto permite que a abordagem tenha um impacto reduzido no total de esforço a ser empreendido no processo como um todo, uma vez que não é preciso a construção de modelos específicos para testes pelos testadores.

Para dar suporte automático à abordagem, foi desenvolvida uma ferramenta utilizando transformações entre modelos segundo práticas de MDA. Além disso, para a documentação dos casos de teste a fim de que os mesmos sejam independentes de plataforma, ou seja, independentes de linguagem de programação, é utilizado o perfil de teste U2TP (*UML 2 Testing Profile*) [37], o qual cobre amplamente os conceitos abordados em um projeto de teste.

Nesse contexto, os objetivos específicos deste trabalho são:

- O desenvolvimento de uma abordagem para a geração de casos de teste de integração independentes de plataforma a partir de modelos de desenvolvimento abstratos descritos por diagramas UML, com base em padrões de teste existentes, dentro de um processo integrado de MDD/MDT;

- A definição de uma forma para adaptar os modelos para o teste de integração;
- A definição de uma linguagem de especificação para formalizar os padrões de teste de acordo com um formato específico;
- A disponibilização de uma ferramenta de suporte automático à abordagem usando práticas de MDT com MDA;
- A realização de estudos experimentais como forma de avaliação da abordagem proposta e da ferramenta de suporte desenvolvida.

1.3 Resultados e Relevância do Trabalho

A principal contribuição deste trabalho é a disponibilização de uma abordagem inserida dentro de um processo integrado de MDD/MDT, para a geração automática de casos de teste de integração usando padrões de teste. A utilização de padrões de teste nesse sentido é importante pois visa a geração de casos de teste mais eficientes e adequados ao contexto dos sistemas que serão testados, uma vez que são reutilizadas estratégias que se mostraram efetivas em contextos similares. Tal característica permite que defeitos conhecidos e relevantes possam eventualmente ser capturados, assegurando uma melhor qualidade do sistema desenvolvido. Além disso, como nem sempre a geração e execução de todos os conjuntos de testes de integração de um sistema é viável, ou até mesmo seja de interesse gerar casos de teste apenas para comportamentos específicos, é fundamental aplicar estratégias para a seleção dos casos de teste mais importantes.

Um ponto relevante deste trabalho é a proposta de uma forma automática de apoio para a atividade realizada na prática pela equipe de testadores, e, consequentemente, a não adição de custos extras associados à tarefa de testes, uma vez que a abordagem não se utiliza de artefatos além da própria especificação funcional do sistema a ser testado. Ainda, no contexto de teste de integração, dependendo da complexidade do sistema, o número dos casos de teste pode ser muito grande, o que inviabiliza qualquer técnica de teste de integração manual. Estas características mostram a grande necessidade de automação nesse aspecto. Além disso, a abordagem permite uma maior integração entre as fases de desenvolvimento e testes, o que

possibilita a verificação automática de que o sistema satisfaz os requisitos especificados até mesmo antes da implementação estar disponível.

Outro ponto importante deste trabalho é a utilização de práticas de MDT com MDA, que permitem que casos de teste sejam definidos de forma abstrata e independente de linguagem de programação, podendo ser utilizados para testar os sistemas em diferentes plataformas. Além disso, os casos de teste podem ser definidos e refinados junto com a aplicação, ou seja, novos casos de teste podem ser incluídos ou detalhados para considerar decisões de projeto, automaticamente.

Uma breve descrição das contribuições relativas a cada resultado obtido é dada a seguir:

- **Um catálogo de padrões de teste utilizáveis em nível de modelos.** A disponibilização de um catálogo de padrões de teste aplicáveis para qualquer sistema orientado a objeto em nível de modelos;

- **A disponibilização de uma ferramenta para a geração automática dos casos de teste de integração.** Uma ferramenta desenvolvida usando práticas de MDA para a viabilizar a utilização da abordagem proposta, a qual permite a geração dos casos de teste de integração de forma automática;

- **Um procedimento detalhado para a avaliação de abordagens de teste baseado em modelos.** O desenvolvimento de um procedimento genérico para a avaliação de abordagens automáticas e não-automáticas de teste baseado em modelos, que pode servir como guia para outros trabalhos.

1.4 Organização do Documento

As demais partes deste documento estão estruturadas da seguinte forma:

- **Capítulo 2: Fundamentação Teórica.** Traz os principais conceitos abordados pelo trabalho desenvolvido. Inicialmente são abordados conceitos de teste de software, ressaltando: níveis e tipos, teste de software orientado a objeto (OO), teste de integração, MBT e padrões de teste. Em seguida conceitos relacionados a MDA e MDT são apresentados;

- **Capítulo 3: Abordagem para Geração de Casos de Teste de Integração baseada em Padrões de Teste.** Apresenta o principal produto deste trabalho que consiste em uma abordagem dirigida por modelos para a geração de casos de teste de integração usando padrões de teste. Uma visão geral da abordagem é apresentada e, em seguida, as diversas etapas que a compõem são apresentadas em detalhes. Além disso, é introduzida uma forma de automação da abordagem utilizando estratégias de MDT com MDA;

- **Capítulo 4: Suporte Ferramental.** Apresenta a arquitetura e as funcionalidades da ferramenta que foi desenvolvida para dar suporte automático à abordagem proposta;

- **Capítulo 5: Avaliação Experimental.** Apresentada o procedimento utilizado para uma avaliação experimental, através de estudos de caso, da abordagem proposta e da ferramenta de suporte desenvolvida neste trabalho;

- **Capítulo 6: Trabalhos Relacionados.** Apresenta alguns trabalhos relacionados ao atual estado da arte no contexto de geração automática de casos de teste a partir de modelos. São apresentadas abordagens e/ou estratégias de teste de integração usando técnicas de MBT e MDT (teoria e na prática) e abordagens que integram MDT com padrões de teste. Além disso, uma comparação desses trabalhos com o aqui proposto é realizada;

- **Capítulo 7: Conclusões.** Apresenta as conclusões deste trabalho através da apresentação dos resultados alcançados de uma descrição das perspectivas de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este Capítulo tem como objetivo fornecer o embasamento teórico sobre conceitos necessários para uma melhor compreensão deste trabalho. Inicialmente, na Seção 2.1 é introduzido o teste de software, ressaltando seus níveis, o teste de software OO e teste de integração, MBT e padrões de teste. Em seguida, na Seção 2.2, conceitos relacionados à MDA no geral são introduzidos. Por fim, definições e conceitos sobre MDT, enfatizando U2TP, são apresentados na Seção 2.3.

2.1 Teste de Software

Teste de software, uma das técnicas de V&V[1] mais usadas, consiste em um processo dinâmico que envolve a execução do software com dados de teste para verificar se o seu comportamento está de acordo com o especificado. O objetivo é mostrar que o software é bom o suficiente para o uso operacional encontrando o maior número de defeitos possível.

Essencialmente, o processo de teste consiste em definir um conjunto de casos de teste para o software a ser testado. Um caso de teste é a especificação de **entradas** do teste, compostas por condições iniciais para que ele possa ser executado e dados utilizados na sua execução, e **resultados esperados**, compostos por condições que devem ser satisfeitas após sua execução, e as saídas que o software deve exibir para as estradas associadas [48]. Uma

[1] Verificação e Validação. Verificação refere-se às atividades que garantem que o software implementa corretamente uma funcionalidade específica e Validação refere-se às atividades que garantem que o software construído corresponde às exigências do cliente.

vez definido o caso de teste, o sistema sob teste deve ser executado de acordo com as entradas (pré-condições e dados necessários), e os resultados obtidos devem ser comparados com os resultados esperados para determinar se o teste passou ou não. Casos de teste definidos para um dado sistema em um determinado contexto é denominado de conjunto de teste (*test suite*).

2.1.1 Níveis de Teste de Software

É possível identificar diferentes níveis de teste com relação à granularidade do código do sistema a ser testado. Cada nível de teste apresenta problemas específicos que requerem técnicas e estratégias específicas. Os diferentes níveis são: teste de unidade, teste de integração e teste de sistema [11]. O **teste de unidade** corresponde à verificação da menor unidade de software individualmente e visa identificar erros de lógica e implementação. O objetivo do **teste de integração** é descobrir defeitos associados à comunicação entre as unidades do software. O nível de **teste de sistema** compreende em testar todo o sistema, levando em consideração o software em seu provável ambiente de produção. Neste nível, são testadas características que estão presentes no sistema como um todo para avaliar o seu comportamento com relação a sua especificação.

Além desses três níveis de teste, alguns autores (como [42,43,48]) consideram um quarto nível chamado de teste de aceitação. O **teste de aceitação**, também conhecido como teste de validação, é um conjunto de atividades (ou testes) realizadas pelo usuário-final com a finalidade de verificar se o software se comporta como o esperado. Enquanto que os níveis de teste apresentados anteriormente estão preocupados com a verificação do software contra a sua especificação formal (ou semi-formal), o teste de aceitação compara o comportamento do software com os requisitos informais do usuário-final.

2.1.2 Teste de Software Orientado a Objeto

A introdução do paradigma orientado a objeto (OO), no contexto de desenvolvimento de software, provocou mudanças significativas na forma como os softwares são criados, testados e mantidos. Essas mudanças foram motivadas pela nova perspectiva adotada pelo paradigma (ênfase nos objetos), em oposição ao paradigma procedural, cujo foco é na funcionalidade e no fluxo de informação dos sistemas [42]. Embora seja possível a aplicação de técnicas

de teste tradicional para softwares OO, características desse paradigma, tais como herança, polimorfismo e abstração [11, 42], introduzem novos interesses de teste, que podem não ser tratados de forma adequada pelas abordagens tradicionais e requerem a definição de novas técnicas e estratégias de teste. Por exemplo, o uso de polimorfismo, ou seja, a possibilidade de uma ou mais entidades se referir, dinamicamente, a objetos de diversos tipos, implica no número de casos a serem testados. Testar todas as combinações pode ser inviável e a criação de conjuntos de teste para cobrir todas as possíveis combinações podem não ser alcançados com as abordagens tradicionais.

Uma das maiores vantagens do paradigma OO é a possibilidade de construir classes independentemente gerenciáveis, que podem ser combinadas para obter um sistema por inteiro. Um software OO bem projetado é composto por classes pequenas que interagem entre si, e a complexidade desse software, diferentemente do software tradicional, é transferida das classes para as interações entre elas. Essa diferença mostra que o teste de unidade é mais simples, e que o teste de integração é mais extensivo.

Teste de Integração de Software Orientado a Objeto

Teste de integração de software OO tenta validar que as classes do software, implementadas e testadas individualmente, provêem a funcionalidade pretendida quando colocadas para interagir umas com as outras. Embora classes individuais possam funcionar corretamente, várias novas faltas podem surgir quando as classes são integradas em conjunto, tais como faltas de interfaces, funções conflitantes e funções ausentes [11].

Uma das questões fundamentais do teste de integração é a escolha da ordem de integração, ou seja, a ordem na qual as classes são integradas. Três estratégias bastante conhecidas, onde a ordem de integração é o interesse principal, são: *bottom-up*, *top-down* e *big-bang* [42]. A estratégia *bottom-up* consiste em testar as classes do software em um baixo nível e continuar, incrementalmente, os testes seguindo uma hierarquia até que a classe de mais alto nível seja testada. Já na estratégia *top-down*, ocorre exatamente o contrário, o sistema é testado do alto nível para um baixo nível. Em ambos os casos, emuladores são necessários para suprir classes ainda inexistentes, ou seja, aquelas que ainda não foram integradas, mas que contêm relações de dependência com o que se deseja testar. Três tipos de emuladores são [6]:

- *Dummies*: são as implementações mais rudimentares de funcionalidades ausentes. Muitas vezes eles só provêem definições de interfaces, tais como funções "cabeçalho", que são apenas necessárias para compilar o sub-sistema sem erros;

- *Stubs*: fornecem funcionalidades suficientes para o teste. Um *stub* pode ser implementado para um determinado conjunto de testes e reagir a esses testes de forma razoável;

- *Mock Objects*: tem mais inteligência do que *dummies* e *stubs*. Eles podem ser completos simuladores da funcionalidades ausentes.

As estratégias *bottom-up* e *top-down* são aplicáveis a software OO desde que as dependências entre as classes sejam corretamente identificadas. Dependendo de uma série de fatores como, o tipo de software e a disponibilidade de recursos, é possível utilizar uma ou outra estratégia, ou ainda, como é muito utilizado, uma mistura das duas estratégias. A integração incremental mostra-se como a técnica mais apropriada, pois é mais fácil isolar a causa das falhas quando se testa por partes menores.

A estratégia *big-bang* consiste em testar todas as classes isoladamente e depois integrá-las de uma só vez. Consequentemente, são precisos emuladores para testar as classes individualmente. Esta estratégia pode ser aplicada para software OO, porém é recomendada apenas para softwares pouco complexos, ou seja, composto de poucas classes. No geral, essa estratégia pode ser considerada inviável e desastrosa devido a complexidade das interações entre as classes, pois a integração tem o objetivo de minimizar número de interfaces a serem exercitas de uma só vez. Além disso, é difícil encontrar a causa de falhas em um software por inteiro, principalmente se as classes nunca foram exercitadas em subconjuntos menores.

É essencial em teste de integração para software OO identificar uma ordem de prioridade das classes para os testes, ou seja, estabelecer critérios de precedência entre as classes para verificar o funcionamento conjunto delas. A forma ou ordem de integração das classes pode influenciar os tipos de teste a serem feitos, e, consequentemente, o custo e a eficácia do teste. Uma estratégia ruim pode levar à construção de muitos emuladores ou à combinação de classes não testadas previamente, dificultando o processo de localização de defeitos. Por exemplo, considere o diagrama de classes mostrado na Figura 2.1. Utilizando a estratégia para a identificação da ordem de integração utilizando heurísticas proposta por Lima e

Travassos [29], implicará na necessidade de implementação de dois emuladores específicos: um emulador de C para testar A e um outro emulador de C para testar H, determinando um esforço de teste igual a dois. Utilizando a estratégia proposta por Briand *et. al* [12] seriam precisos quatro emuladores para o mesmo exemplo (emuladores de C, F, H e B, para testar A, E, C e H, respectivamente). É possível observar que houve uma redução substancial do esforço de teste.

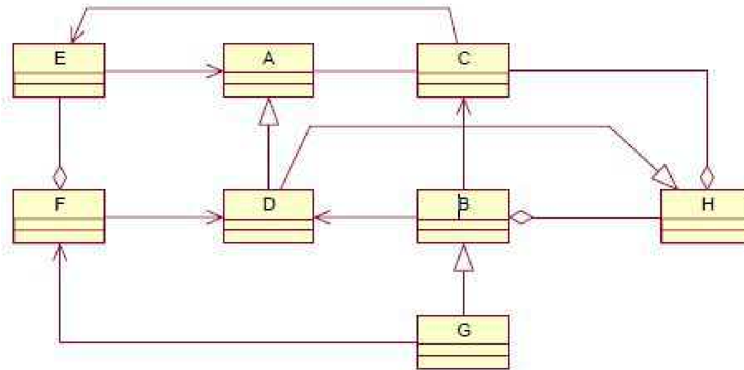


Figura 2.1: Diagrama de classes extraído de [29] para a análise de dependências.

2.1.3 *Model-Based Testing*

Model-Based Testing (MBT) [17] é uma abordagem de teste funcional[2] para a geração de casos de teste a partir de modelos essencialmente relacionados à especificação dos requisitos funcionais da aplicação. Para isto, a especificação da aplicação precisa estar formalmente ou semi-formalmente descrita por meio de um modelo de modo a caracterizar com exatidão o seu comportamento [10, 15]. As principais atividades de MBT, mostradas também na Figura 2.2, são [17, 44]:

- **Construir o Modelo**: Construir o modelo (formal ou semi-formal) a partir dos requisitos do sistema;
- **Gerar Entradas**: Gerar as entradas do teste, as quais farão parte dos casos de teste e servirão para exercitar o sistema que está sendo testado, a partir do modelo;

[2]Teste funcional, ou caixa-preta, é uma abordagem onde casos de teste são gerados baseados na especificação abstrata do sistema, ignorando a implementação do sistema.

- **Gerar Resultados**: Gerar as saídas esperadas do teste, as quais indicam o comportamento esperado do sistema, a partir do modelo;

- **Executar os Testes**: Exercitar o sistema que está sendo testado, denominado sistema sob teste, com as estradas geradas, produzindo novas saídas;

- **Comparar Saídas**: Comparar as saídas do sistema, após a execução dos testes, com as saídas esperadas geradas;

- **Analisar/Tomar Decisões**: Analisar os resultados, e tomar decisões (modificar o modelo? Gerar mais testes? Parar o processo?) com base nos objetivos de teste e critérios de parada.

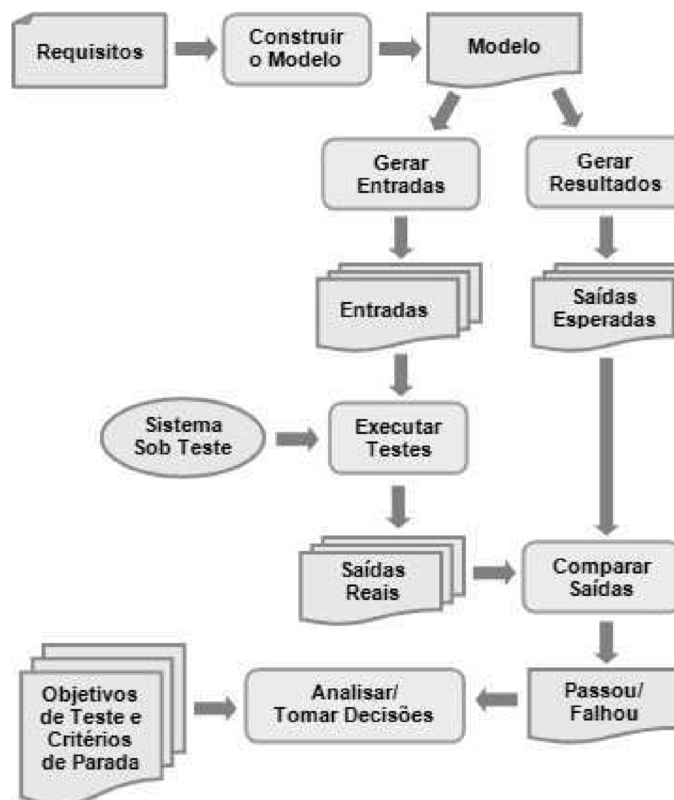


Figura 2.2: Atividades de teste baseado em modelo.

O processo de MBT se inicia assim que os requisitos do sistema são definidos. A partir dos requisitos, o modelo que representa o comportamento esperado do sistema é construído. A partir deste modelo, entradas e resultados esperados são gerados. Estas

informações, dentre outras, fazem parte dos casos de teste. Com as entradas, o sistema sob teste é executado e o seu comportamento é observado. Em seguida, as saídas reais obtidas são comparadas com as saídas esperadas para verificar se o teste passou ou não e, consequentemente, avaliar a presença de defeitos no sistema. Por fim, uma análise é feita acerca dos resultados obtidos e decisões são tomadas com base nos objetivos de teste e critérios de parada do processo.

Em um processo que utiliza a abordagem MBT, é preciso definir, dentre outras coisas, o formalismo (ou notação) utilizado para o modelo. Existe uma grande variedade de formalismos para a representação dos modelos que descrevem os diferentes aspectos estruturais e comportamentais de um sistema. Dentre estes formalismos estão [17]: máquinas de estados finitas e suas variações, a linguagem UML e cadeias de *Markov*. A UML é, por sua vez, o formalismo mais utilizado em processos MBT.

Uma das grandes vantagens de MBT é que a atividade de teste pode ser realizada em paralelo à implementação, reduzindo os custos inerentes associados a esta atividade. Além disso, com MBT há a possibilidade de automação da geração dos casos de teste e o compartilhamento de um modelo único entre as várias equipes e pessoas envolvidas no processo de desenvolvimento [17]. Uma desvantagem de utilizar MBT é a requisição de conhecimento da notação do modelo. Os testadores devem ser familiarizados com a notação que será utilizada, o que culmina na requisição de tempo e investimento em treinamentos por parte deles [17]. Outra desvantagem é que, dependendo do nível de abstração, o modelo pode não conter informação suficiente para derivar casos de teste concretos.

2.1.4 Padrões de Teste

Padrões foram introduzidos pela primeira vez no contexto da construção civil, para arquitetura e construção de casas e vilas com base na combinação de soluções conhecidas e comprovadas. Segundo C. Alexander [1]:

> *"Um padrão é uma entidade que descreve um problema que ocorre repetidamente em um ambiente e então descreve a essência da solução para este problema, de tal forma que você use esta solução milhões de vezes, sem nunca utilizá-la do mesmo modo".*

Basicamente, um padrão pode ser visto como uma regra de três partes que expressa uma relação entre um dado *contexto*, um *problema* e uma *solução*, onde [1, 18]:

- O *contexto* descreve a situação na qual o problema resolvido pelo padrão acontece. É importante destacar que o contexto em si não é problema, mas pode levar ao problema;
- O *problema* descreve um adversidade que ocorre repetidamente em um dado contexto. As chamadas *forças*, geralmente listadas na descrição do problema, referem-se a quaisquer outros objetivos, requisitos ou limitações que precisam ser consideradas ao resolver o problema;
- A *solução* provê uma forma abstrata capaz de balancear as forças associadas ao problema.

Embora estas definições não tenham sido desenvolvidas tendo em mente o desenvolvimento de software, verificou-se que a noção de padrões é importante para a Engenharia de Software. Diante dessa perspectiva, diversos tipos de padrões foram desenvolvidos e têm sido cada vez mais utilizados pelos desenvolvedores de software na busca por maior eficácia e eficiência em seus processos de desenvolvimento. Dentre estes padrões destacam-se: padrões de análise [19], padrões arquiteturais [13], padrões de projeto [20], padrões de processo [4] e padrões de testes [11].

Padrões de teste surgem como interessantes aliados na busca por softwares de qualidade. Utilizando a mesma noção de padrões definido por C. Alexander em [1], um padrão de teste pode ser definido como uma *solução*, aplicável em várias situações, para um *problema* conhecido de no *contexto* de teste de software. Mais especificamente, é um tipo de estratégia de teste que se aplica a um determinado contexto com o intuito de revelar falhas conhecidas e recorrentes [11, 28].

Padrões de teste, por serem modelos abstratos, são compatíveis com muitos processos de teste. Basicamente, usar padrões de teste em um processo de desenvolvimento e teste de software consiste dos seguintes passos [11]:

1. No início do processo de desenvolvimento, selecionar os padrões de teste correspondentes ao escopo e à estrutura do sistema em desenvolvimento;

2. Desenvolver um modelo de teste para a implementação desejada baseado nos padrões de teste selecionados;

3. Gerar o conjunto de teste aplicando o modelo de teste criado;

4. Desenvolver a implementação dos testes;

5. Executar e avaliar os testes. Se a cobertura desejada não for atingida, revisar o conjunto de teste o quanto for necessário.

Padrões de teste têm sido usados em diferentes formas e níveis [11, 18, 34], por exemplo:

- Padrões de projeto de teste: usados para desenvolver testes para funcionalidades específicas. Entre eles estão: (i) padrões de casos de teste, focados na geração de casos de teste para problemas específicos; e (ii) padrões de estratégia/implementação de teste, focados em como organizar/implementar a lógica do teste;

- Padrões de automação de teste: estratégias usadas para a automação e execução de testes.

Embora o conceito de padrões de teste seja um assunto relativamente recente na comunidade de Engenharia de Software, existem alguns propostos na literatura que já vêm sido utilizados e referenciados na área de teste de software. Por exemplo, com o objetivo de testar sistemas OO, Binder [11] propõe uma coleção de padrões de projeto de teste para vários artefatos, incluindo classes, métodos e cenários de integração. Além disso, padrões de automação de teste são apresentados. Adicionalmente, padrões de implementação de teste são propostos Lange [28] e Meszaros [34] para testes de componentes e integração e para testes de unidade, respectivamente. A Tabela 2.1 mostra dois exemplos de padrões de teste existentes na literatura.

Padrões de teste têm potencial para o aumento da eficácia e eficiência das atividades de teste. Com a identificação de contextos recorrentes, estes padrões podem ser utilizados para identificar defeitos também recorrentes, ou seja, defeitos comumente encontradas nestes contextos [11]. Em outras palavras, um padrão de teste proverá uma forma de se obter um determinado conjunto de casos de teste conhecidamente eficazes em revelar certos tipos de defeitos. Entretanto, o fato de ser um assunto recente na comunidade, padrões de teste possuem alguns problemas que, de certa forma, estão relacionados:

Tabela 2.1: Exemplos de padrões de teste.

Nome	**Contexto/Problema**	**Solução**
Round-trip Scenario Test [10]	Cenários *round-trip* são caminhos completos nos diagramas de sequência. Estes geralmente incluem esses cenários, porém nem sempre explicitamente.	Extrair um grafo de fluxo a partir de diagramas de sequência e desenvolver casos de teste que provêem um mínimo de cobertura de caminhos.
Test Case With Time Specification [27]	Em algumas situações, casos de teste baseados na especificação funcional não são suficientes, como testes de funções que devem ser executadas dentro de um limite de tempo. Este padrão provê uma forma para a definição de casos de teste para especificações de tempo.	Adicionar um tempo para o caso de teste. O tempo de sistema atual deve ser armazenado quando o caso de teste inicia. Depois que o caso de teste tiver terminado, o tempo de sistema atual será recuperado novamente. A diferença é então colocada na asserção para a verificação.

- **Carência de formatos de definição mais consolidados e amplamente utilizados:** Não há um formato comum para a documentação dos padrões de teste. Alguns autores propõem formatos baseados no proposto por Gamma et al. [20] para padrões de projeto, como é o caso de Binder [11] e Thomas et al. [49], mas com elementos adicionais. A Tabela 2.2 mostra um comparativo entre os elementos de três formatos de definição existentes na literatura, onde é possível perceber os elementos em comum, alguns deles diferindo apenas no nome, mas que, conceitualmente, consistem nos mesmos elementos;

- **Falta de classificação e catalogação, no intuito de facilitar sua busca e utilização:** Não existe um catálogo único (ou repositório) para a documentação dos padrões, que facilite sua busca e utilização;

- **Poucos trabalhos falando sobre suas utilizações:** Não existem muitos trabalhos que

mostram exemplos de utilização e aplicação dos padrões de teste existentes na prática;

- **A aplicação/utilização ainda bastante manual:** Há uma carência de estratégias para a aplicação e/ou utilização dos padrões automaticamente.

Tabela 2.2: Comparativo entre formatos de definição existentes para padrões de teste.

Binder	Lange	Jon Thomas
• Nome	• Nome	• Nome
• Intenção		• Intenção
• Contexto	• Motivação • Forças	• Motivação • Aplicabilidade
• Modelo de Faltas		• Modelo de Faltas
• Estratégia	• Solução	• Estrutura • Participantes • Colaborações
	• Questões de Implementação	• Implementação
		• Código Exemplo
• Consequências	• Consequências	• Consequências
• Usos Conhecidos	• Usos Conhecidos	• Usos Conhecidos
• Padrões Relacionados	• Padrões Relacionados	• Padrões Relacionados

2.2 *Model-Driven Architecture*

Model-Driven Architecture (MDA) [27] é uma iniciativa da OMG (*Object Management Group*) onde modelos em vários níveis de abstração são os artefatos centrais do desenvolvimento de software. MDA é uma realização de MDD (*Model-Driven Development*) [6] e sua ideia-chave é mudar a ênfase na implementação do software, para o foco na modelagem, meta-modelagem e transformação entre modelos. Além disso, MDA visa a separação entre a especificação de funcionalidades do sistema e a sua realização usando uma especificação mais detalhada e específica de plataforma.

Em MDA, conforme mostrado na Figura 2.3, o sistema é primeiramente analisado e especificado como um CIM (*Computational Independent Model*), também conhecido como modelo de domínio, o qual foca nos requisitos e no domínio do sistema e desconsidera quaisquer detalhes computacionais e de implementação. O CIM é transformado em um PIM (*Platform Independent Model*) que contém informação computacional para a aplicação, mas não contém informação específica sobre a tecnologia da plataforma que será usada para implementá-lo. Contudo, vale ressaltar que essa transformação não é total, porque CIMs,

em geral, são modelos de alto nível muito abstratos e imprecisos. Um PIM é transformado em um PSM (*Platform Specific Model*), o qual inclui descrições detalhadas e elementos específicos da plataforma. Como existem várias tecnologias para a implementação de sistemas de software, pode-se ter vários PSMs associados a um único PIM. Finalmente, a partir de PSMs, código-fonte executável pode ser gerado [27]. Além disso, os modelos podem ser transformados em outras direções, como PIM-PIM, representando refinamentos em um mesmo nível de abstração, PSM-PIM, ou PIM-código diretamente.

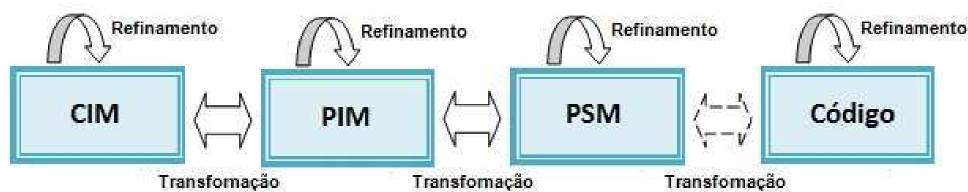


Figura 2.3: Transformações entre modelos em MDA.

Os principais componentes de uma abordagem MDA são: meta-modelos (estruturas que descrevem como os modelos devem ser formados), modelos (instâncias dos meta-modelos) e transformações (regras que definem como modelos de entrada devem ser transformados em modelos de saída, com base em seus meta-modelos).

2.2.1 Meta-modelos

Meta-modelos são estruturas que descrevem sintaticamente como os modelos devem ser formados. Atualmente, os meta-modelos podem ser definidos através de duas principais linguagens para meta-modelagem: MOF (*Meta Object Facility*) [39] e Ecore [16]. Um exemplo de meta-modelo bem definido e conhecido é a especificação da UML.

2.2.2 Transformações

MDA possui um alto potencial de automação alcançado através de transformações entre modelos, especialmente entre os modelos PIM, PSM e código. As transformações definem como os modelos de entrada podem ser mapeados para os modelos de saída, baseados nos seus meta-modelos.

Atlas Transformation Language (ATL) [5] é uma linguagem para a especificação de transformações entre modelos. Esta linguagem é baseada em OCL (*Object Constraint Language*) [38], ou seja, a navegação nos modelos é realizada através de expressões OCL. Um programa de transformações em ATL consiste em um módulo composto de regras que definem como elementos em um modelo são mapeados e navegados para criar e iniciar elementos de outro modelo. Um módulo ATL é composto basicamente por quatro componentes: (i) *header section*, que define unidades relevantes para a transformação, por exemplo, os nomes dos módulos de entrada e de saída; (ii) *import section*, que permite fazer importações de bibliotecas ATL; (iii) *helpers*, que pode ser visto como funções auxiliares; e (iv) *rules*, que são as regras de transformação.

Existem dois tipos de regras de transformação em ATL, são as chamadas *matched rules* (regras declarativas) e *called rules* (regras imperativas). As *matched rules* são regras baseadas em casamento de padrões que especificam para quais tipos de modelos de entrada os modelos de saída devem ser gerados, e como eles devem ser gerados. Numa *matched rule* vários elementos destino (determinado por uma cláusula *to*) podem ser gerados a partir de um elemento fonte (determinado por uma cláusula *from*). Entretanto, um elemento do meta-modelo fonte não pode aparecer em mais de uma *matched rule*. Um bloco imperativo pode ser adicionado em uma regra declarativa, representado pela cláusula *do*. As *called rules* são regras que devem ser explicitamente chamadas a partir dos blocos imperativos, podendo estar contidas em *matched rules* ou em outras *called rules*. Essas regras são semelhantes aos *helpers*. Regras desse tipo não precisam de um padrão de modelos fonte e podem opcionalmente possuir um padrão de modelos de saída.

O Código 2.1 apresenta um modelo da linguagem ATL cujo objetivo é a transformação de elementos entre *Livro* e *Publicacao*. As linhas 3-10 apresentam uma *matched rule* denominada por *Autor*. As cláusulas *from* e *to* representam o padrão do modelo fonte a ser casado e o padrão do modelo destino a ser gerado, respectivamente. Na linha 4, MMAutor representa o meta-modelo de entrada e *Autor* representa o elemento do meta-modelo de entrada. Na linha 5, MMPessoa representa o meta-modelo de saída e *Pessoa* representa o elemento do meta-modelo de saída. Desta forma, ao transformar uma elemento *Autor* em um elemento *Pessoa*, eles terão o mesmo nome e sobrenome. Já as linhas 11-19 apresentam uma *called rule* denominada por *NovaPessoa*, que tem função semelhante da *matched rule*

Autor. Nesta regra, *nome* a e *sobrenomea* são os parâmetros de entrada.

Código Fonte 2.1: Exemplo de regras ATL.

```
module Livro2Publicacao;
create OUT : Publicacao from IN : Livro;
rule Autor {
 from a : MMAutor!Autor
 to
  p : MMPessoa!Pessoa (
   nome <- a.nome,
   sobrenome <- a.sobrenome
  )
}
rule novaPessoa (nomea: String, sobrenomea: String) {
 to
   p : MMPessoa!Pessoa (
    nome <- nomea
   )
 do {
   p.sobrenome <- sobrenomea
 }
}
```

Apesar de não ser o padrão proposto pela OMG para linguagem de transformação (a OMG propõe QVT - *Queries/Views/Transformations* [41]), a ATL é bastante utilizada na prática pelos desenvolvedores. Além disso, como ATL foi construída na mesma época da definição do padrão QVT, ela possui várias similaridades em relação ao padrão, o que a torna simples e fácil de usar.

2.3 *Model-Driven Testing*

Com o intuito de se beneficiar da separação de modelos na geração e execução de testes, a estratégia *Model-Driven Testing* (MDT) [3, 24] se propõe a ser o refinamento das principais atividades de MBT adotando princípios de MDA. Nesse contexto, as atividades diretamente afetadas por MDT são: (i) a geração de casos de teste a partir de modelos de acordo com um dado critério de cobertura; (ii) a geração de oráculos e dados de teste para determinar os resultados esperados de um teste; e (iii) a execução de testes em ambientes de teste. As duas primeiras são independentes de plataforma, enquanto que a última requer o uso de modelos específicos capazes de gerar os ambientes de teste e de mapear os casos de teste nas

plataformas específicas correspondentes. Além disso, um dos objetivos de MDT é a geração automática desses artefatos de teste (casos de teste, dados de teste, oráculos, etc.) a partir de modelos de desenvolvimento através de regras de transformação.

A mesma abstração introduzida por MDA em termos de modelagem (independente e específica) de plataforma e geração de código pode ser aplicada para projetar modelos de teste com MDT. Nesse contexto, Alves *et al.* [3] introduziram os seguintes novos conceitos: (i) CITM (*Computational Independent Test Model*), responsáveis por capturar os objetivos de teste; (ii) PITM (*Platform Independent Test Model*), os quais refletem a arquitetura de teste e casos de teste abstratos; e (iii) PSTM (*Platform Specific Test Model*), que representam os artefatos de testes específicos para uma plataforma. Conforme mostrado na Figura 2.4, em MDT, modelos de teste podem ser gerados tanto a partir de modelos de teste mais abstratos, como diretamente de modelos de projeto do sistema, permitindo uma melhor integração entre as atividades de desenvolvimento e teste.

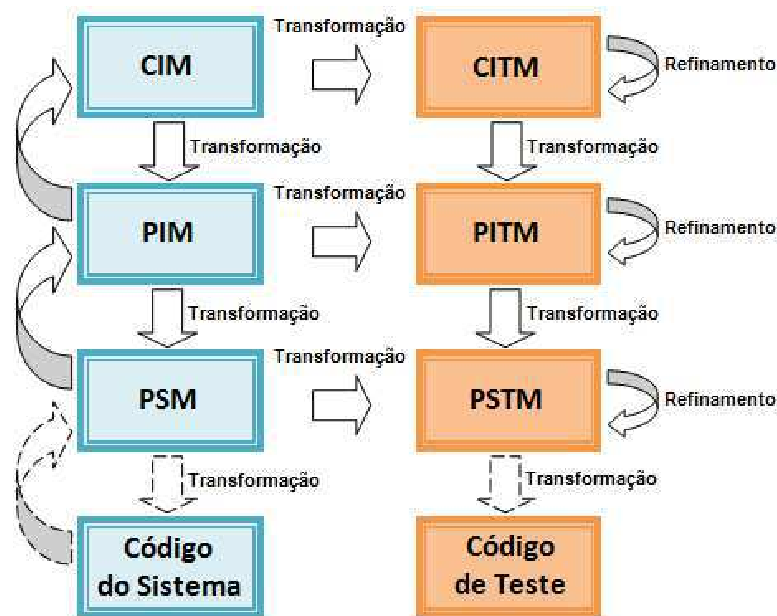


Figura 2.4: Modelos de desenvolvimento (MDD) vs modelos de teste (MDT) [3].

A principal diferença entre MDT e MBT é que muitas das abordagens de MBT não consideram essa distinção entre modelos independentes e específicos de plataforma [24]. Modelos em MBT são geralmente projetados de acordo com uma plataforma específica ou são genéricos demais e representam apenas as informações independentes de

plataforma. Além disso, em MBT, os casos de teste são derivados a partir dos modelos de desenvolvimento fracamente conectados, onde estes são normalmente incompletos no que diz respeito às informações necessárias para os testes (restrições, casos alternativos, etc.). Com a utilização das práticas de MDA, onde modelos são o centro do desenvolvimento, tais informações poderão ser naturalmente incorporadas [3].

2.3.1 Perfil de Teste da UML 2.0 (U2TP)

Um conceito emergente em teste, proposto pela OMG, é o perfil de teste da UML 2.0 (U2TP - *UML 2.0 Testing Profile*) [37]. Este perfil define uma linguagem para projetar, visualizar, especificar, construir e documentar artefatos de sistemas de teste em diferentes domínios. Dessa forma, U2TP atua como elo entre projetistas e testadores provendo formas de usar UML tanto para modelagem de sistemas como para a especificação de testes [14]. O perfil U2TP é um dos formalismos mais utilizados em abordagens MDT por fornecer uma forma de documentar artefatos de teste em um linguagem independente de plataforma e por cobrir amplamente os conceitos abordados em um projeto de teste.

U2TP é definido com base no meta-modelo da UML 2.0 e foi especificado com foco em reuso. Onde possível, conceitos da UML são diretamente usados. A extensão desses conceitos, bem como a adição de novos, são feitas somente quando necessários. Sendo assim, U2TP é organizado em quatro grupos de conceitos: *arquitetura de teste*, *comportamento de teste*, *dados de teste* e *conceitos de tempo*.

Arquitetura de Teste

Este grupo define os principais elementos e seus relacionamentos envolvidos em um teste. Basicamente, consiste em conceitos estritamente relacionados à estrutura e configuração de teste. Alguns desses elementos são [37]:

SUT (*System Under Test*). Consiste no sistema sob teste, ou seja, corresponde ao que será testado. Um SUT pode ter diferentes níveis de abstração: um sistema completo, um subsistema, um pacote, um único componente ou até mesmo uma única classe, e é usado dentro do contexto de teste (descrito mais adiante). A notação para o SUT é nomear o que se deseja testar com o estereótipo «SUT»;

Test Component. Consiste numa classe (ou componente) do sistema em teste, que pode ser simulada por um emulador, ou qualquer outro componente que se comunica com o SUT para realizar o comportamento de teste. A notação para este elemento é uma classe com o estereótipo «*TestComponent*» aplicado;

Test Context. Consiste numa classe (ou componente) que representa o agrupamento de vários casos de teste, ou seja, representa o conjunto de teste. A notação para este elemento é uma classe (ou componente) com o estereótipo «*TestContext*» aplicado;

Arbiter. Consiste numa interface pré-definida fornecida juntamente com o perfil U2TP. O propósito de uma implementação *Arbiter* é determinar o *Verdict* (apresentando mais adiante), ou seja, o resultado final para um caso de teste;

Scheduler. Também consiste numa interface pré-definida fornecida com o perfil U2TP. O propósito de uma implementação *Scheduler* é controlar a execução de diferentes *Test Components*. O *Scheduler* mantém a informação sobre qual *Test Component* está executando e colabora com o *Arbiter*, para informar o *Verdict* final para o caso de teste. Além disso, mantém o controle sob a criação e destruição dos *Test Components* bem como informações sobre quais deles estão participando de cada caso de teste.

Comportamento de Teste

Este grupo define os conceitos necessários para representar todos os elementos que fazem parte dos aspectos dinâmicos dos procedimentos de teste. Alguns desses conceitos são [37]:

Test Case. Consiste na especificação de uma situação de teste do sistema, incluindo o que testar, entradas e resultados esperados de acordo com seu contexto de teste. Um *Test Case* pode ter seu próprio esquema de arbitragem (que verifica se ele passou ou não) usando um *Arbiter* e deve sempre retornar um veredito (*Verdict*). *Test Cases* são geralmente especificados com diagramas comportamentais da UML. A notação para este elemento é uma operação do contexto de teste e um diagrama comportamental com o estereótipo «*TestCase*» aplicado;

Validation Action. As *Validation Actions* são ações usadas para definir vereditos em comportamentos de teste. A notação para este elemento é uma ação com o estereótipo «*ValidationAction*» aplicado;

Verdict. Um *Verdict* é um tipo de dados *enumeration* pré-definido que representa o

veredito final do caso de teste. Possíveis vereditos são: (i) *pass*, indica que o caso de teste é completo e que o SUT se comportou como esperado; (ii) *fail*, descreve que o propósito do caso de teste foi violado, ou seja, o resultado esperado foi diferente do resultado real; (iii) *inconclusive*, usado quando nenhum valor *pass* ou *fail* pode ser fornecido; e (iv) *error*, usado para indicar erros (exceções) dentro do sistema sob teste.

Dados de Teste

Este grupo contém os conceitos necessários para descrever os elementos de dados de teste usados para exercitar os casos de teste. Esses elementos contêm os tipos de dados que são usados para execução de um ou mais casos de teste e são utilizados no estímulo ao SUT e para a coordenação entre os *Test Components*. São exemplos desses conceitos [37]: *wildcards*, *data partitions*, *data pools*, *data selectors* e *coding rules*.

Conceitos de Tempo

Este último grupo define um conjunto de conceitos para especificar restrições e/ou de observações de tempo. O objetivo principal desses elementos é restringir e controlar o comportamento de teste e garantir o término dos casos de teste. São eles [37]:

***Timer*.** É uma interface pré-definida fornecida no perfil U2TP e consiste num mecanismo de temporização. Um *Timer* provê operações como *start*, *stop* e *read*;

***Time Zone*.** Consiste num mecanismo de agrupamento de *Test Components*. Cada *Test Component* pertence no máximo a um *Time Zone* e possui a mesma percepção de tempo dos outros envolvidos, ou seja, os *Test Components* são sincronizados.

2.4 Considerações Finais

O intuito deste Capítulo foi apresentar o embasamento teórico necessário para o entendimento do trabalho. Todos os conceitos aqui apresentados compreendem conceitos e/ou técnicas utilizados para o desenvolvimento do trabalho e para a elaboração dos próximos capítulos. Primeiramente, foram mostrados os principais conceitos relacionados ao teste de software importantes para o trabalho, dentre eles, conceitos acerca dos níveis e tipos de teste, do teste de software OO dando importância ao teste de integração, de MBT e de padrões de

teste. Em seguida, conceitos relacionados à arquitetura MDA e ao MDT, enfatizando o perfil U2TP, foram contextualizados.

Capítulo 3

Abordagem para Geração de Casos de Teste de Integração baseada em Padrões de Teste

Este Capítulo apresenta uma abordagem dirigida por modelos para a geração de casos de teste de integração para sistemas OO. Esta abordagem é uma extensão da técnica apresentada em [31] e sua ideia é, em linhas gerais, fazer uso de padrões de teste de uma maneira integrada com práticas de MDT para a geração de casos de teste de integração a partir de modelos de desenvolvimento. O Capítulo está organizado da seguinte forma. Inicialmente, na Seção 3.1, uma visão geral da abordagem é apresentada. Nas Seções seguintes, as etapas desenvolvidas para o processo de geração são apresentadas com suas justificativas. Na Seção 3.5, uma forma de automação dessa abordagem no intuito da geração automática dos casos de teste é apresentada. Por fim, na Seção 3.6, são apresentadas considerações finais, com uma breve discussão incluindo restrições e extensões da aplicação da abordagem proposta.

3.1 Visão Geral da Abordagem

A abordagem é aplicada dentro de um processo integrado de desenvolvimento e teste dirigidos por modelos realizado baseado em um modelo iterativo e incremental, conforme mostrado na Figura 3.1.

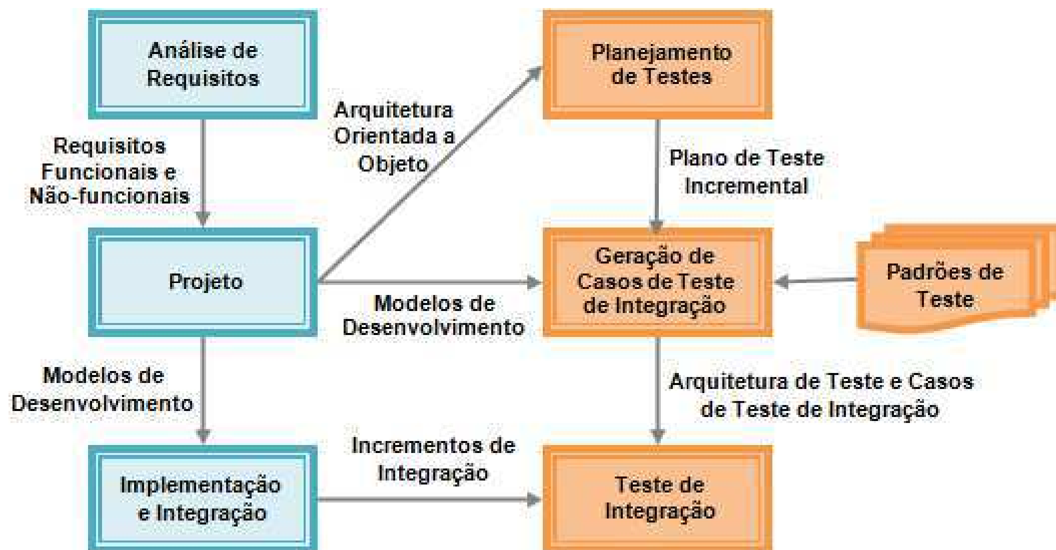


Figura 3.1: Processo integrado de desenvolvimento e testes dirigidos por modelos usando padrões de teste.

Nesse processo, os mesmos modelos de desenvolvimento usados como base para a implementação do software OO, os quais são construídos a partir dos requisitos funcionais e não-funcionais extraídos na fase de análise de requisitos, são utilizados para testes. A ideia é integrar esses modelos com modelos de teste abstratos documentados por padrões de teste para a geração de artefatos de teste (arquitetura e casos de teste) com base em um plano de teste incremental. Este plano, documentado na fase de planejamento de testes, descreve a estratégia global de integração, ou seja, quais componentes serão testados, como devem ser agrupados e a ordem de integração entre eles em cada interação e de acordo com a arquitetura OO definida no projeto do software. Uma vez gerados os casos de teste, os testes de integração poderão ser executados baseado nos incrementos de integração gerados durante a fase de implementação e integração.

Contudo, apesar de ser considerado um processo completo, o foco da abordagem é apenas na geração dos artefatos de teste. Nesse contexto, só são importantes para a abordagem os seguintes artefatos, conforme mostrado na Figura 3.2: (i) os modelos de desenvolvimento, resultantes do projeto do software; (ii) o plano de teste incremental, resultante do planejamento de testes e necessário para o testes de integração; e (iii) os padrões de teste. Como resultado da aplicação da abordagem, tem-se, além dos casos de teste de integração, outros artefatos necessários para a realização deles (arquitetura de teste).

Figura 3.2: Abordagem dirigida por modelos para a geração de casos de teste de integração.

Atualmente, a notação UML 2.0 (ou superior) é um formalismo padrão amplamente utilizado pela academia e pela indústria na especificação de modelos para softwares OO. Essa notação é usada para descrever tanto características estruturais quanto comportamentais do software, a partir de diagramas UML. Nesse contexto, considerou-se que os modelos de desenvolvimento utilizados na abordagem são diagramas UML 2.0, especificamente diagramas de classes e de sequência, os quais devem ser consistentes e refletir a estrutura e o comportamento do sistema adequadamente. Além disso, o interesse é aplicar a abordagem em um processo integrado de desenvolvimento e teste dirigidos por modelos (MDD/MDT) [3]. Logo, os casos de teste gerados devem ser independentes de plataforma, ou seja, independentes de linguagem de programação. Para alcançar esta independência, os casos de teste, bem como todos os outros artefatos gerados, são documentados de acordo com o perfil de teste da UML (U2TP), o qual provê formas de descrever modelos de teste independentes de plataforma e que cobre amplamente os conceitos abordados em um projeto de teste.

A abordagem está dividida em três etapas principais a serem realizadas pelo testador a cada etapa do modelo incremental, conforme mostrado na Figura 3.3:

1. **Adaptação dos modelos de desenvolvimento para teste de integração.** Uma vez que o foco da abordagem é em teste de integração, esta atividade tem o objetivo de adaptar os modelos de desenvolvimento, ou seja, os diagramas de classe e sequência UML 2.0, a esse contexto de teste de acordo com o plano de teste incremental. Para isso, é proposto um perfil UML, cujo objetivo é identificar com anotações (estereótipos) as classes que farão parte do teste de integração. O intuito é simplificar os modelos

de desenvolvimento para que apenas as classes importantes para o teste de integração sejam consideradas;

2. **Identificação da aplicação de padrões de teste.** Esta segunda atividade tem a finalidade de identificar, dentre os padrões de teste considerados, aqueles que são aplicáveis ao contexto do sistema que se deseja testar. Como a abordagem utiliza apenas os modelos de desenvolvimento como entrada do processo, é proposto um formato de definição que inclui um meta-modelo que permite identificar a aplicabilidade do padrão em nível de modelos. O intuito desta etapa é mostrar para o testador quais são os padrões que podem ser utilizados para a geração dos casos de teste para que ele possa decidir quais utilizar na atividade seguinte;

3. **Geração de casos de teste.** Esta terceira e última atividade tem a finalidade de gerar os casos de teste de integração independentes de plataforma com base nos padrões de teste identificados. Também nesta atividade, uma arquitetura de teste importante para auxiliar na execução dos casos de teste é gerada. Todos os artefatos de teste gerados serão também diagramas de classe e sequência UML 2.0, porém documentados de acordo com o perfil U2TP.

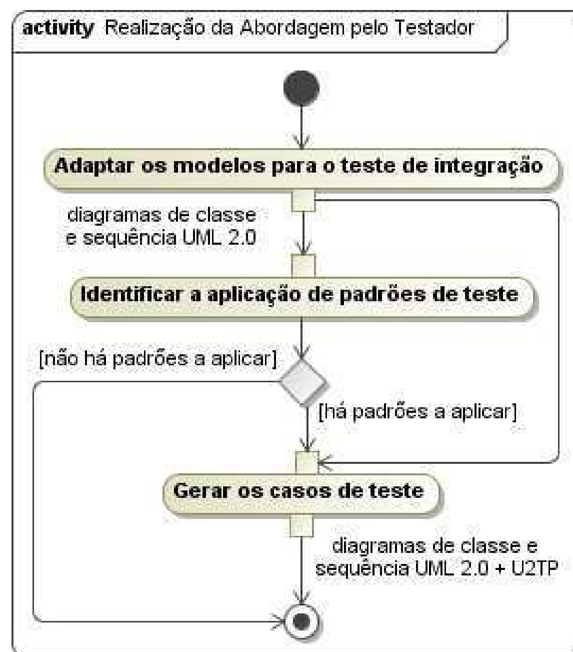


Figura 3.3: Etapas da abordagem a serem realizadas pelo testador.

Partindo do pressuposto que os modelos de desenvolvimento tenham sido desenvolvidos na etapa de projeto, as etapas da abordagem devem ser realizadas repetidamente até que todas as classes do sistema tenham sido devidamente testadas e integradas. A seguir, são apresentadas em detalhes cada uma dessas etapas do processo de aplicação da abordagem bem como todos os artefatos e ferramentas utilizados durante o processo.

3.2 Adaptação dos Modelos de Desenvolvimento para Teste de Integração

Devido aos problemas associados com a integração *big-bang* (Seção 2.1.2), as classes precisam ser integradas uma de cada vez, ou em alguns casos especiais, em pequenos *clusters* [11]. Nesse contexto, em teste de integração para software OO é essencial identificar uma ordem de prioridade das classes para os testes, ou seja, estabelecer critérios de precedência entre as classes para verificar o funcionamento conjunto das mesmas. Em outras palavras, uma das questões fundamentais do teste de integração é a escolha da ordem de integração, ou seja, a ordem na qual as classes são integradas.

Independente da estratégia de teste de integração escolhida, as classes são integradas com outras que, por definição, já se encontram desenvolvidas e testadas individualmente, ou seja, o código da classe foi examinado e revisado, e defeitos, porventura existentes, já foram removidos. No entanto, em algumas situações, as classes podem estar em fases distintas do ciclo de desenvolvimento do software. Enquanto algumas podem estar prontas para executar testes de integração, outras podem ainda estar em fase de codificação ou em fase de testes individuais (teste de unidade). Assim, quando existirem classes necessárias à integração que ainda não estão disponíveis, é preciso que sejam criados emuladores de testes, ou seja, pedaços de software construídos para simular partes do software que ainda não estão disponíveis, para dar prosseguimento aos testes.

A forma ou ordem de integração das classes pode influenciar os tipos de teste a serem feitos, e, consequentemente o custo e a eficácia do teste. Por isto, é tão importante considerar o processo de teste associado ao processo de desenvolvimento. Uma estratégia ruim pode levar à construção de muitos emuladores ou à combinação de classes não testadas previamente, dificultando o processo de localização de defeitos. Existem, na literatura,

alguns trabalhos que propõem metodologias/estratégias para planejar o teste de integração a partir de modelos UML, especificamente de diagramas de classes, com o intuito de analisar as dependências entres as classes e servem como base para que elas sejam testadas em uma ordem/sequência de integração que minimize os esforços gastos para realizar o teste de integração [22, 29, 42].

Uma vez que já existem estratégias que fazem análise de dependências entre classes e encontram ordens de integração eficientes para testes de integração de software OO, e o foco da abordagem aqui proposta é apenas na geração dos casos de teste, na abordagem é considerado que a ordem de integração já foi definida, ou por outras abordagens, ou a critério do testador. A ideia é, além de simplificar a abordagem, reutilizar estratégias eficientes. No entanto, para que fosse possível a geração dos casos de teste de acordo com a ordem de integração definida, foi preciso a adaptação dos modelos de desenvolvimento. Para isso, foi desenvolvido um perfil UML 2.0 no intuito que apenas os artefatos (classes e emuladores) necessários para o teste de integração fossem considerados de acordo com a ordem de integração escolhida pelo testador, a cada etapa do modelo incremental.

Perfis UML são mecanismos para extensão dos diagramas UML para adaptá-los à semântica de domínios específicos. Eles permitem que qualquer meta-classe da especificação de UML seja anotada com estereótipos que representam um conceito específico de um domínio. Conforme descrito, foi definido um perfil UML 2.0, chamado de *Integration Order Profile* (IOP), para adicionar aos modelos UML características que indicam quais classes devem ser agrupadas para os testes e a ordem de integração entre elas. Neste perfil, o papel de cada classe do sistema no teste de integração é especificado com estereótipos (*stereotype*) a serem aplicados em classes do diagrama de classes.

O conjunto de estereótipos do perfil IOP e a meta-classe que deve ser estendida com eles pode ser visto na Figura 3.4. Conforme mostrado na figura, os quatro estereótipos especificados devem ser aplicados à meta-classe *Classifier* uma vez que ela é um meta-elemento usado para modelar classes, objetos e componentes em modelos UML. Além disso, é importante destacar que o elemento *Classifier* a ser anotado com os estereótipos deve ser do tipo *Class* ou *Interface*, conforme descrito na restrição OCL também mostrada na figura. As semânticas associadas a cada estereótipo do perfil IOP são:

- «*ClassToBeIntegrated*». Em um modelo incremental, a cada passo, um

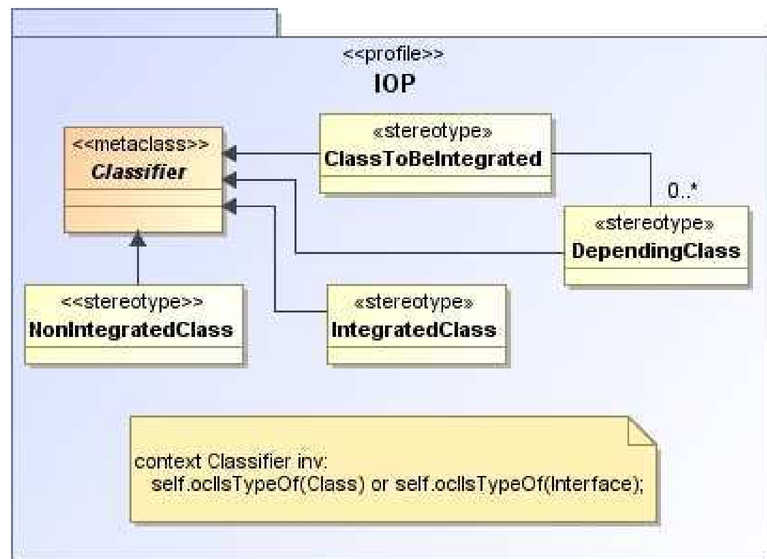


Figura 3.4: *Integration Order Profile* (IOP).

novo componente (também chamado de classe ou entidade), o qual já foi testado individualmente, é integrado e então disponibilizado para o teste de integração. Nesse contexto, o elemento (meta-classe *Classifier*) estereotipado com «*ClassToBeIntegrated*» representa o componente a ser integrado, ou seja, o componente a ser verificado em nível de teste de integração a cada passo. É importante enfatizar que um, e somente um, elemento *Classifier* do diagrama de classes deve ser anotado com este estereótipo. Nos modelos de teste a serem gerados, o elemento anotado com este estereótipo corresponderá ao SUT (*System Under Test*), e será anotado com o estereótipo «SUT» conforme descrito pelo perfil de teste U2TP;

- «*DependingClass*». Os elementos anotados com este estereótipo correspondem aos componentes que são dependentes do elemento estereotipado com «*ClassToBeIntegrated*». Esta anotação é importante porque, em teste de integração, muitas vezes é preciso considerar não apenas a classe a ser integrada, mas também as suas dependências. Elementos anotados com este estereótipo não podem ser estereotipados com «*ClassToBeIntegrated*» ao mesmo tempo;

- «*IntegratedClass*». Os elementos anotados com este estereótipo correspondem aos componentes que já foram testados individualmente e integrados para o teste de integração. Elementos anotados com este estereótipo não podem ser estereotipados

como «*ClassToBeIntegrated*» ao mesmo tempo. Porém, podem ser anotados como «*DependingClass*». Nos modelos de teste a serem gerados, os elementos anotados com este estereótipo serão considerados integralmente para os testes;

- «*NonIntegratedClass*». Os elementos anotados com este estereótipo correspondem aos componentes que ainda não foram integrados, ou seja, ainda não estão disponíveis para o teste de integração. Teste do nível de integração é, portanto, muitas vezes confrontado com o problema que o SUT necessita de funcionalidades de outros componentes que não estão prontos para a integração. Neste caso, emuladores, que simulam funcionalidades ausentes dos componentes requeridos para o SUT prover seu serviço, são necessários, conforme mostrado na Seção (2.1.2). Dessa forma, nos modelos de teste, os elementos anotados com «*NonIntegratedClass*» serão substituídos por emuladores e anotados como componentes de teste («*TestComponent*») de acordo com o perfil de teste U2TP, uma vez que se tratam de componentes construídos especificamente para testes. Elementos estereotipados com «*NonIntegratedClass*» não podem ser estereotipados com «*ClassToBeIntegrated*» ao mesmo tempo. Porém, podem ser anotados como «*DependingClass*». Além disso, elementos não identificados com nenhum estereótipo são considerados como «*NonIntegratedClass*».

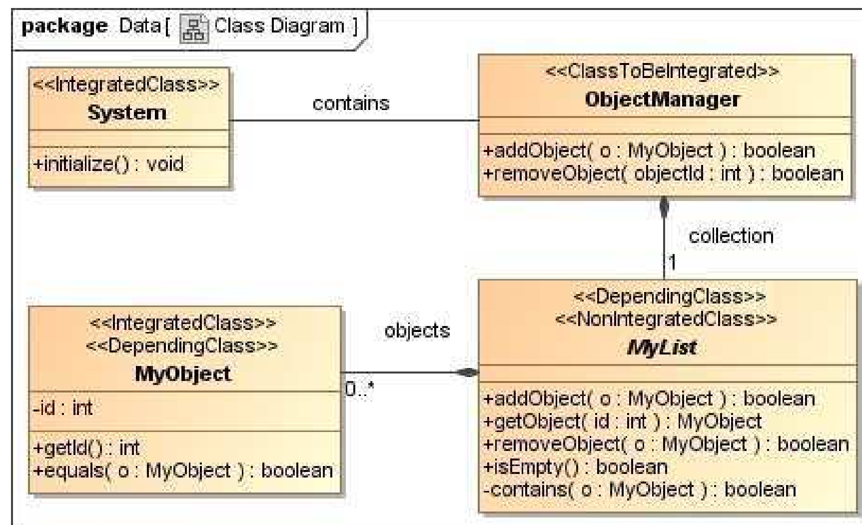


Figura 3.5: Exemplo de aplicação do perfil IOP.

A Figura 3.5 acima ilustra um digrama de classes UML com um exemplo de aplicação do perfil IOP. Neste diagrama, a classe *ObjectManager* é a classe a ser integrada (anotada com

o estereótipo «*ClassToBeIntegrated*»). As classes *System* e *MyObject*, neste caso, já foram testadas individualmente e integradas (anotadas com o estereótipo «*IntegratedClass*»). A classe *MyList*, por sua vez, ainda não está disponível para testes (anotada com o estereótipo «*NonIntegratedClass*»). Finalmente, as classes *MyList* e *MyObject* possuem relacionamento de dependência com classe a ser integrada.

Antes da segunda etapa da abordagem, que consiste na identificação dos padrões de teste aplicáveis aos modelos de desenvolvimento, as classes que não são necessárias para o teste de integração, ou seja, as classes que não interagem diretamente com a classe anotada com o estereótipo «*ClassToBeIntegrated*» devem ser excluídas dos modelos de projeto que serão fonte para a geração os casos de teste. Uma vez realizada a filtragem desses modelos, bem como a construção dos emuladores e a especificação do SUT e de suas dependências, estes estereótipos não serão mais necessários. Isto é, a aplicação deste perfil não será mais visualizada nos modelos de teste.

3.3 Identificação da Aplicação de Padrões de Teste

Para a aplicação de padrões de teste no nível de modelos, foi proposta uma nova forma de documentação de padrões de teste de maneira que os mesmos apresentem soluções independentes de plataforma para a geração de casos de teste. Tais soluções, as quais são documentadas usando o perfil de teste, são aplicadas a partir de modelos UML. Adicionalmente, esta nova forma de documentação possibilitará a identificação automática da possível utilização dos padrões para a geração dos casos de teste.

A seguir é apresentado um formato de definição proposto para a documentação de padrões de teste para sistemas OO com elementos que permitem a identificação da aplicabilidade do padrão, bem como exemplos ilustrando como o padrão deve ser aplicado em nível de modelos. Este formato é baseado no formato para padrões de teste proposto por Binder [11], onde os seguintes elementos, os considerados mais relevantes, são reutilizados: *nome*, *contexto*, *intenção*, *modelo de faltas*, *estratégia*, *padrões relacionados* e *referência*. À estes elementos, foram acrescentados os elementos *identificação*, cujo objetivo é ressaltar como o padrão pode ser aplicado em nível de modelos, e *exemplos*, para apresentar exemplos da aplicação da estratégia do padrão também em nível de modelos. Além disso, o elemento

estratégia foi adaptado no intuito de contemplar novas características de acordo com a abordagem.

O objetivo do desenvolvimento desse formato de definição para padrões de teste é facilitar a utilização deles baseados em modelos independentes de plataforma. Este formato serve tanto para a descrição de padrões de teste já existentes na literatura, como também para descrever novos padrões a serem desenvolvidos. A seguir, são descritos os elementos deste formato de definição. Os elementos *Nome*, *Contexto*, *Intenção*, *Identificação*, *Estratégia* e *Exemplos* são obrigatórios, e os demais são utilizados para enriquecer as definições.

- **Nome:** Palavra ou frase que identifica o padrão e sugere sua abordagem geral.
- **Contexto:** Descreve a aplicabilidade do padrão bem como as pré-condições que fazem com que a solução seja aplicada ao problema e com que esta solução seja vantajosa.
- **Intenção:** Breve descrição do conjunto de teste produzido pelo padrão de teste.
- **Modelo de Faltas:** Descreve os tipos de faltas que o padrão pode detectar. Estas faltas podem ser descritas formalmente ou heuristicamente (usando experiências ou considerações práticas).
- **Identificação:** Mostra como é possível identificar a aplicabilidade do padrão no nível de modelos. Para isso, foi desenvolvido um meta-modelo, chamado de *TestPatternMetamodel*, para a documentação dos padrões que permita essa identificação com base nos modelos. Este meta-modelo estende o meta-modelo de UML uma vez que trabalha-se com padrões de teste aplicados a modelos UML. Esta extensão não captura o que o padrão de teste é no geral, mas como ele é utilizado em um ou mais casos específicos, mostrando como ele pode ser aplicado em um ou mais casos específicos, no nível de modelos, e sua representação estrutural. A Figura 3.6 ilustra como esse meta-modelo estende o meta-modelo de UML, cujas meta-classes reusadas estão com fundo cinza.

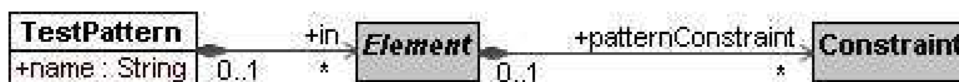


Figura 3.6: Extensão do meta-modelo de UML para padrões de teste.

De acordo com o meta-modelo, um padrão de teste (meta-classe *TestPattern*) é composto por elementos do modelo UML (meta-classe *Element*). Cada elemento UML a ser investigado deverá ter características específicas, representadas por um conjunto de regras (meta-classe *Constraint*), com base na definição do padrão. O objetivo do desenvolvimento deste meta-modelo é permitir a identificação automática da possível utilização dos padrões para a geração de casos de teste.

- **Estratégia:** Determina como o conjunto de teste deve ser projetado e implementado. Aqui é apresentada a solução independente de plataforma, ou seja, o modelo da solução aqui descrito será independente de onde o sistema a ser testado estará sendo executado. Dessa forma, os casos de teste gerados devem ser descritos em uma linguagem que seja passível de ser transformada em código de teste, como é o caso do perfil de teste da UML 2.0 (U2TP).

- **Exemplos:** Provê alguns exemplos de aplicações e utilizações do padrão. É comumente utilizado para facilitar o entendimento dos elementos da Estratégia. Os casos de teste aqui mostrados são documentados de acordo com U2TP.

- **Padrões Relacionados:** Apresenta os padrões que possuem alguma relação (similares ou complementares e necessários para uso).

- **Referência:** Caso o padrão já exista na literatura, este elemento informará o(s) seu(s) autor(es), bem como onde encontrar maiores informações sobre o padrão.

3.3.1 Padrões de Teste Documentados

Os padrões de teste utilizados nesse trabalho são: (i) *Round-trip Scenario Test* [11]; (ii) *Abstract Class Test* [11]; (iii) *Error Simulator* [28]; e (iv) *Test Case With Time Specification* [28]. Alguns desses padrões não são, originalmente, descritos para a geração de casos de teste interessados na interação entre as classes. No entanto, eles podem ser utilizados para este fim. Para isso, foram feitas algumas modificações, o que não descaracteriza o contexto do padrão, de forma a adaptá-los para teste de integração, o foco deste trabalho. A seguir, é apresentado o padrão *Round-trip Scenario Test* documentado de acordo com o formato de definição descrito anteriormente. Os demais padrões podem ser encontrados no Apêndice A.

Round-trip Scenario Test

Este padrão consiste na geração de casos de teste a partir de caminhos em um grafo de fluxo extraído a partir de diagramas de sequência UML. Ele foi escolhido porque sua solução permite a geração de casos de teste para caminhos completos do diagrama de sequência, permitindo a verificação das interações entre classes, e consequentemente, sendo apropriado para teste de integração. Seguindo o formato de definição descrito na seção anterior, tem-se:

- **Nome:** *Round-trip Scenario Test*.
- **Contexto:** Cenários *round-trip* são caminhos completos nos diagramas de sequência. Diagramas de sequência geralmente incluem estes cenários, porém nem sempre explícitos.
- **Intenção:** Extrair um grafo de fluxo a partir de diagramas de sequência e desenvolver casos de teste que provêem um mínimo de cobertura de caminhos.
- **Modelo de Faltas:** Faltas potenciais que podem ser encontradas com a utilização deste padrão para a geração de casos de teste são: saída faltando ou incorreta; falta de função/atributo em um participante; mensagem correta passada a um objeto errado; mensagem incorreta passada a um objeto certo; mensagem enviada a um objeto destruído; etc.
- **Identificação:** O código 3.1 apresenta uma instância do meta-modelo *TestPatternMetamodel*, representada no formato XMI (*XML Metadata Interchang*) [39], para este padrão. Nele, é possível perceber quais os elemento(s) do meta-modelo de UML deve(m) ser investigado(s) nos modelos para a aplicação do padrão. De acordo com a especificação do padrão, para que seja possível sua aplicação, basta que dentre os modelos exista um diagrama de sequência. Dessa forma, o elemento do meta-modelo de UML a ser casado é *Collaboration*, uma vez que tal elemento é a representação de um diagrama de sequência em modelos UML. Contudo, como o interesse é gerar casos de teste que envolvam a classe que deseja-se integrar, é preciso que a classe estereotipada como *ClassToBeIntegrated* faça parte do diagrama. Essa restrição pode ser visualizada pelo elemento (*patternConstraint*).

Código Fonte 3.1: Elementos que identificam a aplicação do padrão *Round-trip Scenario Test* de acordo com o meta-modelo *TestPatternMetamodel*.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tpm:Model xmlns:xmi="http://www.omg.org/XMI" xmlns:tpm="TestPatternMetamodel">
3      <testPattern name="Round Trip Scenario Test">
4        <in type="Collaboration">
5          <patternConstraint>
6            <specification type="tpm:OpaqueExpression"
7              body="context Collaboration inv:
8                  self.ownedBehavior->first().ownedAttribute->
9                  exists(att | att.type.getAppliedStereotypes()->
10                 exists(a | a.name = 'ClassToBeIntegrated'));"
11             language="OCL 2.0"/>
12         </patternConstraint>
13       <in/>
14     </testPattern>
15 </tpm:Model>
```

- **Estratégia:** O procedimento para a geração de casos de teste para esse padrão consiste nos seguintes passos:

  1. Transformar o diagrama de sequência em um grafo de fluxo. Este grafo é construído a partir da identificação de condições, segmentos e ramos, representados por retângulos, hexágonos e setas que levam a um hexágono, respectivamente. A Figura 3.7(b) mostra um grafo de fluxo extraído a partir do diagrama de sequência mostrado na Figura 3.7(a). Cada mensagem ou grupo de mensagens consecutivas, no diagrama de sequência, tornam-se segmentos, e cada condição em um fragmento combinado (*opt*, *alt*, *loop*, etc.) torna-se uma decisão (hexágono);

  2. Identificar caminhos a partir do grafo de fluxo. Cada caminho será equivalente a um cenário para a geração dos casos de teste. Qualquer cenário irá atingir todos os segmentos no grafo se ele não tiver decisões. Contudo, no caso do grafo ter várias decisões, o número de caminhos pode ser muito grande. Logo, é preciso selecionar os caminhos de acordo com um mínimo de cobertura de decisão;

  3. Identificar casos especiais. Em outras palavras, selecionar caminhos onde os casos de teste podem detectar mais defeitos ou defeitos potenciais;

4. Identificar entradas e estados necessários para a execução dos caminhos.

No contexto de geração de casos de teste no nível de modelos, ou seja, modelos de teste independentes de plataforma, apenas os passos 1 e 2 devem ser realizados. Após a realização desses passos, os casos de teste devem ser construídos para cada caminho identificado. Como o passo 3 consiste em estratégias de seleção de casos de teste, ou seja, a seleção dos casos de teste que serão executados, e o passo 4 consiste na escolha de dados para a execução dos casos de teste, eles só precisam ser realizados em nível específico de plataforma e código.

- **Exemplos:** Considerando o diagrama de sequência *Remove Object* mostrado na Figura 3.7(a), seguindo a estratégia do padrão, descrita anteriormente, ele foi transformado em um grafo de fluxo, conforme mostrado na Figura 3.7(b) (passo 1).

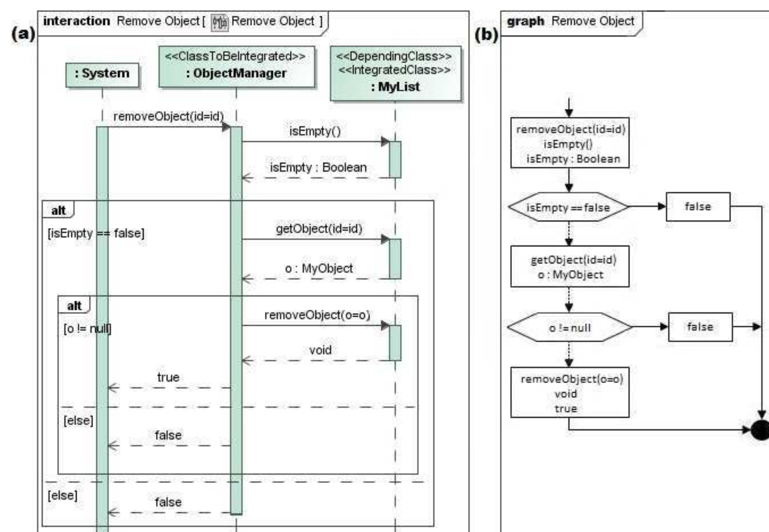


Figura 3.7: (a) Diagrama de sequência *Remove Object*, e (b) seu grafo de fluxo correspondente.

Uma vez construído o grafo de fluxo, três caminhos (cenários *round-trip*) foram identificados, conforme mostrado na Figura 3.8 (passo 2). De acordo com a estratégia do padrão, para cada um dos caminhos identificados, existe pelo menos um caso de teste associado. Nesse contexto, foi criado um caso de teste para cada caminho identificado conforme mostrado na Figura 3.9 (na mesma ordem de apresentação). Estes casos de teste, em nível de modelos, foram documentados de acordo com U2TP.

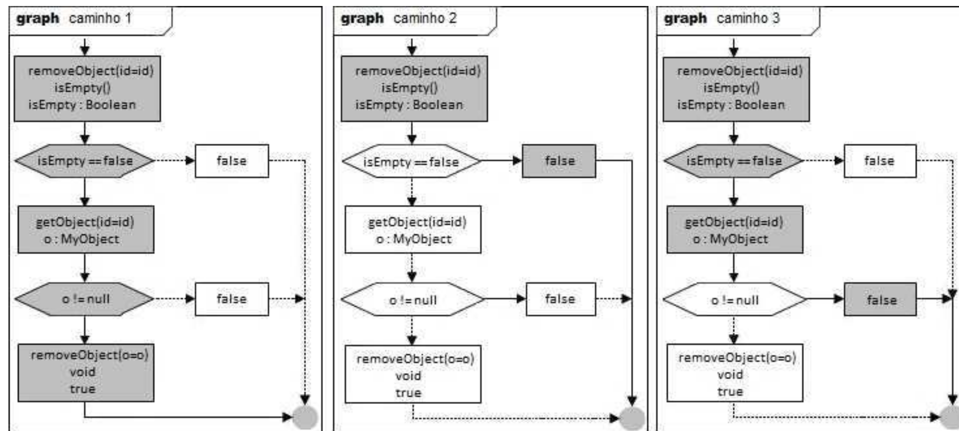


Figura 3.8: Caminhos para o diagrama de sequência *Remove Object*, sem loops ou condições, gerados a partir do grafo de fluxo.

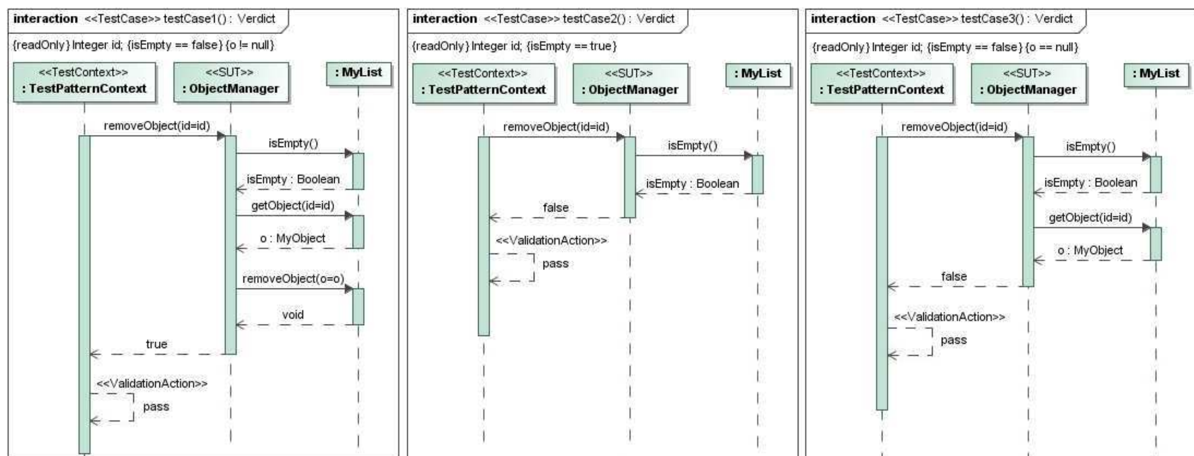


Figura 3.9: Casos de teste gerados para o diagrama de sequência *Remove Object*, documentados de acordo com U2TP e aplicando o padrão.

É importante ressaltar que, para cada cenário descrito, as condições em cada hexágono de decisão, passam a ser as pré-condições, e, consequentemente, algumas entradas do caso de teste. Os resultados esperados do caso de teste consistem na sequência de mensagens executada exatamente como descrito pelo diagrama de sequência que o representa e na «*ValidationAction*» no final do caso de teste, que corresponde ao veredito do caso de teste.

- **Padrões Relacionados:** Outros padrões podem ser utilizados para a seleção dos casos de teste gerados após a aplicação desse padrão. São exemplos: *Extended Use Case Test* [11], *Controlled Exception Test* [11], etc.

- **Referência:** Este padrão foi descrito por R. Binder e pode ser encontrado em [11].

3.4 Geração de Casos de Teste de Integração

Na definição de um padrão de teste, um elemento presente em todo formato é a estratégia ou solução que determina como o conjunto de teste deve ser projetado e implementado. Este elemento consiste de uma sequência de passos que o testador deve seguir para a implementação do padrão que, muitas vezes, são ilustrados por fragmentos de código. Contudo, esses mesmos passos também podem ser aplicados em um nível de abstração mais alto.

A ideia da abordagem proposta é que, uma vez identificada uma possível aplicação de um padrão, modelos de teste sejam desenvolvidos, e que casos de teste, documentados de acordo com U2TP, sejam gerados para verificar determinadas situações (definidas pelos padrões de teste) em que defeitos podem ser encontrados. Nesta abordagem, os casos de teste são representados por diagramas de sequência da UML 2, conforme pode ser visto na estratégia de cada padrão utilizado. Além disso, o diagrama de classes (modelo de desenvolvimento) é estendido, para que as novas classes construídas especificamente para testes sejam incorporadas.

A subseção a seguir apresenta, de forma genérica, como os artefatos (arquitetura e casos de teste de integração) devem ser construídos. Em outras palavras, um conjunto de atividades/orientações gerais, independentes do padrão a ser escolhido, que deve ser realizado para a geração desses artefatos é mostrado.

3.4.1 Documentando e Gerando os Casos de Teste de Acordo com U2TP

Primeiramente, assume-se a definição de um pacote de teste (*package*) referente ao sistema que será testado, o qual deve ser adicionado ao diagrama de classes do sistema. Neste pacote devem ser incluídos todos os artefatos, especificamente a arquitetura de teste e o comportamento de teste, criados especificamente para os testes de acordo com a abordagem proposta e documentados de acordo com U2TP, conforme mostrado a seguir.

Orientações para a Arquitetura de Teste

- Toda classe a ser testada no contexto de teste de integração, ou seja, aquela estereotipada como «*ClassToBeIntegrated*», deverá ser anotada no modelo de teste como «SUT». O elemento «SUT» é obrigatório;

- Deve haver uma, e somente uma, classe no pacote de teste estereotipada com «*TestContext*». Nesta, cada caso de teste deve ser representado através de uma operação estereotipada com «*TestCase*». Deve haver no mínimo uma operação «*TestCase*»;

- Podem ser especificadas várias classes estereotipadas com «*TestComponent*». Estas classes podem ser especificadas tanto no projeto do software, quanto no pacote de teste. Quando definido no projeto do software, especifica-se que a classe SUT depende de funcionalidade de classes desse tipo. No pacote de teste, é possível criar novas classes representando apenas aqueles aspectos necessários ao teste, de acordo com a necessidade do padrão. No contexto de teste de integração, os emuladores, substituição para as classes estereotipadas como «*NonIntegratedClass*», devem também ser anotadas com «*TestComponent*», pois são classes construídas especificamente para testes.

Orientações para o Comportamento de Teste

O comportamento de um caso de teste deve ser especificado através de diagramas de sequência. São assumidas as seguintes convenções para este tipo de diagrama:

- O diagrama de sequência deve ter o nome do caso de teste;

- O objeto iniciador da interação do diagrama de sequência deve ser uma instância da classe estereotipada com «*TestContext*»;

- Os demais objetos do diagrama de sequência são instâncias de classes do tipo SUT, «*TestComponent*» ou de qualquer classe do projeto do software. Estas instâncias podem ser nomeadas, e podem ser usadas como variáveis em outras mensagens do diagrama;

- Só é permitido um veredito por caso de teste, representado pela última mensagem do diagrama, na forma de uma auto-delegação ao objeto *TestContext*. A mensagem é anotada com «*ValidationAction*» e deve ser *pass* ou *fail*. A Figura 3.10 apresenta duas formas de especificar essa mensagem;

Figura 3.10: Exemplos de mensagens de retorno do veredito.

- Geralmente, o veredito é especificado como *pass*, no entanto, se durante a execução do sistema para o caso de teste, algo sair diferente do que foi especificado pelo diagrama, é porque o caso de teste falhou (*fail*) e então existe algum defeito no sistema;

- Admite-se o uso de restrições OCL para especificar pré-condições do caso de teste. Estas pré-condições também podem ser especificadas por variáveis passadas por parâmetro do caso de teste.

As Figuras 3.11 e 3.12, mostram como artefatos de teste foram construídos usando as orientações acima descritas. Estes artefatos de teste, especificamente um caso de teste e a arquitetura de teste, exemplificam a utilização do perfil de teste U2TP para o padrão de teste *Round-trip Scenario Test*. Nas figuras é possível perceber, dentre outros artefatos: (i) o contexto de teste (classe *TestPatternContext* anotada com o estereótipo «*TestContext*»); (ii) o SUT (a classe *ObjectManager* anotada com o estereótipo «SUT»); (iii) o tipo de veredito possível (a classe *Verdict*); (iv) e o caso de teste (*interaction*) cujo nome é o mesmo de um método do contexto de teste. Este último é um caminho para o diagrama de sequência apresentado na Figura 3.7(a), mostrada anteriormente na definição do padrão. Nele é possível verificar as pré-condições (restrições OCL no topo do diagrama) e o veredito (*pass*) ao final.

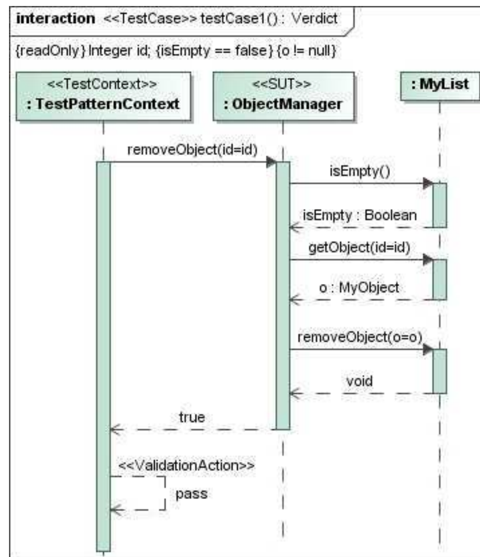


Figura 3.11: Diagrama de sequência mostrando um caso de teste de acordo com U2TP.

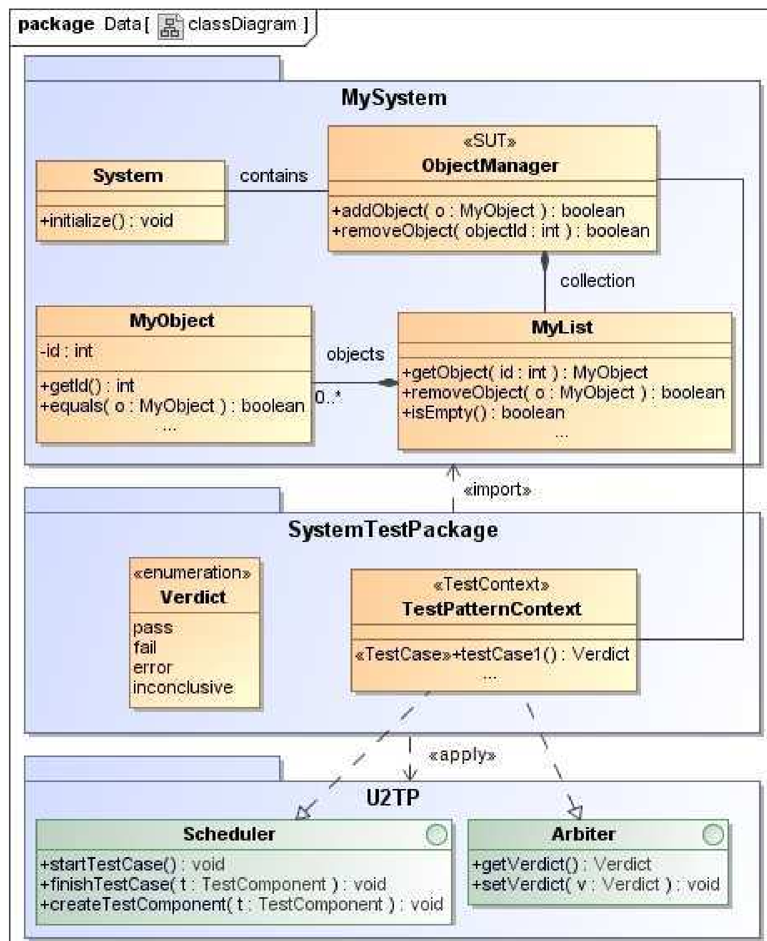


Figura 3.12: Diagrama de classes mostrando a arquitetura de teste de acordo com U2TP.

3.5 Geração Automática de Casos de Teste de Integração baseada em Padrões de Teste

A existência de um suporte automático é de grande importância para viabilizar a aplicação da abordagem de uma forma prática, dado que em teste de integração o número de combinações a serem testadas pode ser muito grande. Uma vez que a abordagem é inserida dentro de um processo integrado de desenvolvimento e testes dirigidos por modelos (MDD/MDT) [3], sua aplicação é passível de automação a partir do momento em que são utilizadas práticas de MDA. Como a ideia-chave de MDA é mudar a ênfase na implementação (código) para o foco na modelagem, meta-modelagem e transformação entre modelos, casos de teste podem ser gerados de forma automática a partir de modelos de desenvolvimento, pois a mesma abstração em termos de modelagem (independente e específica) de plataforma e geração de código definida por MDA pode ser aplicada para projetar modelos de teste com MDT. A Figura 3.13 apresenta a arquitetura da abordagem no contexto de MDA para a geração dos casos de teste de integração usando padrões de teste.

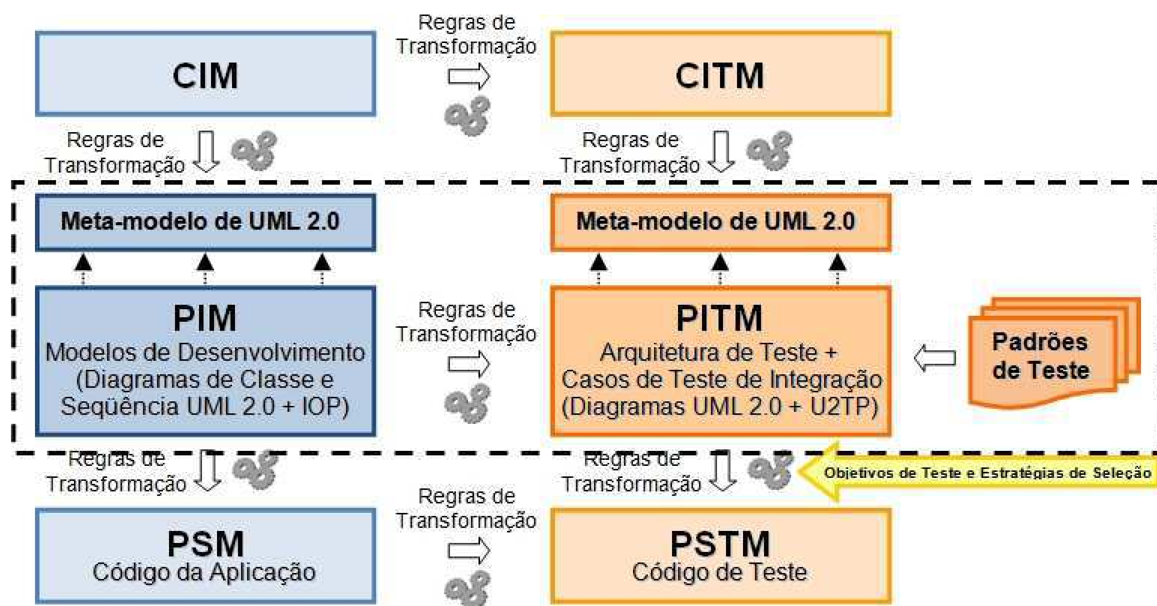


Figura 3.13: Arquitetura da abordagem de acordo com os princípios de MDA.

Nesta arquitetura, os seguintes conceitos de teste foram introduzidos [3]: (i) CITM, modelos responsáveis por capturar os objetivos de teste; (ii) PITM, modelos que refletem a arquitetura de teste e os casos de teste abstratos; e (iii) PSTM, modelos que representam

os artefatos de testes específicos para uma plataforma. Conforme mostrado na Figura 3.13, modelos de teste podem ser gerados tanto a partir de modelos de teste, como diretamente de modelos de desenvolvimento em vários níveis de abstração (por exemplo, CIM-CITM, PIM-PITM e PSM-PSTM). Além disso, toda a estrutura necessária para a execução dos testes em diferentes plataformas também pode ser gerada automaticamente. Embora esta arquitetura descreva todos os níveis de abstração de modelos e geração de código, o foco da abordagem proposta é apenas na geração de casos de teste, ou seja, nos modelos no nível de abstração independente de plataforma PIM-PITM, os quais não incluem qualquer informação acerca da plataforma, detalhes de implementação ou artefatos de construção dos testes. Além disso, o conceito de padrões de teste foi incorporado ao processo.

Segundo Alves *et al.* [3] em um processo integrado de MDD/MDT utilizando MDA, é preciso eleger, antes de qualquer coisa, as notações, metáforas e metodologias a serem utilizadas de modo que estas sejam compatíveis com MDD e MDT. Além disso, os processos de MDD e MDT a serem empregados devem ser fielmente definidos. Para isso, é preciso que seja cuidadosamente planejado um alinhamento entre as características principais de cada processo tais como: meta-modelos utilizados, artefatos gerados (modelos, código, etc.), técnicas, entre outros. As principais características do processo de MDD considerado na abordagem automática são:

- **Modelos:** seguindo o padrão para realização de MDA proposto pela OMG, os modelos desse processo são documentados de acordo com a versão 2.0 da UML. Os modelos do nível PIM são, especificamente, diagramas de classes (estruturais) e de sequência (comportamentais), anotados com o perfil de integração IOP, descrito na seção anterior;

- **Meta-modelos:** o meta-modelo de UML, na sua versão 2.0;

- **Transformações:** para que seja realizados refinamentos no nível PIM-PIM, são utilizadas regras de transformação entre modelos. Para a construção dessas transformações, adotou-se a linguagem de transformação de modelos ATL (*Atlas Transformation Language*) [5]. Apesar de não ser o padrão proposto pela OMG para linguagem de transformação (a OMG propõe o formalismo QVT [41]), a escolha de ATL foi motivada por ser a mais utilizada na prática [3].

Quanto ao processo de MDT são:

- **Modelos:** da mesma forma que os modelos de desenvolvimento, os modelos desse processo são documentados de acordo com a versão 2.0 da UML. Além disso, o perfil de teste U2TP é utilizado para documentação dos artefatos de teste. Os modelos do nível PITM são, especificamente, diagramas de classes, extensões dos modelos PIM para incorporar os conceitos referentes à arquitetura de teste, e diagramas de sequência, representando os casos de teste;
- **Meta-modelos:** o meta-modelo de UML, na sua versão 2.0;
- **Transformações:** para que sejam gerados os modelos na direção PIM-PITM, são utilizadas regras de transformação entre modelos. Para a construção dessas transformações, adotou-se também a linguagem de transformação de modelos ATL.

3.6 Considerações Finais

Neste Capítulo foi apresentada uma nova abordagem de teste, definida dentro de um processo integrado de desenvolvimento e teste dirigidos por modelos (MDD/MDT), para a geração de casos de teste a partir de modelos de desenvolvimento. A abordagem, cujo foco é em teste de integração, utiliza-se de padrões de teste como base para a geração dos casos de teste, e do perfil de teste da UML (U2TP) para a documentação dos casos de teste a fim de que os mesmos sejam independentes de plataforma, ou seja, independentes de linguagem de programação. Todas as etapas da abordagem foram apresentadas em detalhes e exemplos ilustrando como aplicá-la foram mostrados a medida que as etapas iam sendo apresentadas. A abordagem, apesar de ter sido apresentada como um procedimento manual, foi desenvolvida tendo a automação como um dos requisitos, conforme destacado na Seção 3.5 e que será abordado com mais detalhes no Capítulo 4.

Uma das vantagens dessa abordagem é, além de utilizar ideias/estratégias que já foram utilizadas e se mostraram efetivas descritas pelos padrões de teste para a geração de casos de teste mais importantes e adequados, fazer uso apenas de artefatos produzidos pelo processo de desenvolvimento utilizado, especificamente, dos modelos de desenvolvimento, descritos por diagramas UML. Isto permite que a abordagem tenha um impacto reduzido

no total de esforço a ser empreendido no processo como um todo, uma vez que não é preciso a construção de modelos específicos para testes pelos testadores. No entanto, é importante destacar que é preciso que os modelos de desenvolvimento sejam suficientemente completos, detalhados, consistentes e corretos, para que eles possam ser usados diretamente pela abordagem para a a geração dos casos de teste. Uma forma de assegurar a consistência e corretude de modelos é proposta por Rocha [45], que automatiza a técnica de inspeção guiada no intuito de detectar defeitos semânticos e sintáticos em diagramas UML 2.0.

Ainda, como toda abordagem baseada em modelos, a abordagem apresentada é dependente da disponibilização das especificações na forma de diagramas de classes e sequência, principalmente porque ela está inserida em um processo integrado de MDD/MDT. A sua aplicação é, portanto, restrita a ambientes onde modelos são construídos e mantidos como parte do processo de desenvolvimento. Uma forma de aplicação da abordagem para sistemas legados, ou seja, sistemas que já foram desenvolvidos e estão operacionais é possível se for feito um processo de engenharia reversa para a obtenção dos modelos. No entanto, a aplicação da abordagem, nesse caso, perde o sentido porque o intuito é gerar os testes antes do código estar pronto (ou pelo menos em paralelo).

Outra limitação da abordagem é a anotação dos modelos com o perfil IOP. Atualmente, para a realização da abordagem, é preciso que o testador use sua experiência para a escolha de uma ordem de integração a ser testada e anotação das classes do diagrama de classes nos modelos de desenvolvimento para poder gerar os casos de integração desejados. No entanto, essa anotação poderia ser feita com base em análises de dependências entre classes. Existem, na literatura, alguns trabalhos que propõem estratégias para fazer este tipo de análise a partir de modelos UML. Logo, uma forma de interação desses trabalhos a a abordagem aqui proposta deverá ser investigada no intuito de simplificar a aplicação da abordagem bem como reutilizar estratégias eficientes.

Com relação à utilização da abordagem para geração de casos de teste em outros níveis (por exemplo unidade ou sistema) é possível aproveitar a ideia dos padrões apenas. Em particular, para teste de sistema, em alguns casos é possível aplicar a abordagem no último nível de integração, onde todas as classes já foram testadas individualmente e foram integradas.

Capítulo 4

Suporte Ferramental

O objetivo deste Capítulo é apresentar a ferramenta que foi desenvolvida para dar suporte automático à abordagem dirigida por modelos proposta. Inicialmente, na Seção 4.1, uma visão geral da ferramenta é apresentada, destacando as características para as quais o seu projeto foi dirigido. Nas Seções seguintes, são detalhados cada um dos módulos específicos da ferramenta correspondentes à cada uma das atividades principais da abordagem. Por fim, na Seção 4.5, são apresentados alguns comentários gerais sobre o Capítulo, ressaltanto como utilizar e estender a ferramenta.

4.1 Visão Geral

A abordagem proposta no Capítulo 3 possibilita que casos de teste mais adequados sejam gerados, uma vez que são utilizadas soluções existentes (documentadas por padrões de teste) que se mostraram efetivas no processo de geração. Contudo, para a adoção em um ambiente real, ele possui alguns problemas práticos devido aos seguintes fatos:

1. **Necessidade de aprendizado de novos conceitos.** Os testadores terão que conhecer os novos artefatos propostos pela UML 2.0, principalmente, o perfil de teste U2TP, e detalhes minuciosos sobre os padrões de teste (como funciona, sua estratégia, como aplicar o perfil U2TP, etc.);

2. **Criação manual dos casos de teste.** Os testadores terão que analisar os modelos de desenvolvimento e o catálogo de padrões de teste utilizado, bem como criar cenários de

integração para então, a partir da ordem de integração definida para cada cenário, gerar os casos de teste. No contexto de teste de integração, dependendo da complexidade do sistema, vários cenários de integração podem ser críticos e fundamentais para testes, e, além disso, levar a um grande número de casos de teste. Logo, a criação dos casos de teste pode ser muito custosa.

Para solucionar os problemas citados, foi desenvolvida uma ferramenta no intuito de oferecer suporte automático à abordagem proposta. A Figura 4.1 mostra uma visão geral do funcionamento da ferramenta.

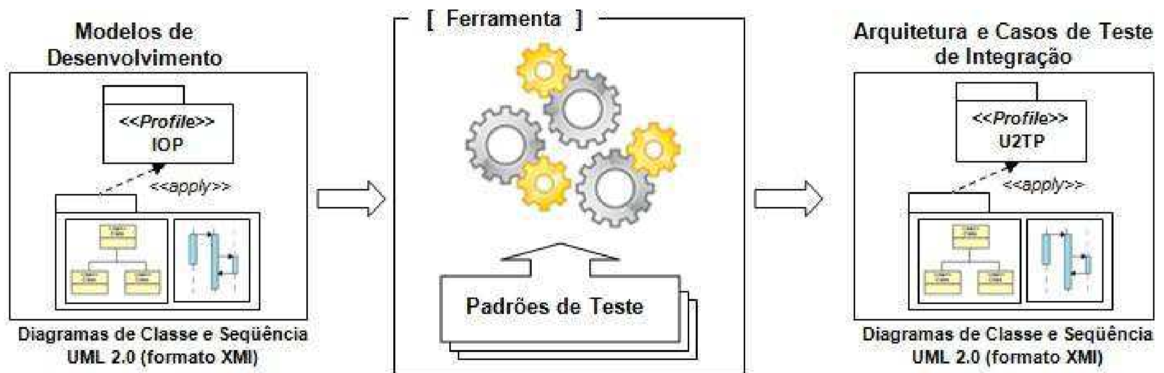


Figura 4.1: Visão geral do funcionamento da ferramenta.

Entre as características gerais da ferramenta, pode-se destacar:

- **Integração com as ferramentas de modelagem UML.** A ferramenta aceita como entrada diagramas de classe e sequência UML 2.0 com o perfil de integração IOP aplicado no formato XMI [39]. Logo, a ferramenta é independente de ferramenta de modelagem, sendo apenas necessário que estas ferramentas disponibilizem a opção de exportação de seus diagramas para este formato;

- **Geração de casos de teste a partir de apenas diagramas UML.** Os casos de teste são gerados a partir de apenas diagramas UML e documentados de acordo com o perfil de teste U2TP. Nenhum outro formalismo, além da linguagem UML, é necessário;

- **Disponibilização dos modelos de teste.** Uma vez feita a geração, os modelos de teste (arquitetura e casos de teste de integração) resultantes ficam disponíveis para possível utilização posterior (refinamentos, geração de código, etc), também no formato XMI;

- **Utilização da ferramenta em outros contextos de teste.** Adicionalmente, a ferramenta pode ser utilizada em um contexto geral de testes e não necessariamente no contexto de teste de integração, ou seja, pode ser utilizada para gerar casos de teste de sistema ou de unidade, por exemplo, dependendo da aplicabilidade dos padrões de teste considerados.

A ferramenta é uma *CASE* para MDT e foi implementada utilizando os princípios de MDA. Ela consiste em um conjunto de módulos contendo meta-modelos, modelos e regras de transformação entre modelos. Conforme introduzido no capítulo anterior, estas regras são especificadas e executadas com a linguagem ATL, e aplicadas em modelos que instanciam o meta-modelo MOF [39] da UML 2.0. Apesar da linguagem ATL não ser o padrão proposto pela OMG para esse tipo de transformação, ela foi utilizada porque, diferente de QVT (o padrão da OMG), possui um framework de criação e execução amplamente utilizado. Além disso, a linguagem está completamente alinhada com os atuais padrões da OMG.

O intuito de usar os princípios da MDA para construir a ferramenta, além do fato da abordagem estar inserida em um processo integrado de MDD/MDT, é que os seguintes requisitos desejáveis para a ferramenta, poderiam ser alcançados:

- **Automação.** Dado que os modelos de desenvolvimento (PIM) são independentes de plataforma, uma vez definidos, as transformações se tornam automáticas, ou seja, apenas estes modelos precisam ser atualizados. Os modelos de teste (PITM), são gerados automaticamente. Tal característica permite reduzir o tempo do processo de desenvolvimento de software como um todo, uma vez que número de interações humanas necessárias é minimizado;

- **Portabilidade.** Uma vez que o PITM é, por definição, independente de plataforma, ele pode ser transformado automaticamente em vários PSTMs específicos para as diferentes plataformas. Dessa forma, os casos de teste gerados podem ser transformados em casos de teste executáveis em qualquer plataforma;

- **Alinhamento entre desenvolvimento e testes.** Uma vez que todo o trabalho de manutenção é realizado no nível de abstração PIM, quando ocorre uma mudança, as partes que não são afetadas pelas mudanças são imediatamente reusáveis. Isto permite

um perfeito alinhamento entre as atividades de desenvolvimento e testes. Além disso, uma melhor facilidade de manutenção e atualização constante da documentação são alcançadas;

- **Reusabilidade.** Dado que os modelos de teste (PITM) gerados são independentes do domínio da aplicação, estes podem ser reutilizados, em parte ou completamente, para outros sistemas.

Para dar suporte à cada uma das três atividades da abordagem proposta (adaptação dos modelos de desenvolvimento para teste de integração, identificação da aplicação de padrões de teste e geração de casos de teste) a ferramenta foi dividida em três módulos principais, conforme pode ser visto na arquitetura mostrada na Figura 4.2:

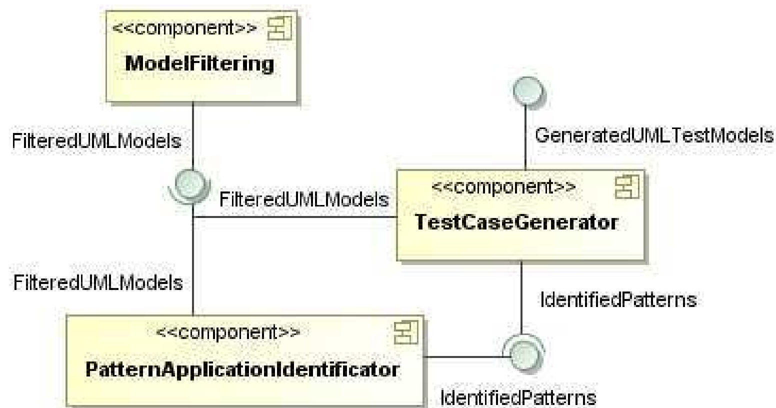


Figura 4.2: Arquitetura geral da ferramenta.

1. **Módulo de filtragem dos modelos:** responsável por filtrar os modelos de desenvolvimento para que apenas as classes necessárias para o teste de integração sejam consideradas;

2. **Módulo de identificação da aplicação de padrões de teste:** responsável por identificar os possíveis padrões de teste que podem ser aplicados aos modelos de desenvolvimento para a geração dos casos de teste;

3. **Módulo de geração de casos de teste:** responsável pela geração dos casos de teste de integração para os modelos de desenvolvimento filtrados.

4.2 Módulo de Filtragem dos Modelos

Este módulo consiste em um refinamento dos modelos de desenvolvimento na direção PIM-PIM com a finalidade de filtrar os diagramas UML 2.0 recebidos como entrada da ferramenta para que apenas as classes necessárias para o teste de integração sejam consideradas. Para isso, é preciso que os diagramas estejam bem formados, ou seja consistentes e de acordo com a especificação da UML 2.0 (meta-modelo) e que o perfil de integração IOP tenha sido corretamente aplicado. Segundo a especificação deste perfil, ele não terá sido corretamente aplicado se uma das três seguintes situações acontecerem:

- Existir mais de uma classe anotada com o estereótipo «*ClassToBeIntegrated*», pois na abordagem as classes são integradas uma de cada vez em cada iteração;
- Existir uma classe anotada com o estereótipo «*ClassToBeIntegrated*» e com alguns dos outros três estereótipos do perfil, «*DependingClass*», «*IntegratedClass*» ou «*NonIntegratedClass*», simultaneamente;
- Existir uma classe anotada com o estereótipo «*IntegratedClass*» e com o estereótipo «*NonIntegratedClass*» simultaneamente.

No contexto de teste de integração, as classes que não são necessárias para o teste são aquelas que não interagem diretamente com a classe que deseja-se integrar e testar, ou seja, aquelas que não têm uma relação de dependência com esta classe. Segundo o perfil, as classes anotadas com o estereótipo «*DependingClass*» representam as classes dependentes da classe que se deseja integrar (anotada com o estereótipo «*ClassToBeIntegrated*») e que serão consideradas para os testes. Estas classes, deverão estar anotadas com o estereótipo «*IntegratedClass*» ou «*NonIntegratedClass*». Estes dois estereótipos definem, respectivamente, se a classe é considerada integralmente no teste ou se ainda não está disponível, sendo então substituída, nos modelos de teste, por um emulador correspondente.

Nesse módulo, as demais classes, ou seja, as que não estão estereotipadas com «*DependingClass*» ou «*ClassToBeIntegrated*», são excluídas dos modelos de entrada PIM, resultando em modelos no mesmo nível de abstração que servirão de entrada para a geração dos casos de teste (refinamento PIM-PIM). Além disso, todos os elementos dos diagramas que fazem referência à estas classes, por exemplo, associações, linhas de vida (*lifeline*),

atributos, entre outros, também são excluídos. Isto se faz necessário para que os diagramas resultantes não contenham informações inconsistentes que interfiram na geração dos casos de teste. É importante ressaltar que a aplicação do perfil IOP não será mais visualizada nos modelos de teste. Uma vez aplicada a filtragem, na geração dos casos de teste, a classe anotada com «*ClassToBeIntegrated*» será especificada como «SUT» e as classes que serão substituídas pelos emuladores serão especificadas como «*TestComponent*», de acordo com a especificação de U2TP.

A Figura 4.3 mostra a abordagem MDA adotada para esse módulo, onde estão representados os seguintes artefatos:

- **Meta-modelos:** como esse módulo gera modelos UML a partir de modelos UML no nível PIM, então o meta-modelo adotado tanto para a origem quanto para o destino é o da UML 2.0 [40], provido pela OMG;

- **Modelos:** os modelos de entrada são os diagramas de classe e sequência UML contendo o projeto estrutural e comportamental do software, enquanto os de saída são os mesmos modelos refinados;

- **Transformações:** as transformações são especificadas na linguagem ATL.

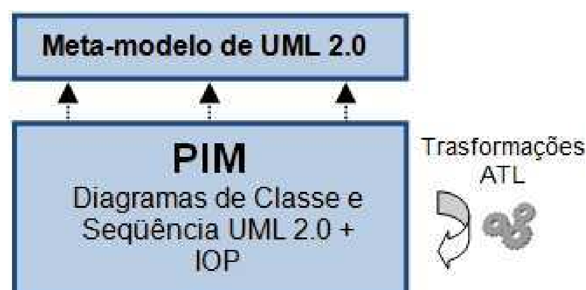


Figura 4.3: Abordagem MDA do módulo de filtragem dos modelos.

Transformações em ATL

As regras de transformação ATL especificadas para esse módulo são, basicamente:

1. **Regra 1 - filtro dos elementos no geral.** Esta é a regra principal, chamada de *filter*, e consiste numa *matched rule* que copia todos os elementos estruturais do *Model*

(elemento raiz de um modelo UML) a serem considerados para o teste de integração do modelo UML origem para o modelo UML destino;

2. **Regras 2 e 3 - verificação da aplicação do perfil IOP.** Esta regras, chamadas de *verifyNumberOfClassToBeIntegrated* e *verifyStereotypeCombinations*, são do tipo *called rule* e são chamadas pelo bloco imperativo *do* da regra principal *filter*. Elas verificam se o perfil IOP foi corretamente aplicado ao modelo de entrada PIM;

3. **Regras secundárias.** São regras do tipo *called rule* que auxiliam no filtro dos elementos desejados do modelo UML origem para o modelo UML destino. São elas: *filterClasses*, *filterAssociations*, e *filterCollaborations*. Esta última permite copiar um elemento do diagrama de sequência do modelo UML origem para o modelo UML destino fazendo referência à regra *filterCollaboration* no mesmo módulo. É importante ressaltar que só são copiados os elementos do diagrama que não fazem referência aos elementos que foram excluídos com a execução das demais regras de filtro.

Todas essas regras fazem referência a regras genéricas, ou seja, regras que são usadas por mais de um módulo, e que estão documentadas em um arquivo ATL separado (*Common.atl*). Cada módulo estende esse arquivo para reutilizar essas regras. O código 4.1 ilustra a regra principal *filter* desde módulo. As demais regras podem ser encontradas no site da ferramenta [32]. O código 4.1 está organizado da seguinte forma:

- Linha 1: identifica o módulo dessa transformação, mostrando que trata-se da filtragem dos modelos;

- Linha 2: identifica que os modelos de entrada e saída seguem o meta-modelo da UML. Além disso, o modelo de entrada segue também o perfil IOP;

- Linha 4: mostra o nome da regra responsável por copiar todos os elementos estruturais necessários do modelo UML origem para o modelo UML destino;

- Linha 5: mostra que essa *matched rule* será executada para o elemento *Model* do modelo UML origem;

- Linhas 6-8: cria o elemento *Model* do meta-modelo de UML no modelo destino;

- Linhas 9-23: é a parte imperativa da regra, é nesse trecho onde são chamadas as regras para a verificação da aplicação do perfil IOP e as regras que auxiliam no filtro dos elementos desejados do modelo UML origem para o modelo UML destino.

Código Fonte 4.1: Regra ATL do módulo de filtragem dos modelos.

```
1  module ModelFiltering;
2  create OUT : UML2 from IN : UML2, IOP : UML2;
3  rule filter {
4   from m_in:UML2!Model
5   to m_out:UML2!Model (
6    packagedElement <- m_in.packagedElement
7   )
8   do {
9    thisModule.model <- m_out;
10   thisModule.IOP_Profile <- UML2!Profile.allInstances()->select(e|e.name='IOP')->first();
11   for (profile in UML2!Profile.allInstances()){
12    m_out.applyProfile(profile);
13   }
14   self.verifyNumberOfClassToBeIntegrated(classes);
15   self.verifyStereotypeCombinations(classes);
16   self.filterClasses(classes);
17   self.filterAssociations(elements->select(e | e.oclIsTypeOf(UML2!Association)));
18   self.filterCollaborations(elements->select(e | e.oclIsTypeOf(UML2!Collaboration)));
19  }
20 }
```

4.3 Módulo de Identificação da Aplicação de Padrões de Teste

Este é um módulo intermediário que tem o objetivo de identificar os padrões de teste que podem ser aplicados para a geração dos casos de teste. A ideia deste módulo é identificar os padrões que podem ser aplicados e quais os elementos do modelo UML que permitem a aplicação do padrão de acordo com o meta-modelo proposto na Seção 3.3. Este módulo é necessário para que o testador possa enxergar onde os padrões poderiam ser aplicados e então decidir quais casos de teste gerar (pode ser do interesse do testador gerar casos de teste para apenas um dos padrões, por exemplo). A Figura 4.5 mostra a abordagem MDA adotada para esse módulo, onde estão representados os seguintes artefatos:

- **Meta-modelos:** este módulo gera um modelo com todos os padrões de teste aplicáveis e os respectivos elementos que identificaram a aplicação do padrão com base nos modelos UML. Nesse caso, dois meta-modelos são adotados: (i) o origem sendo o da UML 2.0, e (ii) o destino como sendo o seguinte meta-modelo, chamado de *ApplicableTestPatterns*:

Figura 4.4: Meta-modelo *ApplicableTestPatterns* para os padrões de teste identificados.

  onde, *TestPattern* é o padrão a ser aplicado e *Element* é o elemento que identificou a aplicação do padrão (*type* é o tipo do elemento UML);

- **Modelos:** os modelos de entrada são os diagramas de classe e sequência UML contendo o projeto estrutural e comportamental do software, enquanto que o modelo de saída é um modelo com todos os padrões de teste identificados e respectivos elementos que o identificaram;

- **Transformações:** as transformações são especificadas na linguagem ATL.

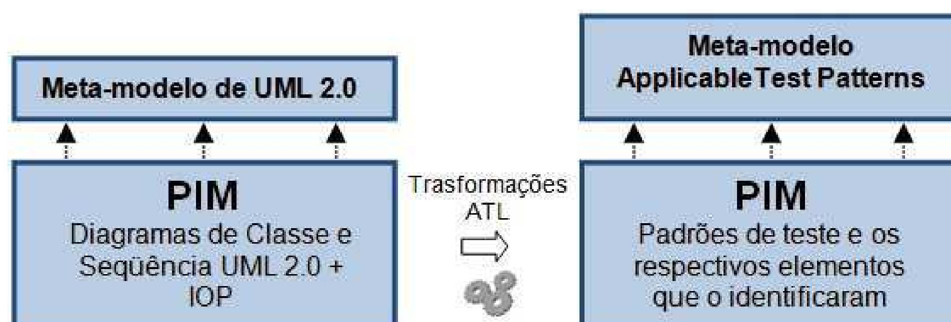


Figura 4.5: Abordagem MDA do módulo de identificação da aplicação de padrões de teste.

Transformações em ATL

As regras de transformação ATL especificadas para esse módulo são do tipo *matched rule*, onde cada uma é responsável por identificar, no modelo origem, a aplicação de um dos

padrões de teste documentados. O nome de cada regra é o nome do respectivo padrão que ela identifica. Cada regra, além de identificar se o padrão pode ser aplicado para a geração dos casos de teste, gera um modelo de acordo com meta-modelo *ApplicableTestPatterns* fazendo referência ao elemento do modelo que identificou a aplicação. O código 4.2 ilustra uma das regras desde módulo. As demais regras podem ser encontradas no site da ferramenta [32]. Ele está organizado da seguinte forma:

- Linha 1: identifica o módulo dessa transformação, mostrando que trata-se da identificação dos padrões;
- Linha 2: identifica que o modelo de entrada segue o meta-modelo da UML e que o modelo de saída segue o meta-modelo *ApplicableTestPatterns* (TP);
- Linha 4: mostra o nome da regra responsável por verificar se o padrão *Round-trip Scenario Test* é aplicável;
- Linhas 5-6: mostra que essa *matched rule* será executada para o elemento *Collaboration* do modelo UML origem que possui a classe a ser integrada (*ClassToBeIntegrated*), de acordo com a especificação do padrão mostrada no Capítulo 3;
- Linhas 7-14: cria os elementos *TestPattern* e *Element* do meta-modelo TP no modelo destino, que correspondem, respectivamente, ao nome padrão de teste que a regra identifica e qual o elemento do modelo UML origem possibilitou essa identificação.

Código Fonte 4.2: Regras ATL do módulo de identificação da aplicação de padrões de teste.

```
1  module PatternApplicationIdentificator;
2  create OUT : TP from IN : UML2;
3  rule RoundTripScenarioTest{
4   from o:UML2!Collaboration(o.ownedBehavior->first().ownedAttribute->exists(att |
5          att.type.getAppliedStereotypes()->exists(a | a.name = 'ClassToBeIntegrated')))
6   to testPattern: TP!TestPattern(
7       name <- 'Round Trip Scenario Test',
8       involvedElement <- Set{involvedElement}
9      ),
10     involvedElement: TP!Element(
11      name <- o.name,
12      type <- o.oclType().toString()
```

```
13         )
14    }
```

4.4 Módulo de Geração de Casos de Teste

Este módulo é responsável por gerar os casos de teste independentes de plataforma de acordo com as estratégias dos padrões de teste documentados. A ideia é que, uma vez identificada uma possível aplicação de um padrão, modelos de teste independentes de plataforma sejam desenvolvidos, e que casos de teste, documentados de acordo com U2TP, sejam gerados. A Figura 4.6 mostra a abordagem MDA adotada para esse módulo, onde estão representados os seguintes artefatos:

- **Meta-modelos:** este módulo gera modelos de teste (arquitetura e casos de teste) independentes de plataforma a partir dos modelos UML no nível PIM-PITM. Nesse caso, o meta-modelo adotado tanto para a origem quanto para o destino é o da UML 2.0, provido pela OMG;
- **Modelos:** os modelos de entrada são os diagramas de classe e sequência UML contendo o projeto estrutural e comportamental do software e os modelos de saída são diagramas de classe e sequência UML representando a arquitetura de teste e os casos de teste documentados de acordo com o perfil U2TP;
- **Transformações:** as transformações são especificadas na linguagem ATL.

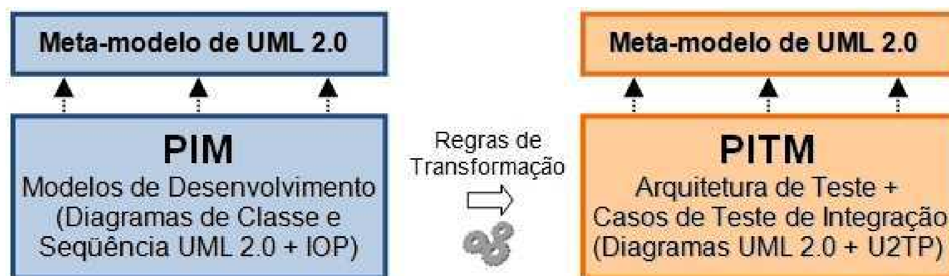


Figura 4.6: Abordagem MDA do módulo de geração de casos de teste.

Transformações em ATL

O conjunto de regras de transformação ATL especificadas para esse módulo é composto por quatro arquivos ATL, cada um especificando regras de transformação para a geração dos modelos de teste de acordo com um dos padrões de teste documentados. O nome de cada arquivo ATL é o nome do respectivo padrão que ele representa. Cada arquivo é composto por uma regra principal do tipo *matched rule*, que: (i) identifica os elementos UML presentes no modelo PIM (cláusula *from*) que permitem a aplicação do padrão teste (da mesma forma que no módulo anterior); (ii) gera os artefatos de teste no modelo PITM (cláusula *to*) conforme a estratégia definida pelo padrão e documentados de acordo com o meta-modelo de UML 2.0 e com o perfil de teste U2TP. No bloco *do* de cada regra, outras instruções imperativas importantes, como a aplicação dos estereótipos de U2TP aos elementos e a criação de outros artefatos de teste são declaradas, ou outras regras do tipo *called rule* especificadas no arquivo são invocadas.

Por exemplo, para o padrão a aplicação do *Round-Trip Scenario Test*, as principais regras ATL, em linhas gerais, são:

1. **Regra 1 - *roundTripScenarioTestAlgorithm*.** Esta é a regra principal e consiste numa *matched rule* cujo objetivo é a identificação dos elementos do meta-modelo de UML presentes no modelo PIM (cláusula *from*) que permitem a aplicação do padrão. No bloco *do* desta regra, são feitas as chamadas às demais regras que compõem a estratégia do padrão para a geração dos casos de teste. Semanticamente, para todo diagrama de sequência identificado no *from*, a regra aplica o padrão e gera os casos de teste;

2. **Regra 2 - *selectPaths*.** Esta regra é uma *called rule* e é chamada pelo bloco imperativo *do* da regra principal *roundTripScenarioTestAlgorithm*. Esta regra consiste na chamada das regras que implementam a estratégia para a geração dos casos de teste: (i) a regra que cria o grafo de fluxo de controle (*createGraph*); (ii) a regra que cria os caminhos a partir do grafo de fluxo (*composeAllPaths*); e (iii) a regra que gera um caso de teste para cada caminho criado (*createTestCaseForPath*);

3. **Regra 3 - *createGraph*.** Esta regra é uma *called rule* e representa o algoritmo que transforma um diagrama de sequência em um grafo de fluxo. Basicamente, este

algoritmo consiste em um mapeamento dos elementos do diagrama de sequência pra o grafo de fluxo, de acordo com os seguintes aspectos: as mensagens (interações do tipo *message* e seus componentes) são mapeadas para uma estrutura que representa um retângulo e as condições dos fragmentos combinados (*alt*, *opt*, *loop*, etc.) são mapeadas para uma estrutura que representa um hexágono no grafo de fluxo;

4. **Regra 4 - *composeAllPaths*.** Esta regra é uma *called rule* e representa o algoritmo que identifica e gera caminhos a partir do grafo de fluxo. Cada caminho gerado é equivalente a um cenário para a geração de um caso de teste. Este algoritmo é implementado seguindo o método de busca em profundidade (DFS - *Depth-First Search*) no grafo. Intuitivamente, este algoritmo começa no nó raiz (o primeiro nó da estrutura de dados que representa o grafo) e explora tanto quanto possível cada um dos seus ramos, antes de retroceder (*backtracking*). No caso do grafo de fluxo em questão, cada nó representa uma mensagem (ou sequência de mensagens) ou um fragmento combinado do diagrama de sequência. Este algoritmo segue o critério de cobertura de caminhos, ou seja, todos os possíveis caminhos são gerados.

5. **Regras 5 - *createTestCaseForPath*.** Esta regra é uma *called rule* e representa a geração de um caso de teste de acordo com o perfil de teste U2TP para cada caminho gerado pela regra *composeAllPaths*. Além disso, é nessa regra que todos os artefatos da arquitetura de testes também são gerados. Basicamente, neste algoritmo é feito o mapeamento inverso entre os elementos do grafo de fluxo e os elementos do diagrama de sequência, uma vez que os casos de teste de acordo com U2TP são representados por diagramas de interação, que no caso da abordagem proposta são diagramas de sequência.

O código 4.3 ilustra trechos que compõem as duas primeiras regras para o padrão *Round-trip Scenario Test*: a regra *roundTripScenarioTestAlgorithm* e a regra *selectPaths*. As demais regras podem ser encontradas no site da ferramenta [32]. O código 4.3 está organizado da seguinte forma:

- Linha 1: identifica o módulo dessas transformações, mostrando que trata-se da aplicação do padrão *Round-trip Scenario Test*;

- Linha 2: identifica que os modelos de entrada e saída seguem o meta-modelo da UML. Além disso, o modelo de entrada segue também o perfil IOP; ;
- Linha 4: mostra o nome da regra *roundTripScenarioTestAlgorithm* responsável por aplicar o padrão;
- Linhas 5-6: mostra que a regra *roundTripScenarioTestAlgorithm* será executada para o elemento *Collaboration* do modelo UML origem, de acordo com a especificação do padrão mostrada no Capítulo 3;
- Linha 7: cria o elemento *Collaboration* do meta-modelo de UML no modelo destino;
- Linhas 8-12: é a parte imperativa da regra *roundTripScenarioTestAlgorithm*, é nesse trecho onde são chamadas a regra responsável pela a criação do contexto de teste (linha 9) de acordo com U2TP e a regra para que aplica o algoritmo para geração dos casos de teste (linha 11);
- Linhas 15-26: mostra a regra *selectPaths*, a qual consiste na chamada das *matched rules* que implementam a estratégia para a geração dos casos de teste: (i) a regra *createGraph* que cria o grafo de fluxo de controle (linha 18); (ii) a regra *composeAllPaths* que cria os caminhos a partir do grafo de fluxo (linha 20); e (iii) a regra *createTestCaseForPath* que gera um caso de teste para cada caminho criado (linha 25).

Código Fonte 4.3: Regras ATL do módulo de geração de casos de teste.

```
1  module RoundTripScenarioTest; -- In TestCaseGenerator Module
2  create OUT : UML2 from IN : UML2, U2TP : UML2, IOP : UML2;
3  rule roundTripScenarioTestAlgorithm {
4    from o:UML2!Collaboration(o.ownedBehavior->first().ownedAttribute->exists(att |
5           att.type.getAppliedStereotypes()->exists(a | a.name = 'ClassToBeIntegrated')))
6   to col:UML2!Collaboration ()
7   do {
8    if (thisModule.testContext = '')
9     self.createTestContext('RTSTPatternContext');
10   thisModule.selectPaths(c, thisModule.testContext);
11  }
12 }
13 rule selectPaths(s : UML2!Collaboration , testContext : UML2!Class){
14  do {
```

```
15    thisModule.firstFather <- s.ownedBehavior->first();
16    self.createGraph(s.ownedBehavior->first());
17    if (thisModule.root.notEmpty())
18      self.composeAllPaths(thisModule.firstFather);
19    else thisModule.paths <- thisModule.paths->including(thisModule.path);
20    self.validatePaths();
21    for (path in thisModule.paths){
22      thisModule.nTestCase <- thisModule.nTestCase+1;
23      self.createTestCaseForPath(path, thisModule.nTestCase, testContext);
24    }
25  }
26 }
```

4.5 Considerações Finais

Neste Capítulo foi mostrada a forma como a ferramenta de suporte automático à abordagem proposta nesse trabalho foi desenvolvida. Esta ferramenta, a qual foi desenvolvida seguindo os princípios de MDA, consiste em três módulos contendo meta-modelos, modelos e regras de transformação especificadas com a linguagem ATL. São eles: (i) módulo de filtragem dos modelos; (ii) módulo de identificação da aplicação de padrões de teste; e (iii) módulo de geração de casos de teste. Cada um desses módulos dá suporte às três atividades da abordagem proposta, respectivamente.

O modo de interação atual do usuário com a ferramenta é feito através do *framework* ATL-DT [5], que é um *plug-in* integrado com a IDE Eclipse. Em linhas gerais, para a execução da ferramenta é preciso que os arquivos ATL correspondentes a cada um dos módulos sejam executados pela IDE Eclipse, com as devidas configurações (meta-modelos de entrada e saída, perfis UML e modelos de entrada e saída). A Figura 4.7 mostra as configurações para a execução *RoundTripScenarioTest.atl* correspondente ao arquivo ATL do módulo para a geração de casos de teste de acordo com o padrão *Round-trip Scenario Test*. Nesta Figura é possível perceber os seguintes artefatos: (1) o meta-modelo de entrada e saída (*UML2*); (2) o modelo de entrada (*IN*) e os perfis U2TP e IOP que seguem o meta-modelo UML2; e (3) o local onde serão gerados os modelos de teste (*OUT*) que também segue o meta-modelo UML2.

Para facilitar o uso da ferramenta, uma forma de interação por linha de comando, ou

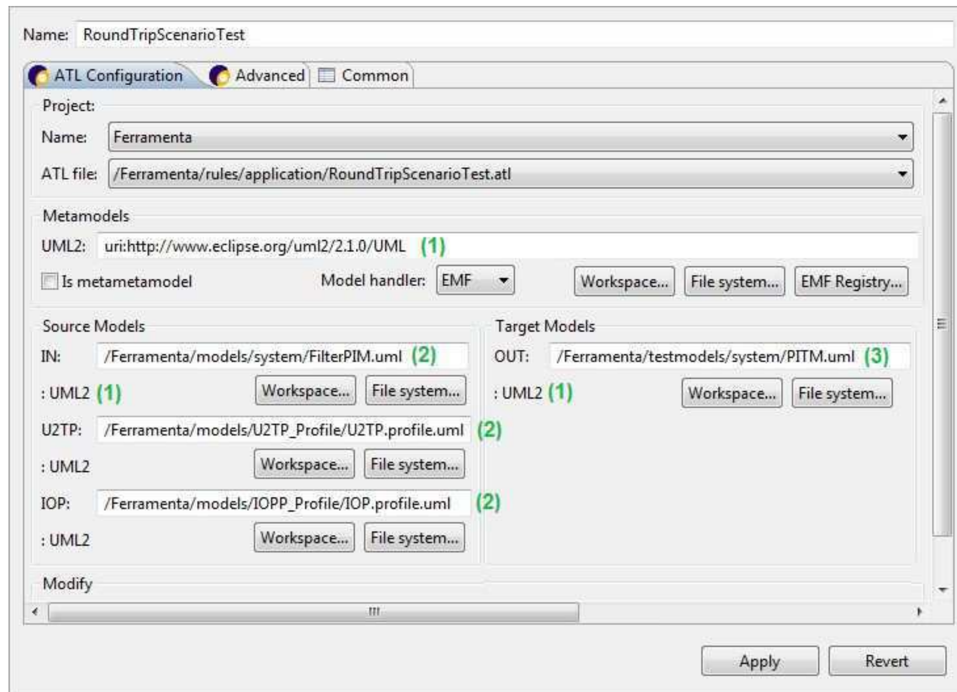


Figura 4.7: Configurações ATL para a execução terceiro módulo para o padrão *Round-trip Scenario Test*.

via interface gráfica, está sendo projetada. O intuito é que o mínimo de informação para execução da ferramenta seja necessário.

Adicionalmente, a ferramenta possui uma licença GPL (*General Public License*), e está disponível em [32] para toda a comunidade, de forma que qualquer outra pessoa pode ter acesso para uso no contexto de teste de integração com MDT ou estendê-la com a implementação de novos padrões de teste. Para essa extensão, é preciso que:

1. Os elementos que identificam a aplicação do padrão que se deseja incorporar à ferramenta em nível de modelos sejam identificados e documentados de acordo com o meta-modelo *TestPatternMetamodel*;

2. Uma *matched rule*, cujo nome deve corresponder ao nome do respectivo padrão que ela identifica, deve ser criada no arquivo ATL correspondente ao segundo módulo da ferramenta. A cláusula *from* desta regra corresponderá à *patternConstraint* especificada para o padrão no meta-modelo *TestPatternMetamodel* e aplicada ao elemento *in*;

3. Um módulo ATL para a implementação da estratégia do padrão. Este módulo deve

conter pelo menos uma *matched rule*, que será a regra principal que: (i) identificará os elementos UML presentes no modelo PIM (cláusula *from*) que permitem a aplicação do padrão teste (da mesma forma que no item anterior); (ii) gerará os artefatos de teste no modelo PITM (cláusula *to*) conforme a estratégia definida pelo padrão e documentados de acordo com o meta-modelo de UML 2.0 e com o perfil de teste U2TP. No bloco *do* da regra, outras instruções imperativas importantes, como a aplicação dos estereótipos de U2TP aos elementos e a criação de outros artefatos de teste devem ser declaradas, ou outras regras do tipo *called rule* especificadas no arquivo podem ser invocadas.

Para a incorporação de novos padrões à ferramenta, a pessoa deve além de saber como implementar o algoritmo da estratégia do padrão de teste, ela deve principalmente conhecer bem a linguagem ATL para que a aplicação do padrão seja feita corretamente. No entanto, como MDA fornece padrões para a realização de transformações entre vários níveis de abstração, uma forma de gerar automaticamente as regras ATL que aplicam os padrões a partir dos modelos documentados de acordo com o meta-modelo *TestPatternMetamodel* estendido será investigada, para minimizar os esforços na extensão da ferramenta, e consequentemente, da abordagem proposta.

É importante ressaltar a ferramenta foi desenvolvida para o nível de abstração PIM-PITM. Para a geração dos casos de teste executáveis, é preciso ainda implementar transformações entre os níveis de abstração PITM-PSTM e PSTM-Código, onde os casos de teste poderão ser gerados para uma plataforma específica, de acordo com os objetivos de teste definidos na fase de planejamento de testes do processo de desenvolvimento. Após isso, é que os casos de poderão ser selecionados para serem executados efetivamente com dados de teste. Com relação aos dados de teste, eles serão gerados semi-automaticamente, de acordo com as pré-condições dos casos de teste gerados no nível PITM.

Capítulo 5

Avaliação Experimental

Este Capítulo apresenta os estudos experimentais realizados para avaliar a abordagem proposta neste trabalho e a ferramenta de suporte desenvolvida. Nesse contexto, foi adotado o método *Goal/Question/Metric* (GQM) [9] para a definição e a análise de objetivos, questões e métricas para esta avaliação utilizando estudos de casos. O Capítulo está organizado da seguinte forma. A Seção 5.1 apresenta uma breve introdução ao método de avaliação utilizado. As Seções 5.2 e 5.3 apresentam os procedimentos realizados na avaliação da abordagem proposta e da ferramenta de suporte, baseados no método escolhido, respectivamente. Por fim, a Seção 5.4 apresenta as conclusões sobre a avaliação realizada.

5.1 Método de Avaliação Experimental Utilizado

Vários métodos foram elaborados com o intuito de organizar e controlar experimentos. Um bom exemplo de método de experimentação é o paradigma *Goal/Question/Metric* (GQM) [9], uma abordagem *top-down* que estabelece um modelo de medição de software dirigido por objetivos. Especificamente, este paradigma define um modelo de avaliação em três níveis (conceitual, operacional e lógico) que engloba a definição de objetivos (*goal*) de medição, a formulação de questões (*question*) que caracterizam os objetivos, e a identificação de métricas (*metric*) que provêem respostas para as questões, conforme mostrado na Figura 5.1.

A abordagem GQM compreende quatro fases: (i) definição, onde o experimento é expresso em termos de objetivos, questões e métricas; (ii) planejamento, onde o projeto do experimento é determinado, o contexto é selecionado, as hipóteses são formuladas e as

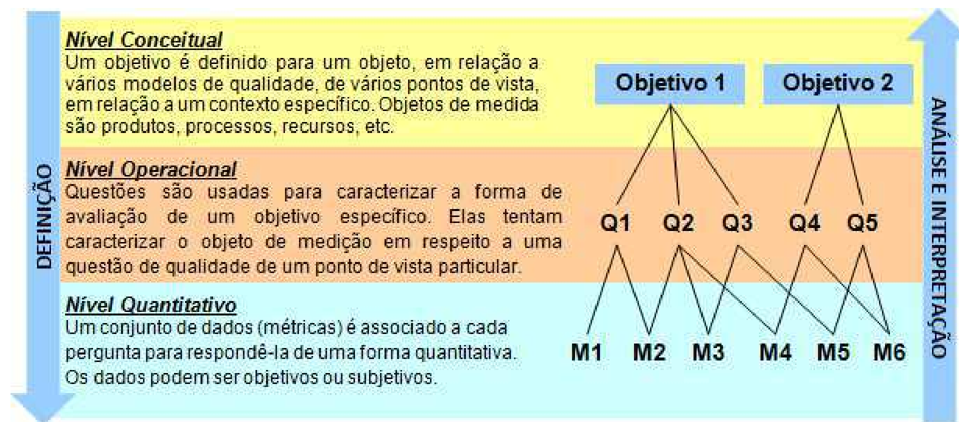


Figura 5.1: O paradigma *Goal/Question/Metric* - GQM (adaptado de [9]).

variáveis são selecionadas; (iii) execução e resultados, onde a execução do experimento é realizada e os dados são coletados resultando em um conjunto de dados pronto para a análise; e (iv) interpretação, onde os dados coletados são processados com relação às métricas, questões e objetivos definidos, e, finalmente, os resultados são apresentados.

O mecanismo utilizado para a realização desta avaliação foi estudo de caso. Esta escolha teve como base os seguintes fatores: (i) o tipo de avaliação que se deseja realizar; (ii) as características desse tipo de estudo (fácil de planejar e de baixo custo); e (iii) a falta de controle necessário sobre diversas variáveis que compõem o contexto em que os estudos foram executados o que impossibilitaria a realização de experimentos formais, por exemplo.

5.2 Estudo Experimental para Avaliação da Abordagem

Nesta Seção, é detalhado todo o procedimento de avaliação da abordagem realizado seguindo, sequencialmente, as quatro fases envolvidas pelo método GQM: Definição, Planejamento, Execução (e Resultados) e Interpretação.

5.2.1 Definição

O objetivo de medição nesse estudo tem a finalidade de:

> Analisar o processo de geração de casos de teste a partir de modelos de desenvolvimento usando padrões de teste (**objeto**), com o propósito de redução do esforço gasto (**propósito**), com respeito à validade e eficácia dos casos de

teste (**foco de qualidade**), do ponto de vista do testador (**perspectiva**), e no contexto de teste de integração com MDT (**contexto**).

Para essa análise, é feita uma comparação entre a utilização da abordagem proposta nesse trabalho de forma automática e de forma manual. Esta decisão foi tomada uma vez que um dos objetivos da abordagem é automatizar o uso de padrões de teste para a geração de casos de teste, o qual, atualmente, ainda é bastante manual. Além disso, outros dois fatores motivaram essa escolha: (1) não foi encontrada nenhuma abordagem semelhante que servisse como instrumento para a comparação; e (2) a não disponibilização de testadores experientes em teste de integração e padrões de teste, para que os resultados fossem comparados adequadamente. Também não se optou avaliar os padrões de teste, pois são padrões já presentes na literatura, logo, já são conceitos importantes e aceitos pela comunidade de teste de software.

Uma vez definido o objetivo, a etapa seguinte, que corresponde à parte operacional da elaboração do modelo de medição, foi feito o levantamento das questões que refinam e caracterizam o objetivo. Dado que o interesse é avaliar o processo automático de geração de casos de teste usando padrões de teste em comparação ao processo manual, três questões a serem levantadas são estão relacionadas aos critérios **esforço**, **validade** e **eficácia**. As questões levantadas para esse objetivo, de acordo com esses critérios, bem como as métricas especificadas para quantificar as questões são: quais têm por objetivo quantificá-las, são:

- **Esforço – Qual o impacto em relação ao esforço despendido pelos testadores?** Em teste de software, uma das grandes preocupações é com a redução do esforço do testador. Este esforço pode ser medido com relação ao tempo despendido para a realização da atividade e com relação à quantidade de casos de teste gerados. O esforço com relação ao tempo E_t é definido como:

$$E_t = T_p + T_e, \tag{5.1}$$

onde T_p é o tempo de preparo, e T_e é o tempo de execução dos testes [35]. Em abordagens MBT (e suas combinações), T_p pode ser compreendido como o tempo de construção do modelo a partir dos requisitos e a geração dos casos de teste. Como o

foco da abordagem alvo de avaliação é apenas na geração dos casos de teste, o tempo T_e não precisa ser considerado, pois os casos de teste não são executados. Logo, a métrica pode ser simplificada em:

$$E_t = T_p, \tag{5.2}$$

ou seja, o esforço com relação ao tempo é definido como o tempo para a construção do modelo e geração dos casos de teste. Em abordagens de teste de software para medir o esforço do testador é importante, além do tempo, considerar também a quantidade de casos de teste gerados. Nesse caso, o esforço com relação à quantidade de casos de teste E_{nct} é definido como:

$$E_{nct} = N_{ct}, \tag{5.3}$$

onde, N_{ct} é o número de casos de teste gerados. Fazendo uma relação entre as duas métricas acima, pode-se medir também o esforço com relação ao tempo gasto pelo testador para a geração de um caso de teste. Essa métrica é importante porque o tempo está diretamente relacionado ao número de casos de teste gerados. Em abordagens de teste de software, um tempo muito grande pode estar aparentemente relacionado a geração de muitos (ou poucos) casos de teste, dependendo da experiência e eficiência do testador. Ao mesmo tempo, um testador pode produzir muitos casos de teste em um tempo curto. Em linhas gerais, o esforço E com relação ao tempo médio por caso de teste gerado é definido como:

$$E = \frac{E_t}{E_{nct}}, \tag{5.4}$$

onde, E_t é o esforço gasto com relação ao tempo e E_{nct} é o esforço gasto com relação ao número de casos de teste gerados.

- **Validade – Qual o número de casos de teste válidos gerados?** Em abordagens MBT, é importante verificar se o testador foi eficiente na abordagem de teste utilizada, ou seja, é importante saber, dentre os casos de teste gerados, qual a porcentagem de casos

de teste que poderiam realmente ser considerados válidos. Neste caso, a validade pode ser medida como:

$$V = 1 - \frac{N_{cti}}{N_{ct}}, \tag{5.5}$$

onde, N_{ct} é o número de casos de teste gerados e N_{cti} é o número de casos de teste considerados inválidos. No contexto da abordagem, um caso de teste é considerado inválido quando: (i) é sintaticamente incorreto, de acordo com a modelagem/especificação do sistema; (ii) é semanticamente inconsistente com a aplicação, ou seja, denota um comportamento de sucesso ou falha que não condiz com o esperado pelos modelos; (iii) não pode ser executado, por estar incompletamente definido, ou por não cobrir a especificação do padrão; ou (iv) não representa o cenário de integração proposto. Nas demais situações, os casos de teste são considerados válidos. Alguns podem não ser os melhores, mas a decisão de utilizá-los, ou não, é do testador.

- **Eficácia – Qual a eficácia dos casos de teste gerados em relação à capacidade de detecção de defeitos importantes?** No contexto de teste de software, a eficácia de uma estratégia está diretamente associada à capacidade detecção de defeitos, ou seja, os casos de teste são considerados eficazes se realmente são capazes de encontrar defeitos no software. Em relação a sistemas OO, casos de teste de integração podem revelar vários tipos de defeitos importantes que podem surgir quando classes que se relacionam entre si são integradas, tais como [11]:

  1. Falta, sobreposição ou conflito de funcionalidades;
  2. Tratamento de erros (exceções) incorreto;
  3. Chamadas a funções erradas devido a erro no código ou exceções inesperadas;
  4. Mensagens enviadas de uma classe cliente que violam pré-condições da classe servidora;
  5. Mensagens enviadas de uma classe cliente que violam restrições seqüenciais da classe servidora;
  6. Objetos incorretos associados a mensagens;

7. Parâmetros errados ou valores de parâmetros incorretos;
8. Conflitos entre componentes (por exemplo, colisão de *threads*);
9. Bloqueio de funções/tarefas.

A maioria dos padrões de teste, incluindo os considerados nesse trabalho, têm um modelo de faltas associado. Esse modelo de faltas descreve os possíveis tipos de defeitos que poderiam ser encontrados pelos casos de teste gerados de acordo com a estratégia do padrão. Como a abordagem avaliada considera apenas a geração dos casos de teste e não a execução, uma vez que não existe código executável (tanto da aplicação como de teste), e os casos de teste são gerados por padrão, ficou resolvido fazer a análise da capacidade de detecção de defeitos da seguinte forma:

$$C_d = \frac{N_d}{E_{nct}}, \tag{5.6}$$

onde N_d é o número de tipos de defeitos que poderiam ser detectados (de acordo com os tipos listados anteriormente), se existissem, quando os testes fossem executados, e E_{nct} é o esforço requerido com relação ao número de casos de teste gerados. Com relação aos casos de teste gerados pelos padrões de teste, de acordo com o modelo de faltas de cada padrão, foi feito um mapeamento dos possíveis tipos de defeitos de integração, dentre os nove acima listados, que poderiam ser detectados pelos casos de teste gerados de acordo cada um dos padrões de teste, conforme resumido na Tabela 5.1. Mais detalhes sobre esse mapeamento podem ser encontrados no Apêndice D.

Tabela 5.1: Tipos de defeitos de integração que poderiam ser detectados pelos padrões de teste.

Padrão de Teste	**Tipo de Defeito**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Round-trip Scenario Test	x		x	x	x	x	x		
Abstract Class Test	x		x	x	x	x	x		
Error Simulator		x							
Test Case With Time Specification								x	x

5.2.2 Planejamento

O planejamento deste estudo de caso consiste na seleção do contexto (ambiente e participantes), na formulação das hipóteses, na seleção das variáveis e na preparação conceitual do projeto do estudo.

Seleção do Contexto (Ambiente e Participantes)

O contexto escolhido para a execução do estudo foi um contexto não-industrial, onde estudantes de graduação e pós-graduação do Curso de Ciência da Computação (CCC) da Universidade Federal de Campina Grande (UFCG), com experiência na área, atuam como sujeitos (participantes) do estudo experimental. O objetivo desta escolha foi diminuir o custo do estudo, uma vez que a aplicação dele em contextos industriais, incluindo a contratação de profissionais reais, não condizia com os recursos disponíveis para a sua realização.

Com relação aos participantes, não há diferenças substanciais com relação à formação deles, pelo fato de serem oriundos da mesma instituição de ensino. Contudo, para a realização do estudo era fundamental que todos tivessem conhecimento em teste de software (área de concentração da abordagem). No entanto, como são utilizadas várias tecnologias/conceitos de teste de software na abordagem, existem alguns fatores de variação na experiência dos participantes que podem implicar nos resultados do estudo e precisam ser considerados, são eles:

- Conhecimento na notação utilizada para a descrição dos modelos (PIM) de entrada, no caso, a linguagem UML, na sua versão 2.0 (ou superior);
- Conhecimento em estratégias de teste de integração;
- Conhecimento em padrões de teste;
- Conhecimento no perfil de teste da UML 2.0 (U2TP).

Formulação das Hipóteses

No contexto deste estudo de caso para avaliação da abordagem para a geração de casos de teste usando padrões de teste, dentro do contexto de teste de integração com MDT, foram definidas hipóteses de estudo para cada uma das questões formuladas na seção 5.2.1. Estas

hipóteses foram avaliadas por meio da avaliação dos dados coletados durante a execução do estudo de caso.

As três primeiras hipóteses referem-se à questão do esforço gasto pelo testador na utilização da abordagem automática em comparação à abordagem manual. Este esforço é medido em relação ao tempo gasto pelo testador para a geração dos casos de teste, ao número de casos de teste gerados e o tempo gasto para a geração de um caso de teste. A quarta hipótese se refere à corretude dos casos de teste gerados, ou seja, o percentual de casos de teste válidos gerados. A quinta hipótese de estudo refere-se à eficácia dos casos de teste, ou seja, à capacidade de detecção de defeitos dos casos de teste gerados na abordagem automática em comparação à abordagem manual. Detalhando cada uma das hipóteses, tem-se:

- **Hipótese 1:** Utilizando a abordagem automática, o esforço com relação ao tempo gasto é menor do que utilizando a abordagem de forma manual, ou seja,

  μE_t(Automática) < μE_t(Manual), onde μE_t corresponde à média do esforço dos testadores com relação ao tempo gasto pelos testadores.

- **Hipótese 2:** Utilizando a abordagem automática, o esforço com relação ao número de casos de teste válidos é maior do que utilizando a abordagem de forma manual, ou seja,

  μE_{nct}(Automática) > μE_{nct}(Manual), onde μE_{nct} é a média do esforço dos testadores com relação ao número de casos de teste gerados.

- **Hipótese 3:** Utilizando a abordagem automática, o esforço com relação a taxa de geração de um caso de teste é menor do que utilizando a abordagem de forma manual, ou seja,

  μE(Automática) < μE(Manual), onde μE é a média do esforço dos testadores com relação ao tempo gasto para a geração de um caso de teste.

- **Hipótese 4:** Utilizando a abordagem automática, porcentagem de casos de teste válidos é maior do que utilizando a abordagem de forma manual, ou seja,

μV(Automática) > μV(Manual), onde μV é a média da porcentagem de casos de teste corretos/válidos.

- **Hipótese 5:** Utilizando a abordagem automática, a capacidade de detecção de defeitos relevantes dos casos de teste é maior do que utilizando a abordagem de forma manual, ou seja,

  μC_d(Automática) > μC_d(Manual), onde μC_d é a média da capacidade de detecção de defeitos dos casos de teste gerados.

Seleção das Variáveis

No contexto de avaliação experimental, há dois tipos de variáveis: independentes e dependentes. Tais variáveis, são, respectivamente, a entrada e a saída do processo de experimentação. As variáveis independentes também são chamadas de fatores e apresentam a causa que afeta o resultado do processo de experimentação. Já as variáveis dependentes apresentam o efeito que é causado pelos fatores. As variáveis independentes serão controladas e executadas pelo estudo de caso, para que as variáveis dependentes sejam obtidas. Diante disto foram definidas as seguintes variáveis:

- **Variáveis Independentes:** (i) o documento com a descrição da abordagem (artefatos, atividades, etc); (ii) os modelos de desenvolvimento de entrada do processo de geração dos casos de teste para os sistemas definidos; (iii) a ordem de integração dos componentes (classes) do sistema, incluindo o que será integrado e suas dependências; (iv) o catálogo com os padrões de teste (apenas para a abordagem manual); e (v) a ferramenta de suporte desenvolvida (apenas para a abordagem automática).

- **Variáveis Dependentes:** (i) os artefatos gerados (arquitetura, casos de teste, etc.); e (ii) os resultados das funções para obtenção dos valores das métricas.

Princípios do Projeto do Estudo de Caso

De acordo com os princípios gerais de projeto *aleatoriedade*, *blocagem* e *balanceamento* [50], ficou definido que:

- Não há *aleatoriedade* na atribuição do objeto de estudo aos participantes, visto que todos os participantes irão utilizar o mesmo objeto. Os participantes não foram selecionados aleatoriamente da população, pois são provenientes do curso de Ciência da Computação da UFCG disponíveis para a realização do estudo de caso. A ordenação dos participantes no estudo de caso não é importante, uma vez que as medidas usadas na avaliação são derivadas da solução do problema proposto;

- Não foi utilizada estratégia de *blocagem* uma vez que a amostra de participantes é pequena e proveniente do mesmo curso de Ciência da Computação da UFCG, portanto, os níveis de conhecimento não são tão discrepantes;

- Como a amostra de participantes é proveniente do mesmo curso, não foi possível garantir o *balanceamento* do conjunto de dados do estudo de acordo com a qualificação individual dos participantes.

5.2.3 Execução e Resultados

A fase de execução e resultados tem como objetivo a aplicação do modelo de medição definido e planejado nas fases anteriores. Esta aplicação consiste em coletar os dados necessários para o cálculo das métricas que serão analisadas na fase de interpretação. A seguir são detalhados a definição da execução e os resultados obtidos para o estudo de caso.

Definição da Execução

Para a realização do estudo de caso, foram selecionados três participantes dentro do contexto definido e levando em consideração os fatores de variação listados na fase de planejamento. Esta seleção foi realizada com base em entrevistas, cujo objetivo era identificar o conhecimento (experiência) nas tecnologias envolvidas. A Tabela 5.2 resume o perfil de cada participante. O grau de conhecimento dos participantes foi classificado em: (1) não tem conhecimento algum sobre o assunto; (2) já ouviu falar, mas nunca trabalhou com o assunto; (3) tem conhecimento, mas não tem muita familiaridade; (4) tem conhecimento e experiência moderada no assunto; e (5) domina completamente o assunto.

Tabela 5.2: Perfil dos participantes do estudo de caso.

	Participante 1	**Participante 2**	**Participante 3**
Conhecimento em UML (2.0 ou superior)	4	4	5
Conhecimento em teste de integração	2	3	5
Conhecimento em padrões de teste	2	3	5
Conhecimento em U2TP	2	4	5

O estudo de caso foi conduzido da seguinte forma: os dois primeiros participantes foram selecionados para realizar a geração de casos de teste de acordo com a abordagem de forma manual e o último para realizar a geração dos casos de teste de acordo com a abordagem de forma automática. A escolha de mais de um participante para realizar a abordagem manual foi tomada porque em abordagens de teste manuais os casos de teste são diretamente influenciados por fatores, além dos apresentados na Tabela 5.2, pessoais do testador como a disponibilidade, a concentração, o humor, entre outros. Para estes dois participantes foram fornecidos os modelos de desenvolvimento para o sistema sob teste definido, alguns cenários de integração do sistema, incluindo o que devia ser integrado e suas dependências, e o catálogo com os padrões de teste documentados. Para o terceiro participante, o qual foi selecionado para realizar a geração dos casos de teste de acordo com a abordagem automática, foram fornecidos os mesmos modelos e cenários disponibilizados para os outros participantes, e a ferramenta de suporte.

Para todos os participantes foi realizado um pequeno treinamento, no qual foi explicado como deveria ser realizado o processo (tanto manual como automático) e um documento detalhado descrevendo a abordagem foi disponibilizado para eventuais dúvidas que pudessem surgir durante o estudo de caso. Em particular, para o participante que realizou a abordagem automática foi mostrado como a ferramenta de suporte é utilizada e executada, incluindo as ferramentas auxiliares necessárias para sua execução. Neste estudo de caso, os participantes foram instruídos a realizar as seguintes atividades (conforme mostrado na Figura 5.2):

- **Abordagem manual:** (i) analisar os modelos de desenvolvimento; (ii) analisar a ordem de integração descrita pelos cenários de integração e aplicar o perfil IOP aos modelos; (iii) filtrar os modelos no intuito de que apenas as classes importantes para

o teste de integração fossem consideradas; (iv) analisar o catálogo de padrões de teste, atividade que pode ser realizada em paralelo às três anteriores; (v) identificar possíveis padrões de teste que podem ser aplicados aos modelos para a geração dos casos de teste; e (vi) caso existissem padrões a serem aplicados, gerar os casos de teste documentados de acordo com U2TP e seguindo as estratégias dos padrões;

- **Abordagem automática:** (i) analisar os modelos de desenvolvimento; (ii) analisar a ordem de integração descrita pelos cenários de integração e aplicar o perfil IOP aos modelos; e (iv) executar a ferramenta.

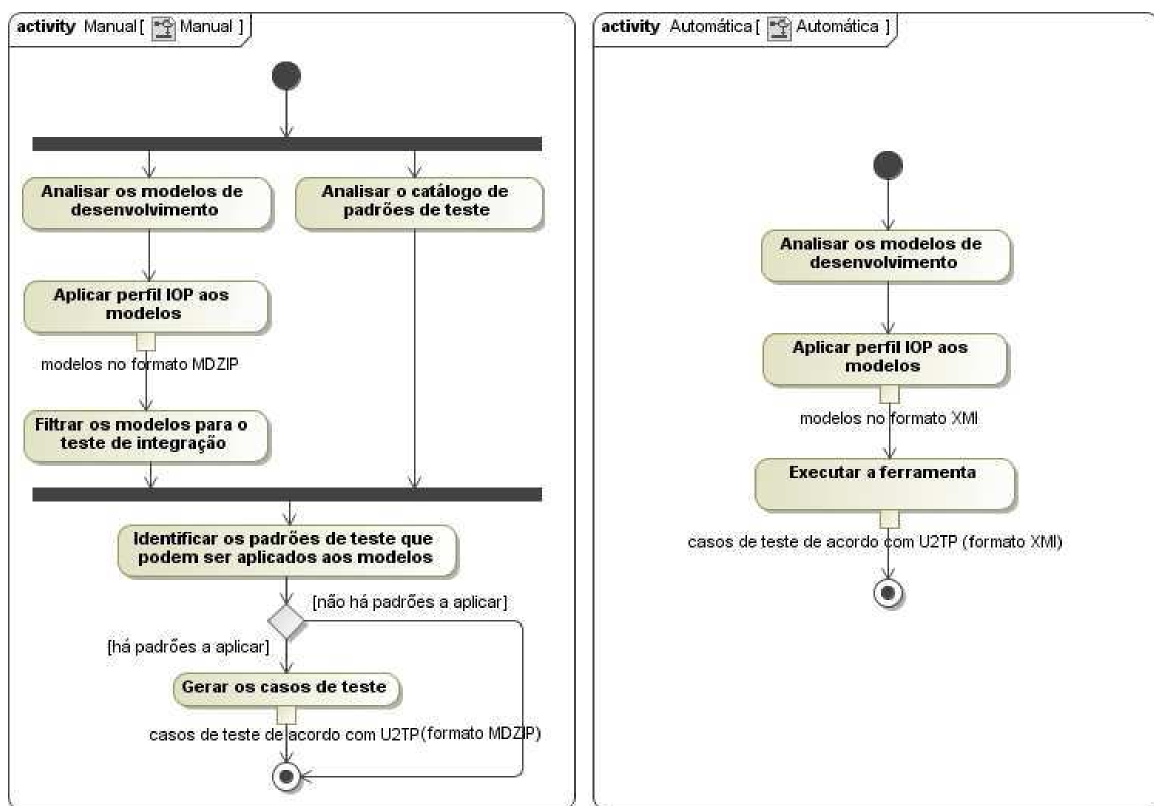


Figura 5.2: Atividades realizadas pelos participantes durante o estudo experimental.

Os modelos de desenvolvimento, especificamente o diagrama de classes e os diagramas de sequência, foram construídos utilizando a ferramenta de modelagem MagicDraw [33]. Logo, foram disponibilizados aos participantes os modelos no formato aceitável por essa ferramenta (formato MDZIP), já com os perfis IOP e U2TP devidamente importados para que fossem aplicados aos modelos de desenvolvimento e de teste, respectivamente. Contudo, para a abordagem automática, estes modelos tiveram que ser convertidos para o formato

XMI (a ferramenta MagicDraw dá suporte a isso), que é um padrão para a representação de modelos UML. Isso foi necessário porque a ferramenta foi projetada para ser genérica e independente de ferramenta de modelagem. Conseqüentemente, os casos de teste gerados nessa abordagem também são representados no formato XMI.

Nesse estudo de caso, foi utilizado o Sistema *ATM System* como entrada do processo de geração de casos de teste. Este sistema consiste em um sistema de caixa eletrônico que permite ao usuário realizar transações financeiras básicas tais como saque e depósito de fundos. A modelagem deste sistema, compreende seis diagramas de sequência, cada um representando um caso de uso do sistema (comportamentos), e um diagrama de classes representando a estrutura do sistema. Para cada diagrama de sequência foram criados cenários de integração para os testes. A Tabela 5.3 apresenta detalhes sobre a modelagem desse sistema. Especificamente, é mostrado o número de classes, casos de uso, interações (troca de mensagens e condições nos diagramas de sequência) e cenários de integração construídos para os testes. No Apêndice B está documentada a especificação e todos os artefatos do *ATM System*.

Tabela 5.3: Informações sobre os artefatos do sistema *ATM System*.

	Nº de Artefatos
Classes	25
Casos de uso	6
Interações	248
Cenários de integração	22

Durante o estudo de caso, o tempo de realização de cada atividade foi monitorado e cronometrado no intuito de que dados fossem coletados para algumas das métricas anteriormente especificadas. Além disso, após a conclusão do estudo, foi pedido a cada participante o preenchimento do Questionário de Avaliação Pós-Estudo Experimental (ver Apêndice F), de forma a obter-se a sua avaliação qualitativa a respeito da aplicação da abordagem e dos procedimentos do estudo experimental. Ainda, entrevistas individuais foram conduzidas, onde cada participante foi incentivado a emitir opiniões a respeito da abordagem e da condução do estudo. Esta avaliação qualitativa servirá como *feedback* para o trabalho, para futuras melhorias.

Resultados Obtidos

A Tabela 5.4 mostra alguns dados obtidos após a execução do estudo. Nesta tabela são apresentados, para cada participante: (i) o tempo gasto, em minutos, na aplicação da abordagem para a geração dos casos de teste; (ii) o número total de casos de teste gerados; (iii) o número total de casos de teste considerados inválidos ou incorretos; e (iv) o número total de casos de teste considerados válidos ou corretos.

Tabela 5.4: Dados coletados por participante para o primeiro estudo de caso.

	Tempo Gasto	**CTs Gerados**	**CTs Inválidos**	**CTs Válidos**
Participante 1	544	65	19	46
Participante 2	1045	73	11	62
Participante 3	69	105	0	105

Nesse estudo, os participantes 1 e 2 realizaram a abordagem manual e o participante 3 realizou a abordagem automática. É importante ressaltar que para o participante 3, o tempo gasto referiu-se basicamente à aplicação do perfil IOP aos diagramas e à configuração da ferramenta. O tempo de geração dos casos de teste, neste caso, não foi medido pelo fato de ser insignificante (poucos segundos) e não atribuir valor à métrica.

Uma vez que em abordagens de geração de casos de teste medir apenas o número de casos de teste não é suficiente, após a seleção dos casos de teste válidos de cada participante, foi feita uma avaliação de cada um deles no intuito de analisar a sua eficácia com relação à capacidade de detecção de defeitos. A Tabela 5.5 mostra o número de defeitos que poderiam ser identificados, caso existissem, pelos casos de teste válidos de cada participante de acordo com a lista pré-definida de tipos de defeitos apresentada na Seção 5.2.1. É importante destacar que estes casos de teste não foram executados, uma vez que ainda não existe o código do sistema sob teste implementado, uma vez que o foco da abordagem é apenas na geração de casos de teste no processo de MDD/MDT. Portanto, os dados com relação a esse número de defeitos são apenas suposições dos defeitos que poderiam ser detectados, casos existissem.

Finalmente, a Tabela 5.6 resume os dados obtidos, por participante, para calcular os valores de todas as métricas propostas. Nesta tabela, T_p corresponde ao tempo total gasto

Tabela 5.5: Tipos de defeitos de integração detectados pelos casos de teste agrupados por participante e pelo tipo de defeito.

	Defeito por Tipo								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Participante 1	9	3	9	9	9	9	9	4	4
Participante 2	16	0	16	16	16	16	16	3	3
Participante 3	32	3	32	32	32	32	32	3	3

(em minutos) para a geração dos casos de teste, N_{ct} corresponde ao número total de casos de teste gerados, N_{cti} corresponde ao número total de casos de teste considerados inválidos dentre os que foram gerados e N_d corresponde ao número total de defeitos detectados pelos casos de teste.

Tabela 5.6: Resumo dos dados coletados por participante para obtenção dos valores das métricas.

	T_p	N_{ct}	N_{cti}	N_d
Participante 1	544	65	19	65
Participante 2	1045	73	11	102
Participante 3	69	105	0	203

No Apêndice D, é mostrado em detalhes como todos esses dados foram obtidos e analisados para cada um dos participantes que realizou o estudo.

5.2.4 Interpretação

Esta fase tem como objetivo fazer uma análise dos dados resultantes da fase de execução e resultados apresentada anteriormente com relação às hipóteses, métricas, questões e objetivos definidos. Para isso, foi realizada uma análise quantitativa desses dados de acordo com cada uma das questões levantadas e uma análise qualitativa a partir de questionários e entrevistas realizadas com os participantes do estudo de caso.

Análise Quantitativa dos Resultados

Para o cálculo das métricas, os dados apresentados na Tabela 5.6 da seção anterior foram agrupados por abordagem realizada, ou seja, foi calculada a média dos valores obtidos por participante em duas categorias: manual e automática. É importante ressaltar que, para este estudo de caso, a média para a categoria automática compreende aos resultados calculados para apenas um participante. Optou-se por deixar o termo média para esse caso porque o estudo poderá ser realizado, posteriormente, com outros participantes.

A questão **Esforço** foi medida, a princípio, sob duas perspectivas: o tempo gasto pelos testadores para a geração dos casos de teste (E_t) e o número de casos de teste gerados E_{nct}. A Figura 5.3 relaciona as médias do tempo gasto pelos participantes para a geração dos casos de teste e a Figura 5.4 relaciona as médias do número de casos de teste gerados para as duas categorias.

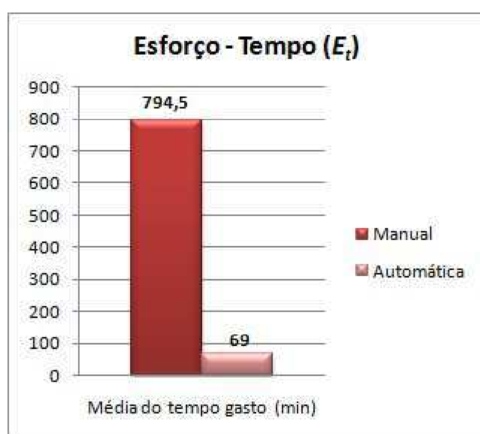


Figura 5.3: Comparativo do esforço com relação ao tempo.

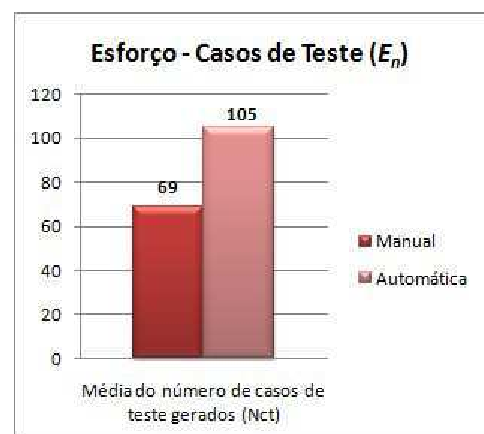


Figura 5.4: Comparativo do esforço com relação ao número de casos de teste.

De acordo com os resultados apresentados, observou-se que a média do tempo gasto para a geração de casos de teste usando a abordagem automática foi aproximadamente onze vezes menor do que o tempo gasto pela abordagem manual. Isso confirma a hipótese 1, definida no planejamento, que utilizando a abordagem automática, o esforço com relação ao tempo gasto é menor do que utilizando a abordagem de forma manual. Além disso, observou-se que o número de casos de teste gerados foi maior, numa proporção de aproximadamente um e meio (1,5) caso de teste gerado automaticamente para um caso de teste gerado manualmente.

Em se tratando da comparação manual x automático, já era esperado que os resultados

com a realização da abordagem automática com relação ao tempo, nesse caso, seriam melhores. No entanto, com relação ao número de casos de teste, era esperado os resultados fossem parecidos, uma vez que foi dado a cada participante que realizou a abordagem manualmente um guia de como construir os casos de teste de acordo com os padrões de teste. Entretanto, os fatores de variação com relação à experiência impactaram nos resultados, principalmente para o participante 1, o qual não tinha muito conhecimento em teste de integração, padrões de teste e U2TP, que ressaltou que teve dificuldades na construção de alguns casos de teste. Para o participante 2, o principal fator para a não construção de todos os casos de teste foi o tempo (ele ressaltou que deixou de construir muitos dos casos de teste porque estava cansado). É importante destacar que se um participante experiente em todos os quesitos inicialmente definidos (por exemplo o participante 3) realizasse a abordagem manualmente, é provável que os resultados com relação número de casos de teste fossem equivalentes no cenário automático x manual.

Com os dados obtidos para o cálculo das métricas anteriores, foi possível medir o esforço gasto pelo testador para a geração de um caso de teste. O cálculo dessa métrica se mostrou importante porque apesar da diferença do número de casos de teste não ter sido muito grande, o tempo foi um fator impactante. Em abordagens de teste de software, um tempo muito grande (ou pequeno) pode estar aparentemente relacionado a geração de muitos (ou poucos) casos de teste, dependendo da experiência e eficiência do testador. A Figura 5.5 relaciona a média do tempo gasto pelos participantes para a geração de um caso de teste.

Figura 5.5: Comparativo do esforço para a geração de um caso de teste.

De acordo com os resultados dessa métrica, tem-se que o a média do tempo gasto para a geração de um caso de teste usando a abordagem manual foi de aproximadamente doze

minutos. Já na abordagem automática, o tempo gasto foi de menos de um minuto. Vale salientar que na abordagem automática, o tempo refere-se apenas ao tempo de preparo dos modelos e da ferramenta, o tempo de execução da ferramenta não foi considerado. Isso implica que esse tempo poderá ser menor, dependendo da agilidade do testador.

Embora o número de casos de teste gerados seja uma métrica importante, é preciso medir também quantos casos de teste são considerados corretos/válidos. Essa métrica é importante porque testadores podem cometer erros na construção de casos de teste em abordagens manuais. Nesse contexto, a questão **Validade** foi medida com relação à porcentagem dos casos de teste considerados válidos dentre os casos de teste gerados. A Figura 5.6 relaciona as médias da porcentagem de casos de teste válidos para as duas categorias.

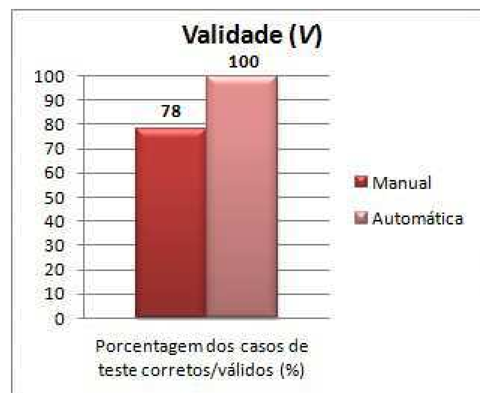


Figura 5.6: Comparativo da porcentagem de casos de teste válidos gerados.

De acordo com os resultados dessa métrica, tem-se que apenas 78% dos casos de teste gerados manualmente foram considerados válidos. Isso aconteceu porque muitos dos casos de teste gerados manualmente pelos participantes ou não representavam o cenário de integração proposto ou continham inconsistências como aplicações erradas dos estereótipos do perfil de teste, falta de emuladores para o teste de integração, mensagens erradas (fazendo com que o caso de teste não possa ser executado corretamente), entre outras. Possivelmente, os motivos para esses problemas também estão relacionados à experiência dos participantes, principalmente em relação ao conhecimento em teste de integração e em U2TP. Já os casos de teste gerados automaticamente foram considerados todos válidos, uma vez que eles foram gerados pela ferramenta de suporte desenvolvida. Mais detalhes sobre a análise da validade dos casos de teste para todos os participantes podem ser encontrados no Apêndice D.

Com relação à questão **Eficácia**, ela foi medida com relação à capacidade de detecção

de defeitos dos casos de teste gerados. A Figura 5.7 relaciona as médias da capacidade de detecção de defeitos para cada uma das categorias.

Figura 5.7: Comparativo da capacidade de detecção de defeitos dos casos de teste.

De acordo com os resultados dessa métrica, tem-se que a média da capacidade de detecção de defeitos dos casos de teste gerados pela abordagem automática foi de aproximadamente três defeitos por caso de teste e da abordagem manual foi de aproximadamente um defeito por caso de teste. Isso mostra que os casos de teste gerados automaticamente se mostraram mais eficazes (numa proporção três vezes maior) em comparação aos gerados manualmente. Embora os casos de teste gerados pelos participantes que realizaram a abordagem manual sejam eficazes, muitos defeitos poderiam escapar uma vez que o conjunto de casos de teste gerados não foi completo, ou seja, nem todos os casos de teste foram desenvolvidos de acordo com a especificação dos padrões e no contexto dos cenários de integração inicialmente definidos. Já os casos de teste gerados pela abordagem automática, além de aplicarem corretamente os padrões, cobrem todos os caminhos (usando o padrão *Round-trip Scenario Test*) da especificação do sistema sob teste.

É importante ressaltar que um ou mais casos de teste podem detectar o mesmo defeito. Por isso, para medir a eficiência de uma abordagem de teste, é preciso medir também a taxa de defeitos detectados com relação ao tempo despendido para a construção dos mesmos. Além disso, na prática, uma abordagem de teste pode ter uma eficiência maior que outra, e no entanto, esta última ser mais interessante por detectar defeitos mais relevantes. Por esse motivo, uma outra métrica que deve ser investigada para avaliar a eficiência é a relevância média dos defeitos detectados. Esta relevância pode ser medida com relação ao número de defeitos detectados e um valor atribuído ao defeito conforme sua relevância, seguindo algum

critério pré-definido. Contudo, essas métricas só teriam efeito significante se os casos de teste pudessem ser executados, por isso não foram abordadas nesse estudo de caso.

Finalmente, pode-se concluir que todas as hipóteses de estudo levantadas no planejamento foram confirmadas. No entanto, apesar dos resultados das métricas serem relevantes, é importante que novos estudos de caso sejam realizados para que conclusões mais precisas possam ser tiradas.

Análise Qualitativa dos Resultados

Após a conclusão do estudo experimental, foram avaliadas as impressões dos participantes com relação ao estudo realizado, utilizando um Questionário de Avaliação Pós-Estudo Experimental apresentado no Apêndice F e realizando entrevistas individuais, onde cada participante foi incentivado a emitir opiniões a respeito da abordagem e da condução do estudo.

Primeiramente, foi avaliado o nível de compreensão e aplicabilidade da abordagem manualmente, com o intuito de saber o grau de dificuldade na sua utilização. Todos os participantes acharam o grau de dificuldade de aplicar a abordagem mediano. Para estes participantes, a dificuldade de aplicação da abordagem está associada ao fato de que para alguns dos padrões, torna-se quase impossível gerar todos os casos de teste esperados devido ao número relativamente grande de casos de teste a serem gerados para o sistema sob teste (mais de 100). Além disso, como o objetivo era de gerar casos de teste para vários cenários de integração, muitos deles bem similares, um dos participantes alegou que a abordagem, para estes casos, se tornou cansativa por ser repetitiva e muito detalhada. Outro fator que implicou na aplicabilidade da abordagem foi que a ferramenta de modelagem utilizada não facilitou o trabalho, por ser uma versão shareware bastante limitada.

Embora os participantes que realizaram a abordagem manualmente tenham identificado algumas limitações, eles informaram que o impacto da abordagem na geração dos casos de teste no contexto de teste de integração foi positivo, uma vez a utilização de padrões guiando a equipe de teste faz com que situações em que defeitos, que antes poderiam passar despercebidos, fossem mais facilmente localizados. Em outras palavras, segundo eles, a abordagem ajudou a construir casos de teste que são importantes e que talvez eles não tivessem pensado em construir alguns dos propostos pelos padrões de teste, no caso

de utilizarem outra abordagem de teste. Além disso, um dos participantes ressaltou a importância do perfil de integração proposto na abordagem, pois a sua utilização facilitou bastante na visualização e identificação de quais estruturas (classes) estão sob teste em cada cenário do teste de integração.

Também foi avaliada a possibilidade de utilização da abordagem manual em projetos futuros em que os participantes estivessem envolvidos para que fosse estimado o nível de receptividade da abordagem. Todos os participantes informaram que utilizariam a abordagem sem qualquer restrição, mesmo considerando o grau de dificuldade mediano. Entretanto, todos ressaltaram que se existisse uma ferramenta de suporte automático à abordagem manual, facilitaria bastante o trabalho empregado para a geração dos casos de teste, poupando tempo e recursos, e ainda possibilitando a aplicação da abordagem para sistemas com maior grau de complexidade.

Por fim, com relação à efetividade da abordagem naquilo que se propõe e à sua relevância para o contexto na qual está inserida, todos os participantes que realizaram a abordagem manualmente concordaram que a abordagem é realmente importante e efetiva para a geração de casos de teste de integração no contexto de teste dirigido por modelo.

5.3 Procedimento do Estudo Experimental para Avaliação da Ferramenta

Da mesma forma que o estudo experimental anterior, o método GQM foi utilizado para a análise e definição da avaliação experimental da ferramenta de suporte à abordagem proposta nesse trabalho.

5.3.1 Definição

No contexto de abordagens automáticas, é importante também avaliar se as ferramentas que dão o suporte automático têm um desempenho adequado. Logo, o objetivo de medição nesse estudo tem a finalidade de:

> Analisar a ferramenta de suporte automático ao processo de geração de casos de teste (**objeto**), com o propósito de melhoria do processo (**propósito**), com

respeito ao desempenho, especificamente quanto à precisão e à escalabilidade (**foco de qualidade**), do ponto de vista do usuário-testador (**perspectiva**), e no contexto de automação de teste de software (**contexto**).

Como agora o objetivo é analisar o desempenho da ferramenta no contexto de geração automática de casos de teste usando padrões de teste, duas questões a serem analisadas são a **precisão** e a **escalabilidade**. Dado que se trata de uma ferramenta para a identificação e aplicação de padrões de teste para qualquer sistema OO, ela é precisa em relação à aplicação dos padrões de teste, ou seja, todos os padrões são aplicados para a geração dos casos de teste independente do sistema? Ainda, a ferramenta é escalável? Nesse contexto, as questões levantadas para esse objetivo, bem como suas respectivas métricas, as quais têm por objetivo quantificá-las, são:

- **Precisão.** No desenvolvimento de software é imprescindível verificar se o produto produzido faz o que realmente era esperado dele, ou seja, se ele é realmente preciso naquilo que é destinado a fazer. No contexto da abordagem proposta, é importante que a ferramenta identifique a aplicabilidade dos padrões de teste e gere os casos de teste satisfatoriamente, para qualquer sistema, desde que o sistema contenha as características necessárias. Considerando esse aspecto, a precisão *P*, para cada sistema, pode ser definida como:

$$P = \frac{N_{pi}}{N_{pa}}, \tag{5.7}$$

  onde N_{pi} é o número de vezes que os padrões de teste foram identificados para a aplicação, e N_{pa} é o número de vezes que os padrões de teste deveriam ser aplicados de acordo com as características do sistema.

- **Escalabilidade.** A escalabilidade é uma característica desejável em todo o software. É um assunto extremamente importante, pois implica no seu desempenho. Basicamente, escalabilidade é a capacidade do software de atender uma demanda crescente de uso. No contexto da abordagem proposta, é importante que a ferramenta seja executada com sucesso para qualquer caso, preferencialmente com o menor consumo de tempo possível, ou seja, que o tempo gasto pela ferramenta seja proporcional ao tamanho e

complexidade de qualquer sistema. Considerando esse aspecto, a escalabilidade S é definida como:

$$S = \frac{T_{exec}}{C_s}, \tag{5.8}$$

onde T_{exec} é tempo de execução (em milissegundos) gasto pela ferramenta para a geração dos casos de teste, e C_s é o tamanho do software para o qual se deseja gerar os casos de teste. O tamanho do software, nesse estudo de caso, é medido com relação ao número de interações, ou seja, o número mensagens enviadas/recebidas e fragmentos combinados dos diagramas de sequência. Essa decisão foi tomada porque os algoritmos para a geração dos casos de teste estão diretamente associados à essas interações, quanto maior o número de interações, maior é o tempo gasto pelos algoritmos para a geração dos casos de teste.

5.3.2 Planejamento

Seleção do Contexto (Ambiente e Participantes)

Para este estudo, o contexto escolhido também foi não-industrial, uma vez que o objetivo do estudo era avaliar o comportamento da ferramenta de suporte desenvolvida antes de finalizar uma versão para ser utilizada na academia e por empresas. Para este estudo, especificamente, não foram selecionados participantes externos, ou seja, a ferramenta foi experimentada pela sua própria equipe de desenvolvimento.

Formulação das Hipóteses

No contexto deste estudo de caso para a avaliação da ferramenta de suporte à abordagem automática para a geração de casos de teste usando padrões de teste, dentro do contexto de teste de integração com MDT, foram definidas hipóteses de estudo para cada uma das questões formuladas na seção 5.3.1. Estas hipóteses foram avaliadas por meio da avaliação dos dados coletados durante a execução do estudo de caso.

A primeira hipótese refere-se à questão da precisão da ferramenta, ou seja, se a ferramenta gera casos de teste para todos os padrões de teste documentados. Neste caso, a hipótese de estudo formulada foi:

- **Hipótese de estudo 1:** A ferramenta identifica e aplica corretamente todos os padrões de teste que devem ser aplicados em qualquer sistema, de acordo com suas características e independente de sua complexidade, ou seja,

  $\mu P = 1$, onde μP é a média da precisão da ferramenta para todos os sistemas.

A segunda hipótese refere-se à escalabilidade da ferramenta, ou seja, se o tempo gasto pela ferramenta é proporcional ao tamanho e complexidade de qualquer sistema. Neste contexto, a hipótese de estudo formulada foi:

- **Hipótese de estudo 2:** A ferramenta é escalável, ou seja,

  μS é satisfatório, onde μS é a média da escalabilidade da ferramenta para todos os sistemas.

Seleção das Variáveis

Para este estudo, as seguintes variáveis foram definidas:

- **Variáveis Independentes:** (i) os modelos (PIM) de entrada do processo de geração dos casos de teste para os sistemas definidos; (ii) a ordem de integração dos componentes (classes) do sistema, incluindo o que será integrado e suas dependências; e (iii) a ferramenta de suporte desenvolvida.
- **Variáveis Dependentes:** (i) os artefatos gerados com a utilização da ferramenta; e (ii) os resultados das funções para obtenção dos valores das métricas..

Princípios do Projeto do Estudo de Caso

Para este estudo de caso, não foi preciso definir o projeto do estudo seguindo os princípios gerais de projeto (aleatoriedade, blocagem e balanceamento), uma vez que a execução do estudo foi realizada pela própria equipe de desenvolvimento da ferramenta.

5.3.3 Execução e Resultados

A seguir, são detalhados a definição da execução e os resultados obtidos para o cálculo das métricas que são analisadas na fase de interpretação para este estudo de caso.

Definição da Execução

O estudo foi conduzido da seguinte forma: foram modelados quatro sistemas diferentes usando a ferramenta de modelagem MagicDraw e seus modelos no formato XMI foram submetidos à ferramenta para a aplicação da abordagem proposta. Para cada sistema foram desenvolvidos um diagrama de classes representando a estrutura do sistema e diagramas de sequência representando comportamentos do sistema para cada caso de uso. No contexto de teste de integração, o cenário inicial considerado para todos os sistemas foi aquele onde todas as classes são necessárias para o teste, ou seja, já foram testadas individualmente. Para a medição da precisão, mutações foram realizadas nos modelos para verificar o comportamento da ferramenta para outras situações.

A Tabela 5.7 apresenta detalhes sobre a modelagem desse sistemas. Especificamente, é apresentado o número de classes, o número de casos de uso (diagramas de sequência) e o número de interações dos diagramas de sequência (mensagens enviadas/recebidas e fragmentos combinados).

Tabela 5.7: Informações sobre os artefatos dos quatro sistemas utilizados.

Artefatos	**Sistema 1**	**Sistema 2**	**Sistema 3**	**Sistema 4**
Classes	4	12	6	25
Casos de uso	2	3	1	6
Interações	20	54	13	248

Durante o estudo, o tempo de execução dos módulos da ferramenta foi cronometrado para que os dados para medir a escalabilidade da ferramenta fossem coletados. Para que os dados para a obtenção das métricas com relação à escalabilidade fossem precisos, foi utilizada uma mesma estação de trabalho para a execução da ferramenta. Com relação à precisão, as configurações da estação de trabalho utilizada não impactam nos resultados.

Resultados Obtidos

A Tabela 5.8 mostra os dados obtidos após a execução do estudo para o cálculo da métrica precisão. Nesta tabela são apresentados, para cada sistema utilizado: (i) o número de vezes que os padrões de teste deveriam ser aplicados de acordo com as características do sistema

(N_{pa}); (ii) o número de vezes que os padrões de teste foram identificados para a aplicação (N_{pi}); (iii) o tempo gasto, em segundos, na execução do módulo de filtragem dos modelos e do módulo de geração de casos de teste (T_{exec}); e (iv) a complexidade do sistema com relação ao número de interações (C_s). No Apêndice D, é mostrado em detalhes como todos esses dados foram obtidos e analisados.

Tabela 5.8: Dados coletados com a execução da ferramenta para os quatro sistemas utilizados.

	N_{pa}	N_{pi}	T_{exec}	C_s
Sistema 1	6	6	12	20
Sistema 2	9	9	34	54
Sistema 3	3	3	7	13
Sistema 4	16	16	165	248

5.3.4 Interpretação

A fase de interpretação tem como objetivo fazer uma análise dos dados resultantes da fase de execução e resultados apresentada anteriormente com relação às hipóteses, métricas, questões e objetivos definidos. Para isso, foi realizada uma análise quantitativa desses dados de acordo com cada uma das questões levantadas na Seção 5.3.1.

Análise Quantitativa dos Resultados

A questão **Precisão**, levantada na Seção 5.3.1, foi medida com relação ao número de vezes que os padrões de teste foram identificados em comparação ao número de vezes que os padrões de teste deveriam ser aplicados. Conforme mostrado na seção anterior, todos os padrões foram corretamente identificados, de acordo com as características de cada sistema. Assim pode-se concluir que, nesse estudo, a ferramenta foi 100% precisa (μP = 1). É importante destacar também que todos os quatro padrões de teste documentados foram identificados e aplicados corretamente pela ferramenta, mostrando também que a ela é capaz de aplicar todos os padrões documentados independente do tipo e da complexidade do sistema.

Com relação à questão **Escalabilidade**, também levantada na Seção 5.3.1, ela foi medida com relação ao tempo gasto para a execução da ferramenta e com a complexidade do sistema (em número de interações). A Figura 5.8 apresenta as médias da escalabilidade (*S*) para cada um dos sistemas.

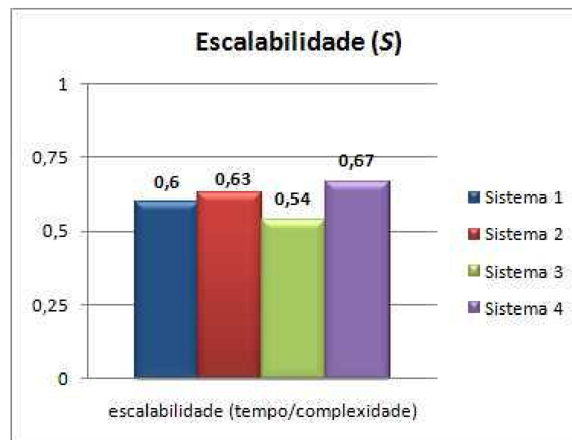


Figura 5.8: Comparativo da escalabilidade da ferramenta para os quatro sistemas utilizados.

De acordo com os dados da Figura 5.8, tem-se a relação do tempo gasto pela ferramenta para a geração de casos de teste para sistemas de complexidades diferentes foi, em média igual a 0,61. É possível perceber, no gráfico, que o valor da métrica para todos os sistemas, foi próximo dessa média. O único sistema que teve um valor mais distante (menor) foi o Sistema 3. Essa situação é justificada pelo fato de que, para esse sistema, o qual só tinha um caso de uso (embora ele tivesse treze interações), apenas dois padrões de teste foram aplicados e apenas um total de quatro casos de teste foram gerados. Dessa forma, é possível concluir que, para esse estudo de caso, a escalabilidade foi satisfatória (μS é satisfatório).

Finalmente, pode-se concluir que todas as hipóteses de estudo levantadas no planejamento foram confirmadas. No entanto, apesar dos resultados das métricas serem relevantes, é importante que novos estudos de caso sejam realizados para que conclusões mais precisas possam ser tiradas. Principalmente com relação à questão escalabilidade, pois nesse estudo de caso foram utilizados sistemas pequenos. Para medir, de fato, a escalabilidade de um software, é preciso de entradas bem mais complexas, ou seja, no caso dessa ferramenta, softwares com muitas classes, casos de uso e interações.

5.4 Considerações Finais

Este Capítulo apresentou a maneira como foram dirigidos os estudos experimentais de um processo de avaliação para a abordagem proposta neste trabalho e para ferramenta de suporte desenvolvida. Para a definição e análise desta avaliação, foi utilizado o método *Goal/Question/Metric* (GQM) [9] e os resultados das medições foram avaliados sobre as seguintes características: esforço, corretude, eficácia, precisão e escalabilidade.

O esforço, a corretude e a eficácia foram as principais características mensuradas e avaliadas nessa avaliação. O objetivo era mostrar o quanto a abordagem proposta pode ser efetiva, em termos de redução do esforço gasto na atividade de testes bem como a produção de casos de teste válidos e capazes de detectar defeitos importantes. Com relação à precisão e escalabilidade, apesar de serem características essenciais, elas estão relacionadas com a avaliação da ferramenta em si, uma vez que é importante que independente da complexidade dos sistemas verificados, todos as possíveis aplicações dos padrões sejam realizadas (dado que os sistemas estejam dentro do contexto dos padrões), e que a geração dos casos de teste demande o menor tempo possível.

Como forma de prover valores às métricas sugeridas, dois estudos de caso foram realizados dentro de um contexto não-industrial. No primeiro, cujo objetivo era avaliar a abordagem proposta neste trabalho, estudantes de graduação e pós-graduação do Curso de Ciência da Computação (CCC) da Universidade Federal de Campina Grande (UFCG), com experiência na área, atuaram como sujeitos do estudo experimental. O objetivo desta escolha foi diminuir o custo do estudo, uma vez que a sua aplicação em contextos industriais, incluindo a contratação de profissionais reais, não condizia com os recursos disponíveis. O segundo estudo de caso, cujo objetivo era avaliar a ferramenta de suporte automático à abordagem desenvolvida, foi realizado dentro do próprio contexto de desenvolvimento, com a equipe que desenvolveu a ferramenta.

Foi escolhida a realização de estudos de caso, e não experimentos formais e controlados, pelos seguintes motivos: os estudos de caso possuem um controle da investigação e custos em nível médio, enquanto que os experimentos formais possuem um controle da investigação e custos altos para a sua realização. Entretanto, a condução de estudos de caso também tem suas vantagens, tais como, uma maior proximidade entre o pesquisador e os fenômenos

estudados, a possibilidade de aprofundamento das questões levantadas, do próprio problema e da obtenção de novas e úteis hipóteses, e a grande capacidade de levantar informações e proposições para serem estudadas com métodos mais rigorosos de experimentação.

Os resultados obtidos com os cálculos das métricas mostraram que a abordagem automática proposta nesse trabalho é eficiente no que diz respeito à redução do esforço gasto pelos testadores e à geração de casos de teste confiáveis. Além disso, para essa avaliação, a ferramenta de suporte desenvolvida mostrou-se precisa, com relação à aplicabilidade dos padrões de teste documentados, e satisfatória em termos de tempo de execução. Não foi possível afirmar, de fato, que a ferramenta é escalável porque os sistemas utilizados como entrada são pequenos. Apesar dos resultados obtidos com o cálculo das métricas terem sido favoráveis, é necessário que outros estudos mais rigorosos sejam conduzidos para que seja possível garantir com segurança as questões levantadas.

Capítulo 6

Trabalhos Relacionados

Este capítulo tem a finalidade de relatar os principais trabalhos realizados no contexto de geração automática de casos de teste a partir de modelos, com foco em teste de integração e padrões de teste. Na Seção 6.1 são apresentadas abordagens e/ou estratégias de teste de integração usando técnicas de MBT. Na Seção 6.2 são apresentadas abordagens, na teoria e na prática, de MDT no geral. Na Seção 6.3 são apresentadas abordagens que integram MDT com padrões de teste. Por fim, na Seção 6.4 são apresentadas algumas considerações acerca da comparação entre tais abordagens com a abordagem proposta neste trabalho.

6.1 Teste de Integração Baseado em Modelos

Modelos são informações, denotadas em notações diagramáticas ou textuais, que descrevem uma abstração de um sistema [46]. Atualmente, a linguagem UML é um dos formalismos mais utilizados para a descrição de modelos de sistemas. Muitas estratégias de MBT que fazem uso de modelos UML para o desenvolvimento de testes já foram desenvolvidas e têm sido utilizadas na prática. Para testar diferentes aspectos da interação entre objetos, vários pesquisadores têm proposto diferentes técnicas baseadas em diagramas UML, especialmente, diagramas de interação [2, 8, 23, 47].

Basanieri e Bertolino [8] propõem um método para a geração de casos de teste usando diagramas de caso de uso e diagramas de sequência da UML. O foco é a verificação, por meio de testes de integração, de que as classes, previamente testadas, interagem corretamente. Para isso, é usado o método *Category Partition* [11] e são derivados, manualmente, casos

de teste baseados nas sequências de mensagens entre classes dos diagramas de sequência. Uma abordagem similar foi proposta por Hartmann *et al.* [23]. A diferença é que esta abordagem provê um ambiente para a geração e execução de testes usando *statecharts*. Um fator importante que difere esses dois trabalhos do aqui proposto é que eles não usa padrões de teste como guia para a geração de casos de teste, elas são estratégias baseadas em um padrão, e não estratégias sistemáticas que permitem a incorporação de outros padrões. Sem padrões de teste, pode se ter o problema de perda de foco na geração, uma vez que, em teste de integração, o conjunto de possíveis combinações para a geração dos casos de teste é normalmente muito grande.

Com relação à estratégias para a geração de casos de teste baseados nas interações entre classes a partir de modelos UML, existem alguns trabalhos na literatura, porém eles não usam padrões de teste como guia para a geração. Por exemplo, Sarma *et al.* [47] propõem uma abordagem que usa diagramas de sequência como fonte para a geração de casos de teste, onde cada diagrama é transformado em um grafo, cujos nós armazenam informações necessárias para a geração dos casos de teste. Estas informações são coletadas a partir de casos de uso estendidos, diagramas de classes e de um dicionário de dados expresso em OCL. A estratégia consiste em percorrer os grafos e gerar automaticamente casos de teste baseado em um critério de cobertura e em um modelo de faltas. Outra abordagem é proposta por Ali *et al.* [2], a qual combina diagramas de colaboração e *statecharts* para gerar automaticamente um modelo de teste intermediário (*State COllaboration TEst Model* - SCOTEM). Este modelo é utilizado para gerar caminhos de teste válidos. O gerador de teste usa o SCOTEM para gerar caminhos que correspondem a casos de teste que visam detectar faltas que possam surgir devido a estados inválidos durante as interações entre objetos.

Embora as abordagens acima mencionadas considerem modelos UML para a geração de casos de teste no contexto de teste de integração, elas não são abordagens dirigidas por modelos e não usam especificações da UML 2.0. Além disso, algumas dessas abordagens não são automáticas, o que pode fazer com que muitas delas não sejam utilizadas. Com relação à geração de código de teste executável, apenas o trabalho de Hartmann *et al.* e Ali *et al.* citam a geração de casos de teste executáveis para uma linguagem de programação específica, o que difere-nos do trabalho aqui proposto que não gera código de teste executável (o que será provido apenas quando o nível de abstração PSTM for desenvolvido). Com

relação à critérios de cobertura, da mesma forma que o trabalho aqui proposto, todos os trabalhos acima mencionados permitem a cobertura de todos os caminhos, uma vez que os casos de teste são gerados a partir de diagramas de interação.

6.2 Teste Dirigido por Modelo: Teoria e Prática

Um relevante número de trabalhos já foi desenvolvido tanto para a teoria quanto para a prática de MDT. Por exemplo, o trabalho proposto por Heckel e Lohmann [24] foca em modelos e processos de teste de componentes distribuídos. Embora nessa abordagem a MDA seja utilizada, a automação por meio de regras de transformação não é considerada. Além disso, a utilização de MDT é apenas motivada, nada de concreto é realmente apresentado. Da mesma forma, Javed *et al.* [26] propõem uma abordagem MDA, que consiste na geração de casos de teste de unidade a partir de modelos independentes de plataforma baseados em sequências de chamadas a métodos (SCM). Estes modelos são automaticamente convertidos em modelos de casos de teste concretos e executáveis aplicando regras de transformação entre modelos. Embora esta abordagem de MDT considere artefatos de MDA, diagramas UML não são utilizados para a modelagem, os modelos independentes de plataforma são construídos com SCMs, que são apenas partes de diagramas de sequência. Além disso, ela foca apenas em transformações no nível PITM-PSTM.

O uso do perfil U2TP em práticas de MDT para a geração automática de casos de teste se tornou comum nos últimos anos, como pode ser visto em [14, 30]. Dai [14] apresenta uma metodologia que aplica conceitos U2TP tanto para os modelos de desenvolvimento (do sistema) quanto para modelos de teste. Esta metodologia foi formalizada definindo regras de transformação entre modelos com a especificação QVT da OMG. No entanto, não é mostrado como a metodologia e as regras de transformação foram realmente desenvolvidas e experimentadas. O exemplo apresentado é bem simples e não mostra uma validação concreta da metodologia proposta. Lima *et al.* [30] propõem uma solução MDT para testes de componentes que utiliza U2TP para a geração automática de casos de teste baseado nas metodologias KobrA e BIT. Para a solução, foi desenvolvida a ferramenta MoBIT (*Model-driven Built-in contract Testers*) que consiste de transformações especificadas e executadas em ATL e que são aplicadas a modelos de componentes KobrA, baseados em

UML, instanciados a partir de meta-modelos UML e OCL da OMG. Apesar de considerar todos os artefatos MDA e por apresentar uma ferramenta que pode ser aplicável em qualquer domínio de aplicação, esta abordagem é melhor aplicada para sistemas simples, compostos por apenas dois componentes (um servidor e um cliente). Além disso, os casos de teste gerados não são em nível de sistema ou integração, e sim em nível de testes para o comportamento dos componentes individualmente.

Embora os trabalhos acima citados sejam abordagens dirigidas por modelos, o que possibilita que código de teste possa ser gerado automaticamente, apenas os trabalhos de Javed *et al.* e Lima *et al.* propõem a geração de código de teste concreto e executável (PITM-PSTM e PSTM-Código). Já com relação à critérios de cobertura, não foi possível fazer nenhuma análise concreta pois nenhum dos trabalhos deixa explícito quais critérios utilizaram, uma vez que não se tratam de estratégias para teste de integração.

Adicionalmente, Alguns trabalhos como o de Baker *et al.* [6] exploram a possibilidade do uso MDT e de U2TP para teste de integração. No entanto, até o momento da conclusão desse trabalho, não foram encontradas abordagens MDT sistemáticas para a geração de casos de teste nesse contexto.

6.3 Teste Dirigido por Modelo e Padrões de Teste

No contexto de abordagens que usam padrões de teste para a geração automática de casos de teste, pouco se tem feito. Uma única abordagem dirigida por modelos que relaciona padrões de teste para a geração de casos de teste a partir de modelos é apresentada por Im *et al.* [25]. Contudo, os casos de teste são gerados no contexto de teste de sistema. Nesta abordagem, cujo foco é a automação da definição de casos de teste usando padrões de teste no contexto de linhas de produto, casos de uso são construídos a partir de uma linguagem específica de domínio que, juntamente com templates de teste (desenvolvidos a partir de padrões de teste), especificam os casos de teste. No entanto, é importante ressaltar que nessa abordagem, diferentemente da que propomos neste trabalho, a identificação e aplicação dos padrões não são automatizadas e não são considerados artefatos da MDA e conceitos de U2TP. Além disso, nenhum exemplo concreto da utilização de padrões de teste para a geração dos casos de teste é apresentado. Com relação à geração de código de teste executável, os autores

apenas citam que a abordagem dá suporte à geração de casos de teste executáveis, nada de concreto é explicitado. Com relação à critérios de cobertura para os testes, não foi possível fazer nenhuma análise, uma vez que os autores não ressaltaram nada a respeito.

6.4 Considerações Finais

É perceptível o grande número de trabalhos no contexto de geração, manual ou automática, de casos de teste a partir de modelos, principalmente de modelos descritos com UML. Ainda, é crescente o aumento de técnicas MDT nesse contexto. Contudo, o escopo de teste de integração é ainda um pouco limitado. Praticamente, as abordagens dirigidas por modelos para a geração de casos de teste atuam no contexto de teste de unidade e componentes individuais e de sistema. Além disso, no contexto de abordagens sistemáticas que fazem uso de padrões de teste para a geração automática de casos de teste, pouco se tem feito, principalmente no contexto de teste de integração. Apesar de alguns dos trabalhos supracitados proporem abordagens que fazem uso de padrões de teste não é possível classificá-las como abordagens sistemáticas para a geração de casos de teste com aplicação de padrões de teste genéricas e de contexto geral.

A Tabela 6.1 apresenta de forma resumida um comparativo entre todos os trabalhos relacionados apresentados neste Capítulo, incluindo este trabalho. Cada coluna da Tabela identifica: (i) **Trabalho** - a referência do trabalho relacionado (o último refere-se a este trabalho); (ii) **Abordagem** - o contexto de teste baseado em modelo da abordagem que o trabalho usa/propõe; (iii) **Linguagem** - a linguagem de modelagem utilizada para descrição dos modelos e geração dos testes; (iv) **Tipo de Modelo** - o(s) tipo(s) de modelo(s) utilizado(s) de acordo com a linguagem; (v) **Tipo de Teste** - o nível do teste (ex. unidade, componente, integração ou sistema); (vi) **Padrões de Teste** - a relação do trabalho com padrões de teste (uso); (vii) **Código de Teste** - se a abordagem proposta gera casos de teste executáveis (código de teste); e (viii) **Cobertura Caminhos** - se o algoritmo para a geração dos casos de teste provê a cobertura de caminhos (critério de cobertura dos testes).

Esta Tabela mostra que nenhum dos outros trabalhos que utilizam MDT são para teste de integração, inclusive o trabalho de Dai [14] não deixa claro qual o tipo de teste utilizado. Os trabalhos que consideram teste de integração são abordagens MBT e são mais

Tabela 6.1: Resumo dos trabalhos relacionados.

Autor(es)	**Abordagem**	**Linguagem**	**Tipo de Modelo**	**Tipo de Teste**	**Padrões de Teste**	**Código de Teste**	**Cobertura Caminhos**
Basanieri e Bertolino [8]	MBT	UML	Diag. Sequência	Integração	Usou um padrão	Não	Sim
Hartmann *et al.* [23]	MBT	UML	*Statechart*	Integração	Usou um padrão	Sim	Sim
Sarma *et al.* [47]	MBT	UML	Diag. Sequência	Integração	Não	Não	Sim
Ali *et al.* [2]	MBT	UML	Diag. Colaboração	Sistema	Não	Sim	Sim
Heckel e Lohmann [24]	MDT	UML 2.0	Diag. Classes e Sequência	Sistema	Não	Não	Não
Javed *et al.* [26]	MDT	SCM	Diag. Sequência	Unidade	Não	Sim	Não
Dai [14]	MDT	UML 2.0	-	Sistema	Não	Não	Não
Lima *et al.* [30]	MDT	KobrA e UML 2.0	Máquina de Estados	Componentes	Não	Sim	Não
Im *et al.* [25]	MDD/MDT	DSL	Caso de Uso	Sistema	Sim	Não	Não
Este trabalho	MDD/MDT	UML 2.0	Diag. Classes e Sequência	Integração	Sim	Não	Sim

antigos. Ainda, poucos trabalhos utilizam a versão da UML 2.0. Em versões anteriores da UML, muitos recursos, tais como perfis UML e fragmentos combinados nos diagramas de sequência, não estão disponíveis. Além disso, a versão 2.0 da UML permite o alinhamento com conceitos da MDA para a geração automática de código, o que também não é provido nas versões mais antigas da UML. Com relação aos padrões de teste, apenas um trabalho faz referência ao uso de padrões de teste no geral. No entanto, os padrões não são aplicados automaticamente. A aplicação depende da experiência e conhecimento do testador em padrões de teste. Com relação à geração de casos de teste executáveis, a tabela mostra que existem alguns trabalhos que provêem isso, diferentemente da abordagem proposta neste trabalho. No entanto, a implementação do nível de abstração PSTM, e consequentemente as transformações de PSTM para código é possível e está planejada como trabalho futuro. Com relação à cobertura de caminhos, os trabalhos no contexto de teste de integração atingem esse objetivo, as demais não deixam explícitos comentários sobre esse critério. Por fim, a tabela mostra que não existe nenhum trabalho diretamente relacionado ao aqui proposto, apenas são trabalhos que relacionam alguma(s) das tecnologias e estratégias utilizadas em comum.

Capítulo 7

Conclusões

O trabalho aqui apresentado propõe uma nova abordagem de teste de integração, definida dentro de um processo integrado de desenvolvimento e teste dirigidos por modelos (MDD/MDT), para a geração automática de casos de teste a partir de modelos de desenvolvimento, descritos por diagramas UML 2.0, usando padrões de teste. A utilização de padrões de teste nesse sentido é gerar casos de teste de integração mais eficientes e adequados ao contexto dos sistemas que serão testados, uma vez que são reutilizadas técnicas/estratégias que se mostraram efetivas em contextos similares. Tal característica permite que defeitos comuns e relevantes possam eventualmente ser capturados, assegurando uma melhor qualidade do sistema desenvolvido. Além disso, no contexto de teste de integração, dependendo da complexidade do sistema, o número dos casos de teste pode ser muito grande. Neste sentido, o uso de padrões pode guiar a escolha de casos de teste mais efetivos dentre um número muito grande de possibilidades.

O ponto principal deste trabalho é a proposta de uma forma automática de apoio para a atividade de teste de integração realizada na prática pela equipe de testadores, e, consequentemente, a não adição de custos extras associados a tarefa de testes, uma vez que a abordagem não se utiliza de artefatos além da própria especificação funcional (modelos descritos por diagramas UML) do sistema a ser testado. Esta característica permite que a abordagem tenha um impacto reduzido no total de esforço a ser empreendido no processo como um todo, uma vez que não é preciso a construção de modelos específicos para testes pelos testadores. Contudo, é importante destacar que é preciso que estes modelos sejam suficientemente completos (do ponto de vista dos padrões de teste documentados e das

funcionalidades a serem testadas), consistentes e corretos, para que eles possam ser usados diretamente pela abordagem para a a geração dos casos de teste. Caso contrário, os casos de teste não serão gerados corretamente.

A abordagem desenvolvida consiste em três atividades principais: (i) adaptação dos modelos de desenvolvimento para teste de integração, no intuito de simplificar os modelos de desenvolvimento para que apenas as classes importantes para o teste de integração sejam consideradas; (ii) identificação da aplicação de padrões de teste, cujo objetivo é mostrar para o testador quais são os padrões que podem ser utilizados para a geração dos casos de teste para que ele possa decidir quais utilizar; e (iii) geração de casos de teste de integração independentes de plataforma documentados de acordo com o perfil de teste U2TP e com base nos padrões de teste identificados. Para dar apoio a essas atividades, foi desenvolvido um perfil UML, cujo objetivo é identificar as classes que farão parte do teste de integração, e um meta-modelo que permite identificar a aplicabilidade do padrão em nível de modelos. Como suporte automático à abordagem proposta, foi desenvolvida uma ferramenta baseada nos princípios de MDA. Esta ferramenta, consiste em três módulos contendo meta-modelos, modelos e regras de transformação especificadas com a linguagem ATL.

Ao todo, foram documentados e implementados quatro padrões de teste existentes na literatura. Contudo, foi disponibilizado um catálogo e um formato de definição para que novos padrões de teste possam ser documentados e posteriormente incorporados à ferramenta. Para isso, é preciso que os padrões de teste de interesse sejam documentados de acordo com o formato de definição proposto e, para a automação, as respectivas regras de transformação sejam desenvolvidas.

Como forma de avaliação da abordagem proposta nesse trabalho foi realizado um estudo experimental utilizando o método *Goal/Question/Metric* (GQM). Nesta avaliação observou-se que a abordagem, mesmo sendo realizada de forma manual, ou seja, sem suporte automático, é viável, embora trabalhosa. Contudo, o estudo mostrou a importância da ferramenta de suporte automático desenvolvida para viabilizar, efetivamente, a aplicação da abordagem proposta para a geração de casos de teste mais adequados, efetivos e consistentes em tempo hábil, reduzindo assim o esforço despendido na prática na atividade de teste como um todo, e, especificamente, no contexto de teste de integração.

7.1 Trabalhos Futuros

Esta seção apresenta possíveis trabalhos futuros. Alguns destes possíveis trabalhos futuros são limitações deste trabalho, outros são trabalhos importantes que o complementam:

- **Extensão do catálogo de padrões:** atualmente o catálogo de padrões tem apenas quatro padrões documentados e implementados pela ferramenta. Embora os padrões documentados sejam importantes para a detecção de defeitos relevantes no contexto de teste de integração, novos padrões devem ser incorporados para assegurar que mais defeitos podem ser revelados com os casos de teste gerados;

- **Submeter o trabalho a outros experimentos:** foram realizados dois estudos de caso com o objetivo de avaliar tanto a abordagem proposta quanto a ferramenta de suporte. Contudo, para que seja obtida uma maior confiança com relação a abordagem proposta, é preciso submetê-la a experimentos formais com um maior nível de controle, envolvendo mais participantes e utilizando sistemas maiores e mais complexos, devendo, inclusive, ser realizados em contextos industriais. Além disso, é importante avaliar a capacidade de detecção de defeitos real dos casos de teste, através da execução com códigos das aplicações. Adicionalmente, experimentos também devem ser realizados para comparar a abordagem proposta com outras abordagens semelhantes para a geração de casos de teste de integração, no intuito de analisar a sua efetividade com relação às abordagens existentes;

- **Acrescentar um módulo para a automação da definição dos cenários de integração:** atualmente, a definição dos cenários de integração é feita de forma manual, com o auxílio do perfil IOP desenvolvido. É importante que esse passo também seja automatizado, utilizando estratégias de análise de dependência entre classes. Existem ferramentas na literatura que já resolvem isso [22, 29, 42]. Uma possível integração com essas ferramentas deve ser explorada;

- **Implementar o nível de abstração PSTM:** neste trabalho, o foco foi direcionado apenas nos modelos independentes de plataforma. É importante estender a abordagem para que o nível de abstração dependente da plataforma (PSTM) seja contemplado. Neste nível, regras ATL deverão ser desenvolvidas utilizando dois diferentes

meta-modelos (1) o mata-modelo de UML 2.0 e (2) o meta-modelo de alguma linguagem de programação orientada a objeto específica (por exemplo JAVA [7]), para que os modelos de teste sejam mapeados para uma plataforma específica. Em particular, para a linguagem JAVA existe um *framework* para a escrita de testes automáticos, o JUNIT, para o qual o U2TP estabelece mapeamento dos seus conceitos [6]. Assim, através de especificações de testes modeladas com o U2TP é possível tanto representar os artefatos utilizados no processo de teste, como também utilizá-los como ponto de partida para a geração automática do código de teste. Logo, a implementação do nível de abstração PSTM é importante para que o código dos casos de teste seja efetivamente gerado com dados de teste reais (transformação PSTM-Código através de regras modelo-texto) para que eles possam ser executados de acordo com os objetivos de teste a serem atingidos;

- **Estender o meta-modelo para a documentação dos padrões**: o meta-modelo *TestPatternMetamodel* para documentação dos padrões permite apenas a identificação automática dos padrões e não aplicação deles. Atualmente, a construção dos algoritmos para a aplicação das estratégias dos padrões de teste é feita manualmente nas regras ATL. Dessa forma, o usuário que desejar inserir um novo padrão na ferramenta deve conhecer bem a linguagem ATL. Logo, no intuito de uma possível geração automática das próprias regras ATL, uma extensão do meta-modelo proposto deve ser investigada, para que a incorporação de novos padrões à ferramenta seja facilitada;

- **Investigar uma forma para geração automática das regras de transformação ATL:** atualmente, a construção das regras ATL que aplicam os padrões é feita manualmente. Como MDA fornece padrões para realização de transformações entre vários níveis de abstração, é interessante que as próprias regras ATL que aplicam os padrões sejam geradas automaticamente, a partir dos modelos documentados de acordo com o meta-modelo *TestPatternMetamodel* estendido (trabalho futuro anterior), utilizando, entre outros artefatos, regras de transformação modelo-texto. Embora não seja uma tarefa trivial, é um ponto interessante a ser investigado, até mesmo para outros trabalhos no contexto de MDA.

Bibliografia

[1] ALEXANDER, C. *The Timeless Way of Building*. Oxford University Press, 1979.

[2] ALI, S., BRIAND, L. C., REHMAN, M. J., ASGHAR, H., IQBAL, M. Z., AND NADEEM, A. A state-based approach to integration testing based on uml models. *Information and Software Technology 49*, 11-12 (Nov. 2007), 1087–1106.

[3] ALVES, E. L. G., MACHADO, P. D. L., AND RAMALHO, F. Uma Abordagem Integrada para Desenvolvimento e Teste Dirigido por Modelos. In *Proceedings of 2nd Brazilian Workshop on Systematic and Automated Software Testing* (Campinas, São Paulo, 2008).

[4] AMBLER, S. W. *Process Patterns: Building Large-scale Systems Using Object Technology*. Cambridge University Press/SIGS Books, 1998.

[5] AMMA PROJECT. Atlas transformation language, 2005. http://www.sciences.univ-nantes.fr/lina/atl/.

[6] BAKER, P., DAI, Z. R., GRABOWSKI, J., HAUGEN, O., SCHIEFERDECKER, I., AND WILLIAMS, C. *Model-Driven Testing: Using the UML Testing Profile*. Springer, 2007.

[7] BARBERO, M. Javaabstractsyntax, 2006. Disponível em http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/#JavaAbstractSyntax.

[8] BASANIERI, F., AND BERTOLINO, A. A Practical Approach to UML-Based Derivation of Integration Tests. In *4th International Software Quality Week Europe (QWE'2000)* (Brussels (Belgium), Nov. 2000), pp. 20–24.

[9] BASILI, V. R., CALDIERA, G., AND ROMBACH, H. D. Goal Question Metric Paradigm. *Encyclopedia of Software Engineering 1* (1994), 528 a 532.

[10] BEIZER, B. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1995.

[11] BINDER, R. V. *Testing Object-Oriented Systems: Models, Patterns and Tools*, 5a. ed. Pearson Addison-Wesley, 2000.

[12] BRAIND, L. C., FENG, J., AND LABICHE, Y. Experimenting with Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders. Tech. Rep. SCE-02-03, Carleton University, 2002.

[13] BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[14] DAI, Z. R. Model-Driven Testing with UML 2.0. In *Proceedings of Second European Workshop on Model Driven Architecture (MDA) with an Emphasis on Methodologies and Transformations* (Canterbury, England, Sept. 2004), pp. 179–187.

[15] DALAL, S. R., JAIN, A., KARUNANITHI, N., LEATON, J. M., LOTT, C. M., PATTON, G. C., AND HOROWITZ, B. M. Model-Based Testing in Practice. In *International Conference on Software Engineering* (1999), pp. 285–294.

[16] ECLIPSE MODELING FRAMEWORK PROJECT (EMF). http://www.eclipse.org/modeling/emf.

[17] EL-FAR, I. K., AND WHITTAKER, J. A. Model-based Software Testing. *Encyclopedia of Software Engineering* (2001).

[18] ETSI DRAFT TECHNICAL REPORT. Methods for Testing and Specification (MTS); Patterns in Test Development (PTD). Tech. Rep. DTR/MTS-00091, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2005.

[19] FOWLER, M. *Analysis Patterns: Reusable Object Models (Addison-Wesley Object Technology Series*. Addison-Wesley Professional, 1997.

[20] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[21] GOMAA, H. *Designing Concurrent, Distributed, And Real-time Applications with UML*. Addison-Wesley, 2000.

[22] HANH, V. L., AKIF, K., TRAON, Y. L., AND JÉZÉQUEL, J. M. Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies. In *Proceedings of ECOOP 2001, LNCS 2072* (Berlin Heidelberg, 2001), pp. 381–401.

[23] HARTMANN, J., IMOBERDORF, C., AND MEISINGER, M. UML-Based Integration Testing. *SIGSOFT Softw. Eng. Notes 25*, 5 (2000), 60–70.

[24] HECKEL, R., AND LOHMANN, M. Towards Model-Driven Testing. *Electronic Notes in Theoretical Computer Science 82* (2003), 6–16.

[25] IM, K., IM, T., AND MCGREGOR, J. D. Automating Test Case Definition Using a Domain Specific Language. In *Proceedings of the 46th Annual ACM Southeast Conference* (Auburn, Alabama, 2008).

[26] JAVED, A. Z., STROOPER, P. A., AND WATSON, G. N. Automated Generation of Test Cases Using Model-Driven Architecture. In *of the 2nd International Workshop on Automation of Software Test at the 29th International Conference on Software Engineering* (Minneapolis, USA, 2007).

[27] KLEPPE, A., WARNER, J., AND BAST, W. *MDA Explained: The Model Driven Architecture - Practice and Promise*. Addison-Wesley, 2003.

[28] LANGE, M. It's Testing Time! Patterns for Testing Software. In *Proceedings of Sixth European Conference on Pattern Languages of Programs (EuroPLoP 2001)* (Irsee, Germany, 2001).

[29] LIMA, G. M. P. S., AND TRAVASSOS, G. H. Testes de Integração Aplicados a Software Orientado a Objetos: Heurísticas para Ordenação de Classes.

[30] LIMA, H., RAMALHO, F., MACHADO, P., AND ALVES, E. Automatic Generation of Platform Independent Built-in Contract Testers. In *Brazilian Symposium on Software Components, Architectures and Reuse* (Aug. 2007), pp. 47–60.

[31] MACIEL, C. L., MACHADO, P. D. L., AND RAMALHO, F. Uma Técnica MDT para a Geração Automática de Casos de Teste Usando Padrões de Teste. In *Anais do 3nd Brazilian Workshop on Systematic and Automated Software Testing (SBMF/SAST 2009)* (Gramado, Rio Grande do Sul, 2009), vol. 1, pp. 1–10.

[32] MACIEL, C. L. Ferramenta de Suporte. Disponível em http://www.dsc.ufcg.edu.br/∼camila/toolsuport.

[33] MAGICDRAW UML. http://www.magicdraw.com/.

[34] MESZAROS, G. *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[35] NASCIMENTO, L. H. O. Abordagens para avaliação experimental de teste baseado em modelos de aplicações reativas. Master's thesis, Universidade Federal de Campina Grande, 2008.

[36] OBJECT MANAGEMENT GROUP. http://www.omg.org/.

[37] OBJECT MANAGEMENT GROUP. UML 2.0 Testing Profile. Technical Report formal/05-07-07, OMG, July 2005. http://www.omg.org/cgi-bin/doc?formal/05-07-07.

[38] OBJECT MANAGEMENT GROUP. OCL Specification, v2.0. Technical Report formal/2006-05-01, OMG, May 2006. ttp://www.omg.org/technology/documents/formal/ocl.htm.

[39] OBJECT MANAGEMENT GROUP. MOF 2.0/XMI Mapping, v2.1.1. Technical Report formal/2007-12-01, OMG, Dec. 2007. http://www.omg.org/spec/XMI/2.1.1/.

[40] OBJECT MANAGEMENT GROUP. UML superstructure, v2.1.1. Technical Report formal/07-02-05, OMG, Feb. 2007. http://www.omg.org/cgi-bin/doc?formal/07-02-05.

[41] OBJECT MANAGEMENT GROUP. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Technical Report formal/08-04-03, OMG, Apr. 2008. http://www.omg.org/spec/QVT/1.0/.

[42] ORSO, A. *Integration Testing of Object-Oriented Software*. PhD thesis, Politecnico di Milano, 2000.

[43] PRESSMAN, R. S. *Engenharia de Software*, 6a. ed. McGraw-Hill, 2006.

[44] REID, S. Model-Based Testing. Canfield University Royal Military College of Science, 2006. http://shakti.it.bond.edu.au/sand/TAW06/SReid-paper.pdf.

[45] ROCHA, A. C. O. Automação da técnica de inspeção guiada para conformidade entre requisitos e diagramas uml. Master's thesis, Universidade Federal de Campina Grande, 2010.

[46] RUMPE, B. Model-based Testing of Object-Oriented Systems, 2003. citeseer.ist.psu.edu/675219.html.

[47] SARMA, M., KUNDU, D., AND MALL, R. Automatic Test Case Generation from UML Sequence Diagrams. In *In Proceedings of the 15th International Conference on Advanced Computing and Communications* (2007).

[48] SOMMERVILLE, I. *Engenharia de Software*, 8a. ed. Pearson Addison-Wesley, 2008.

[49] THOMAS, J., YOUNG, M., BROWN, K., AND GLOVER, A. *Java Testing Patterns*. Wiley, 2004.

[50] WOHLIN, C., RUNESON, P., HOST, M., OHLSSON, C., REGNELL, B., AND WESSLÉN, A. *Experimentation in Software Engineering: an Introduction*. Kluver Academic Publishers, 2000.

Apêndice A

Catálogo de Padrões de Teste

Os padrões de teste utilizados nesse trabalho são: (i) *Round-trip Scenario Test* [11]; (ii) *Abstract Class Test* [11]; (iii) *Error Simulator* [28]; e (iv) *Test Case With Time Specification* [28]. A seguir, estes padrões são apresentados de acordo com o formato de definição proposto nesse trabalho.

A.1 *Round-trip Scenario Test*

Descrito no Capítulo 3.

A.2 *Abstract Class Test*

Este padrão consiste basicamente na geração de casos de teste para classes abstratas e interfaces. Este padrão foi escolhido porque pode ser necessário para testar cenários onde há interações entre estes tipos de classe com as demais do sistema. Seguindo o formato de definição descrito no Capítulo 3, tem-se:

- **Nome:** *Abstract Class Test*.
- **Contexto:** Uma classe abstrata, ou interface, não pode ser testada diretamente ou usada para teste, pois ela não pode ser instanciada. No entanto, casos de teste podem ser construídos fazendo-se uso de uma sub-classe dela com uma implementação para todos os seus métodos (no caso da classe abstrata, cada método abstrato).

- **Intenção:** Desenvolver casos de teste para cenários que envolvem classes abstratas ou interfaces.

- **Modelo de Faltas:** Classes abstratas não apresentam muitos riscos de faltas além simples erros de *design* e de código. Erros comuns destes tipos são: identificação de responsabilidades comuns incompleta ou inconsistente; representação de responsabilidades incorreta ou incompleta; etc. Contudo, algumas faltas relacionadas com a interação entre as classes podem ser propagadas.

- **Identificação:** O Código A.1, apresenta uma instância do meta-modelo *TestPatternMetamodel*, representada no formato XMI, para este padrão. Neste Código, é possível perceber quais os elemento(s) do meta-modelo de UML deve(m) ser investigado(s) nos modelos para a aplicação do padrão.

  Código Fonte A.1: Elementos que identificam a aplicação do padrão *Abstract Class Test* de acordo com o meta-modelo *TestPatternMetamodel*.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tpm:Model xmlns:xmi="http://www.omg.org/XMI" xmlns:tpm="TestPatternMetamodel">
3    <testPattern name="Abstract Class Test">
4      <in type="Class">
5        <patternConstraint>
6          <specification type="tpm:OpaqueExpression"
7           body="context Class inv: self.isAbstract = true and Lifeline->allInstances()
8                  ->exists(l | l.represents.type.name = self.name);"
9           language="OCL 2.0"/>
10       </patternConstraint>
11     </in>
12     <in type="Interface">
13       <patternConstraint>
14         <specification type="tpm:OpaqueExpression"
15          body="context Interface inv: Lifeline->allInstances()
16                 ->exists(l | l.represents.type.name = self.name);"
17          language="OCL 2.0"/>
18       </patternConstraint>
19     </in>
20   </testPattern>
21 </tpm:Model>
```

  De acordo com a especificação do padrão, para que seja possível sua aplicação, basta que existam os elementos *Class*, com a propriedade *isAbstract* igual a *true*

(restrição), e *Interface* devem ser investigados. Além disso, como o interesse é testar as interações entre classes, este padrão deve ser aplicado apenas para os elementos (*Class* ou *Interface*) que estejam representados por linhas de vida em diagramas de sequência nos modelos, sendo assim uma restrição. Esta restrição pode ser visualizada pelo fragmento *Lifeline->allInstances()->exists(l | l.represents.type.name = self.name)* (linhas 11-12 e 20-21).

- **Estratégia:** Uma sub-classe deve ser desenvolvida em substituição à classe abstrata (ou interface) com uma implementação para cada método abstrato. Casos de teste devem ser desenvolvidos para testar a interação da sub-classe com outras classes usando outros padrões de teste. Neste caso, é utilizado o padrão *Round-trip Scenario Test*, para a geração de casos de teste para verificar potenciais defeitos nas iterações entre as demais classes do sistema. Além disso, para cada sub-classe que implementa tal classe abstrata (ou interface), os casos de teste desenvolvidos devem ser reusados, à medida que tais sub-classes são desenvolvidas, testadas individualmente e integradas.

- **Exemplos:** Utilizando o mesmo exemplo apresentado no padrão *Round-trip Scenario Test*, considere que a classe *MyList* é agora uma classe abstrata. Nesse contexto, uma sub-classe *MyListTest* foi desenvolvida com uma implementação para cada método abstrato, e utilizada em substituição à classe *MyList* para a geração dos casos de teste, conforme pode ser observado na Figura A.1. A Figura A.1(a) apresenta um fragmento do diagrama de classes que mostra a classe *MyListTest*, a qual estende a classe *MyList*. Como esta classe foi construída exclusivamente para testes, ela foi anotada com o estereótipo «*TestComponent*», conforme especificado pelo perfil U2TP. A Figura A.1(b) apresenta um caso de teste, documentado de acordo com U2TP, para o diagrama de sequência *Remove Object* considerando o padrão aqui definido e o padrão *Round-trip Scenario Test*.

- **Padrões Relacionados:** O padrão *Round-trip Scenario Test* [11]. Outros padrões também poderiam ser utilizados em conjunto.

- **Referência:** Este padrão foi descrito por R. Binder e pode ser encontrado em [11].

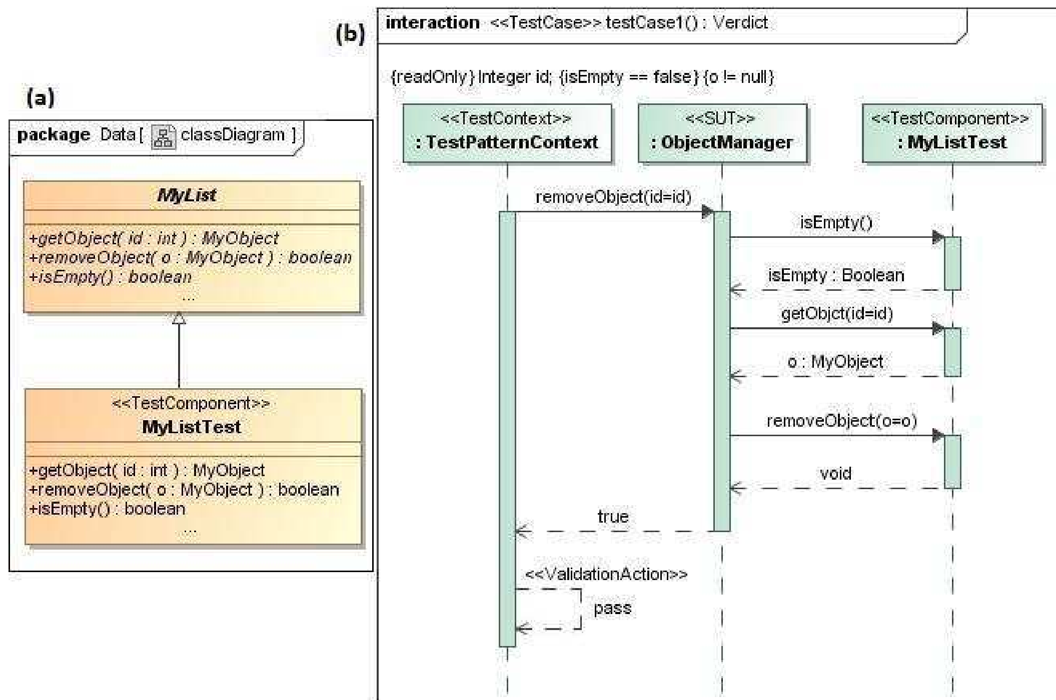


Figura A.1: (a) Fragmento do diagrama de classes mostrando o «*TestComponent*» desenvolvido. (b) Um caso de teste exemplo para o diagrama de sequência *Remove Object*, documentado de acordo com U2TP e aplicando o padrão *Abstract Class Test*.

A.3 *Error Simulator*

Este padrão consiste basicamente em ocasionar erros de condição de corrida com o intuito de testar o tratamento de erro/exceções do sistema. Este padrão foi escolhido porque casos de teste podem ser gerados para verificar se o tratamento de exceções que apenas são lançadas em tempo de execução foi realizado corretamente. Além disso, é possível verificar a interação entre classes servidoras (que lançam exceções) e clientes (que tratam exceções), bem como a interação com suas respectivas classes dependentes. Seguindo o formato de definição descrito no Capítulo 3, tem-se:

- **Nome:** *Error Simulator*.
- **Contexto:** Sistemas que consistem de múltiplas *threads* e processos, e sistemas que usam recursos compartilhados hierarquicamente, contém um conjunto de erros adicionais causados por condições de corrida. Este cenário - múltiplas *threads* e processos e recursos compartilhados - pode torna-se muito mais complexo.

Dependendo do real caminho da execução, na prática, é impossível prever quando conflitos nesse contexto podem ocorrer. Dessa forma, para suficientemente testar um componente, também é preciso testar os casos onde esses conflitos ocorrem.

- **Intenção:** Desenvolver casos de teste para a verificação dos tratamentos de erros/exceções do sistema.

- **Modelo de Faltas:** Várias faltas podem acontecer relacionadas ao tratamento de exceções, como: uma exceção inapropriada é passada de um servidor a um cliente; uma exceção é propagada fora do escopo, um tratador de exceção cria um efeito colateral indesejado; etc.

- **Identificação:** O Código A.2, apresenta uma instância do meta-modelo *TestPatternMetamodel*, representada no formato XMI, para este padrão. Neste Código, é possível perceber quais os elemento(s) do meta-modelo de UML deve(m) ser investigado(s) nos modelos para a aplicação do padrão. De acordo com a especificação do padrão, para que seja possível sua aplicação, basta que exista uma *Class*, com uma operação que lança exceções (com a propriedade *raisedException*, que é uma coleção, povoada com classes que representam exceções, ou seja, estereotipadas com «*Exception*»). Além disso, deve existir uma classe que faz o tratamento das exceções lançadas. Uma alternativa para a representação, em nível de modelos, de tratamento de exceções é, no diagrama de sequência, a combinação de um fragmento combinado do tipo *break* com (a condição *caught exception (e)*) com outro fragmento combinado do tipo *alt* com um *operand* representativo para cada exceção lançada pela operação da classe em questão. Um exemplo desse comportamento é mostrado na Figura A.2.

Código Fonte A.2: Elementos que identificam a aplicação do padrão *Error Simulator* de acordo com o meta-modelo *TestPatternMetamodel*.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <tpm:Model xmlns:xmi="http://www.omg.org/XMI" xmlns:tpm="TestPatternMetamodel">
3   <testPattern name="Error Simulator">
4     <in type="Class">
5       <patternConstraint>
6         <specification type="tpm:OpaqueExpression"
7          body="context Class inv:
8            self.ownedOperation->exists
```

```
9             (op | op.raisedException.notEmpty() and CallEvent->allInstances()
10             ->select(e1|not e1.operation.oclIsUndefined())->exists(e2 |
11               e2.operation = op) and CombinedFragment->allInstances()->exists
12                (cb | cb.interactionOperator = #break and cb.operand->first()
13                 .guard.specification.body.first() = 'caught exception (e)' and
14                  cb.operand->first().fragment->first().oclIsTypeOf(
15                   UML2!CombinedFragment) and b.operand->first().fragment->first()
16                   .interactionOperator = #alt and op.raisedException->forAll
17                      (e | cb.operand->first().fragment->first().operand->exists
18                       (o1 | o1.guard.specification.body.first()
19                        = 'isClassOf(e, '+ e.name.toString() + ')')));"
20           language="OCL 2.0"/>
21        </patternConstraint>
22      </in>
23    </testPattern>
24  </tpm:Model>
```

- **Estratégia:** Este padrão provê um mecanismo para simular conflitos de recursos compartilhados. Para isso, primeiro desenvolva um tratamento de erro que independe do erro propriamente e condicione sua execução à geração de um número randômico. Gere casos de teste que verifiquem as funcionalidades executadas após o tratamento da exceção, usando o padrão *Round-trip Scenario Test*. Para testar, execute o sistema diversas vezes de forma a garantir as diversas execuções.

- **Exemplos:** Considere o cenário mostrado na Figura A.2 onde o método *open()* da classe *File* lança duas exceções: *WrongFileNameException* e *AccessDeniedException*, (identificado pelo campo *raisedException*).

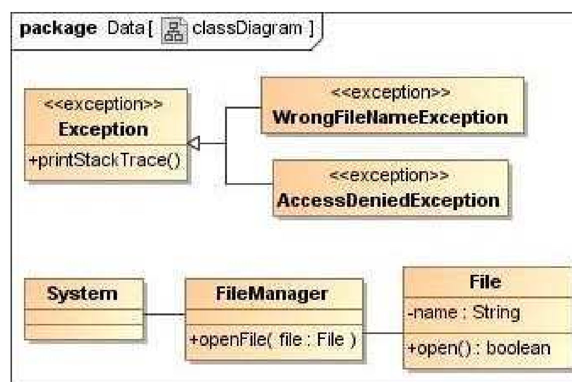


Figura A.2: Fragmento do diagrama de classes do *Object Manager System*.

O método *openFile()* da classe *FileManager*, por sua vez, realiza o tratamento das

exceções, conforme mostrado na Figura A.3.

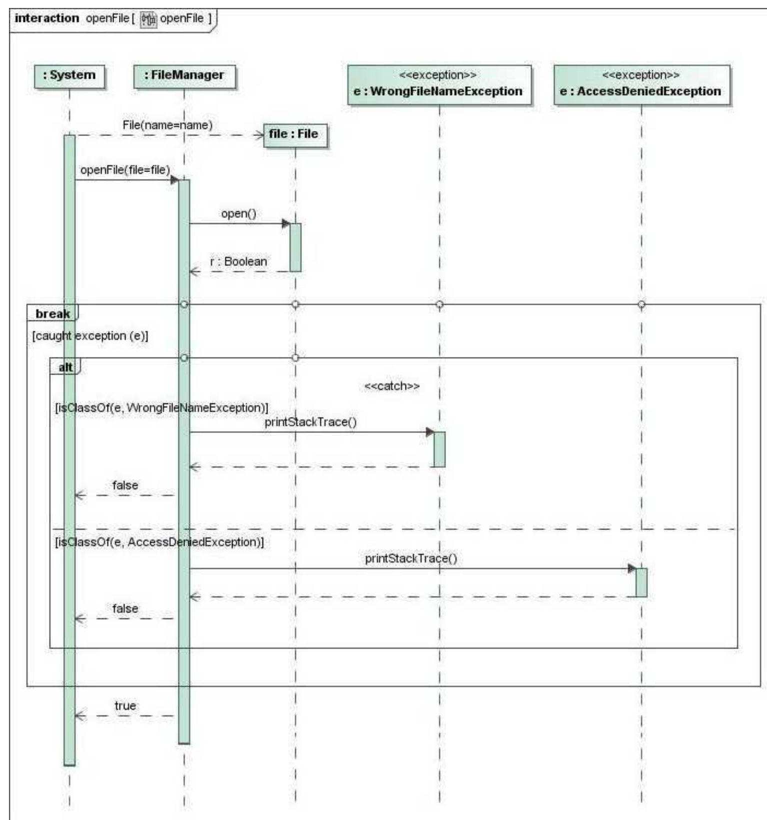


Figura A.3: Diagrama de sequência *openFile* do *Object Manager System* mostrando o tratamento de exceções.

Aplicando o padrão tem-se os seguintes artefatos de teste gerados: classe *FileHandler*, que é uma subclasse de *File* e re-implementa o método *open()* para que o lançamento da exceção seja forçado condicionado a um número randômico; e um diagrama de sequência mostrando como os casos de teste foram desenvolvidos. Como a classe *FileHandler* foi construída exclusivamente para testes, ela foi anotada com o estereótipo «*TestComponent*» conforme especificado pelo perfil U2TP. A Figura A.4 apresenta um caso de teste, documentado de acordo com U2TP, para o diagrama de sequência *openFile* considerando o padrão aqui definido e o padrão *Round-trip Scenario Test*.

- **Padrões Relacionados:** Um padrão relacionado é o *Controlled Exception Test* [11], que trabalha de forma parecida.
- **Referência:** Este padrão foi descrito por M. Lange e pode ser encontrado em [28].

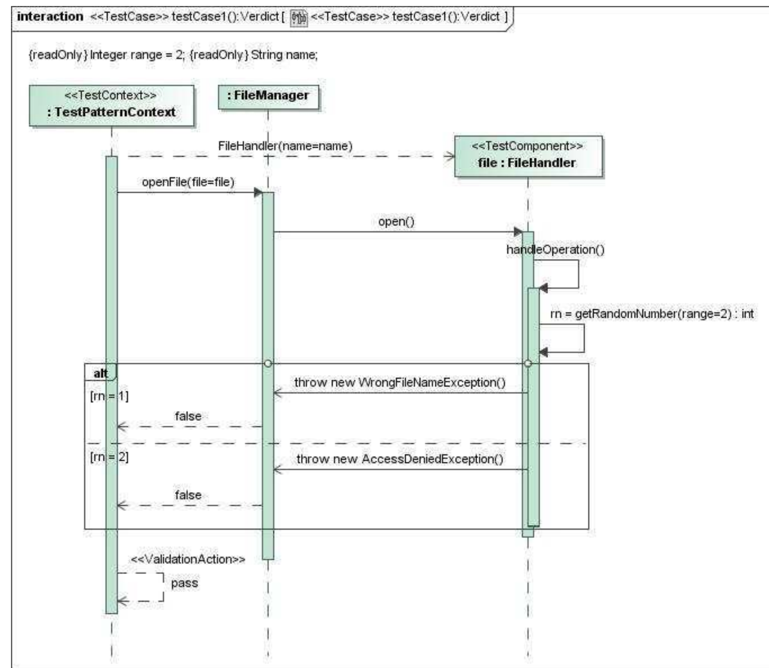


Figura A.4: Diagrama de sequência mostrando os casos de teste para o tratamento de exceções, documentados de acordo com U2TP e aplicando o padrão.

A.4 *Test Case With Time Specification*

Este padrão consiste basicamente na geração de casos de teste para verificar especificações de tempo. Este padrão foi escolhido porque faltas nas interações entre classes pode ferir as restrições de tempo especificadas para o sistema (*deadlocks*, bloqueios, etc). Dessa forma, é importante verificar qual classe (ou classes) é responsável pela falta. Seguindo o formato de definição descrito no Capítulo 3, tem-se:

- **Nome:** *Test Case With Time Specification*.

- **Contexto:** Em algumas situações, é importante verificar especificações de tempo, ou seja, funções que devem ser executadas dentro de um limite de tempo. Este padrão provê uma forma para a definição de casos de teste para especificações de tempo.

- **Intenção:** Desenvolver casos de teste para verificar especificações de tempo.

- **Modelo de Faltas:** Este padrão detecta alterações de código arbitrário que resulta em efeitos colaterais negativos no desempenho do sistema. Dependendo da complexidade do sistema, a detecção de tais alterações pode ser muito importante.

- **Identificação:** O Código A.3, apresenta uma instância do meta-modelo *TestPatternMetamodel*, representada no formato XMI, para este padrão.

  Código Fonte A.3: Elementos que identificam a aplicação do padrão *Test Case With Time Specification* de acordo com o meta-modelo *TestPatternMetamodel*.

```
<?xml version="1.0" encoding="UTF-8"?>
<tpm:Model xmlns:xmi="http://www.omg.org/XMI" xmlns:tpm="TestPatternMetamodel">
    <testPattern name="Test Case With Time Specification">
        <in type="DurationConstraint">
           <patternConstraint>
             <specification type="tpm:OpaqueExpression"
                body="context DurationConstraint inv:
                self.constrainedElement->exists(c | c.receiveEvent.covered.first()
                .represents.type.getAppliedStereotypes()->exists
                (a | a.name = 'ClassToBeIntegrated') or c.sendEvent.covered.first()
                .represents.type.getAppliedStereotypes()->exists
                (a | a.name = 'ClassToBeIntegrated'));"
         language="OCL 2.0"/>
      </patternConstraint>
        </in>
    </testPattern>
</tpm:Model>
```

  Neste Código, é possível perceber quais os elemento(s) do meta-modelo de UML deve(m) ser investigado(s) nos modelos para a aplicação do padrão. De acordo com a especificação do padrão, para que seja possível sua aplicação, basta que existam nos modelos elementos que identifiquem restrições com intervalos de tempo. Logo, para o padrão *Test Case With Time Specification*, restrições de duração nos diagramas de sequência serão investigadas, então, o elemento do meta-modelo a ser casado é *DurationContraint*. Contudo, no contexto de teste de integração, a aplicação deste padrão só é importante se a classe a ser integrada (*ClassToBeIntegrated*) seja restringida pela *DurationConstraint*. Essa restrição pode ser visualizada no código pelo elemento*patternConstraint*.

- **Estratégia:** Adicionar um tempo para o caso de teste. O tempo de sistema atual deve ser armazenado quando o caso de teste inicia. Depois que o caso de teste tiver terminado, o tempo de sistema atual será recuperado novamente. A diferença deve então ser colocada na asserção para a verificação.

- **Exemplos:** Considere o diagrama de sequência mostrado na Figura A.5 abaixo. Neste diagrama é possível perceber a especificação de uma restrição de tempo (elemento *DurationConstraint*) para o caso de uso *Deposit Funds*. Logo, o padrão *Test Case With Time Specification* pode ser perfeitamente aplicado.

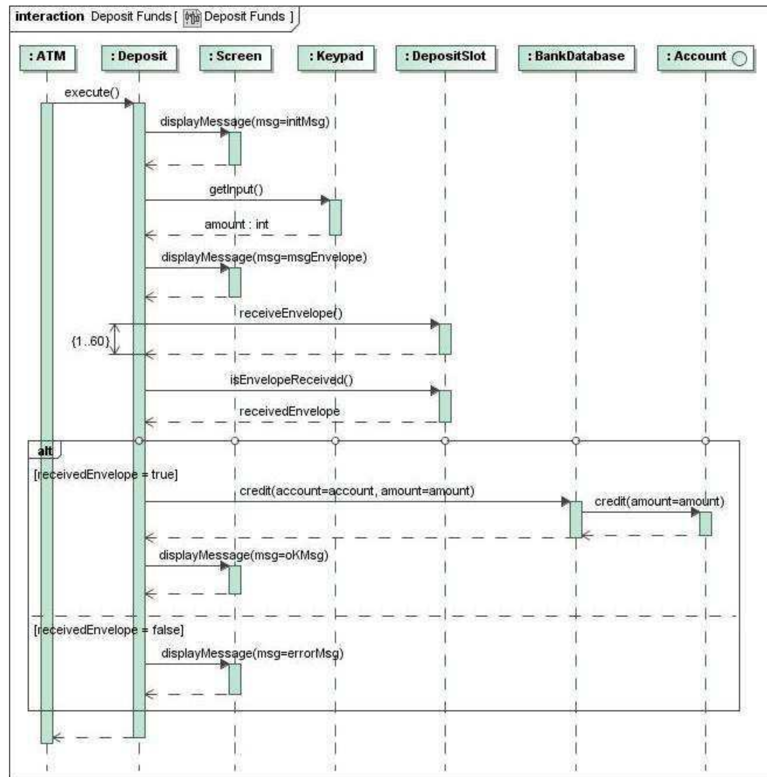


Figura A.5: Diagrama de sequência para o caso de uso *Deposit Funds*.

O caso de teste, que representa o comportamento de teste de acordo com a estratégia definida pelo padrão, é descrito com o diagrama de sequência mostrado na Figura A.6. No diagrama, primeiramente, o componente de teste *Timer* responsável pelo gerenciamento do tempo é iniciado (método *start*). Em seguida, a sequência de mensagens restringida pela *DurationConstraint* é executada (neste caso, a chamada ao método da classe *DepositSlot* estereotipada como SUT). Após a execução, o tempo final é recuperado (método *timeInInteger*) e utilizado para verificação. Se o tempo for menor que o tempo mínimo ou maior que o tempo máximo descrito na restrição (documentados agora como pré-condições do caso de teste), o caso de teste retorna um veredito *fail*, caso contrário, retorna *pass* (representado por uma «*ValidationAction*»).

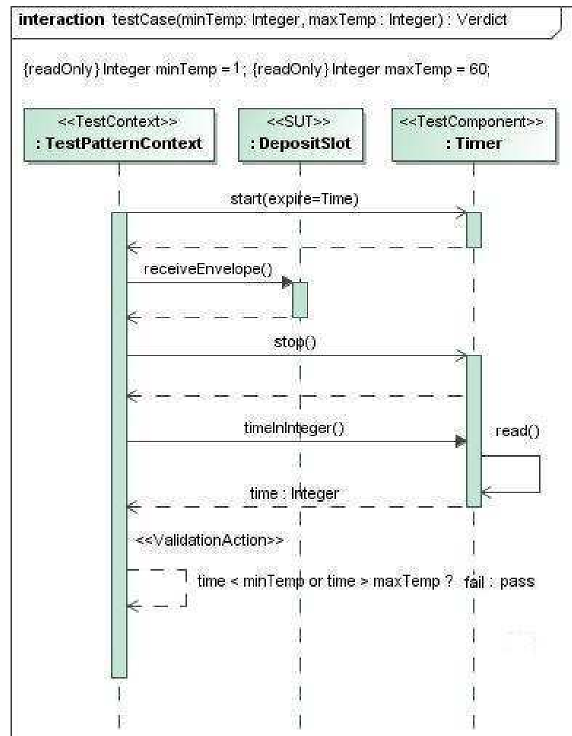


Figura A.6: Diagrama de sequência mostrando o caso de teste, documentado de acordo com U2TP e aplicando o padrão *Test Case With Time Specification*.

- **Padrões Relacionados:** Nenhum.
- **Referência:** Este padrão foi descrito por M. Lange e pode ser encontrado em [28].

Apêndice B

Especificação do *ATM System*

B.1 Descrição do Domínio

Um banco tem diversos terminais ATM (*Automated Teller Machine*) [21], os quais são geograficamente distribuídos e conectados através de uma grande rede a um servidor central. Basicamente, um ATM tem os seguintes componentes de *hardware*: uma leitora de cartões (*card reader*), um emissor de dinheiro (*cash dispenser*), um receptor de envelopes (*deposit slot*), um teclado (*keypad*), uma tela de exibição (*screen*) e uma impressora (*receipt printer*). Usando um ATM, um cliente *customer* pode realizar transações financeiras básicas em contas do tipo cheque (corrente) ou poupança, tais como: consulta de dados, saques, depósitos e transferências entre duas contas. Contudo, antes de realizar qualquer uma destas transações, uma operação de validação/autenticação do cartão e do cliente é realizada.

B.2 Modelo de Casos de Uso: Diagrama de Casos de Uso

A Figura B.1 apresenta quatro casos de uso do *ATM System* realizados pelo cliente (*customer*) representando as transações de consultar os dados de uma conta (*query account*), sacar dinheiro de uma conta (*withdraw funds*), depositar dinheiro numa conta (*deposit funds*), e transferir dinheiro de uma conta para outra (*transfer funds*). Tais casos de uso são dependentes do casos de uso de validação, pois antes realizar qualquer uma destas transações, uma operação de validação/autenticação do usuário (*validate PIN*) deve realizada.

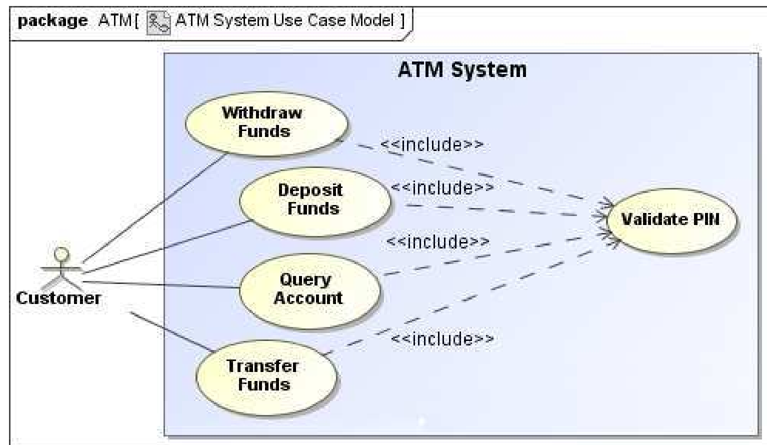


Figura B.1: Casos de uso do *ATM System*.

B.3 Modelo Estrutural: Diagrama de Classes

O modelo estrutural de um sistema pode ser representado por um diagrama de classes UML. A Figura B.2 apresenta um diagrama de classes para o *ATM System*. Neste diagrama de classes estão modeladas as seguintes entidades fundamentais para a representação estrutural do sistema:

- Uma classe *ATM* composta das classes *Screen*, *Keypad*, *CardReader*, *CashDispenser*, *DepositSlot*, e *ReceiptPrinter* (que representam os componentes de *hardware* do sistema) e responsável pela integração entre as classes de interface com o usuário e a lógica do sistema;
- Uma classe *Bank*, a qual representa o servidor do banco que é acessado pelas transações realizadas no terminal ATM;
- Uma classe abstrata *ATMTransaction*, representando uma transação que pode ser realizada pelo caixa automático e que pode ser do tipo *PINValidationTransaction*, *WithdrawTransaction*, *DepositTransaction*, *QueryTransaction*, *TransferTransaction*;
- Uma classe *Card* que provê acesso à classe *Account* que representa uma conta de um cliente (classe *Customer*) e que pode ser do tipo *ChequeAccount* e *SavingsAccount*.

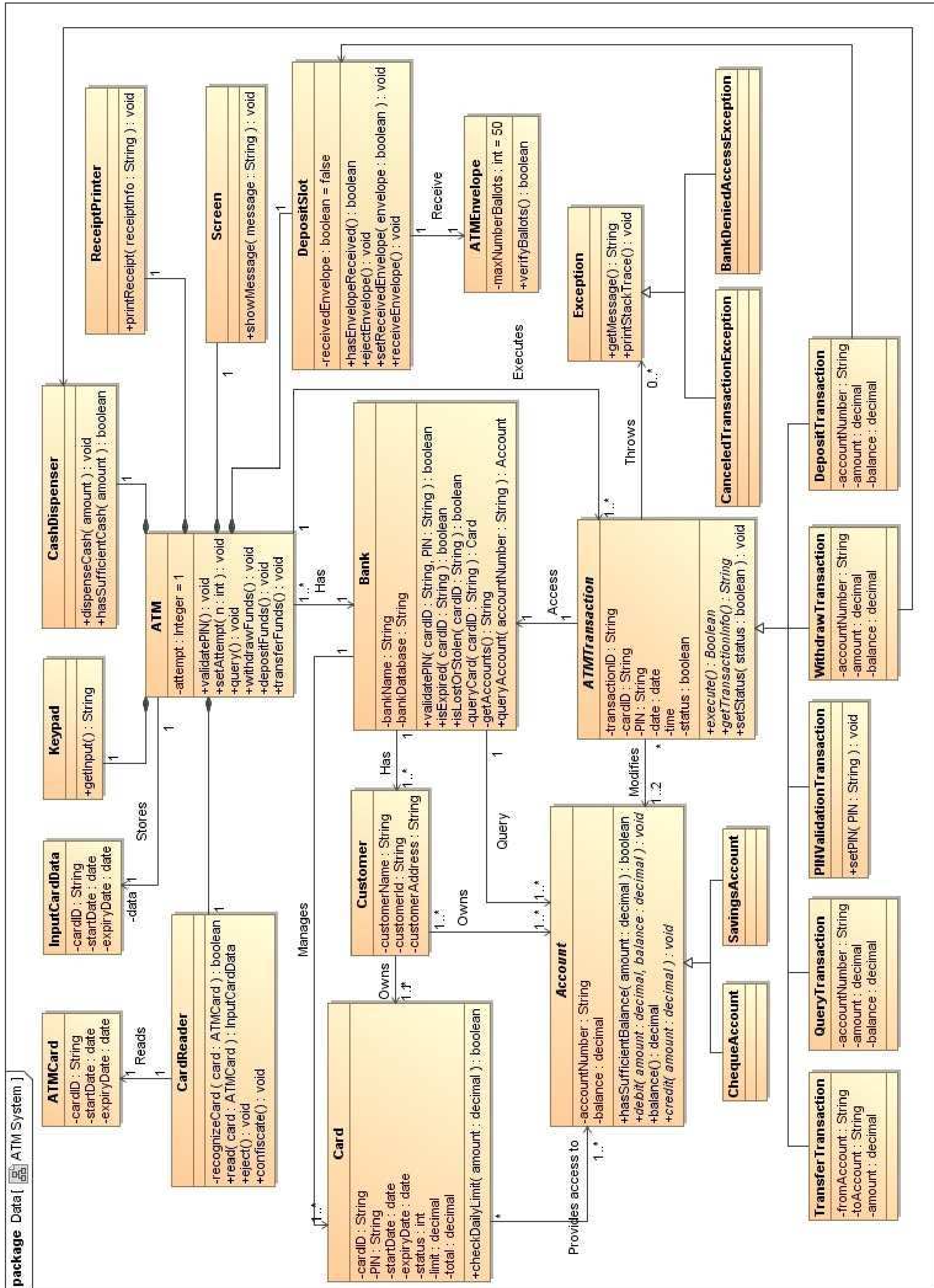


Figura B.2: Diagrama de classes do *ATM System*.

B.4 Modelos Comportamentais: Diagramas de Sequência

O comportamento do sistema pode ser representado por diagramas de sequência UML, os quais são responsáveis por modelar aspectos dinâmicos. As Figuras B.3, B.4, B.5, B.6 e B.7 mostram diagramas de sequência que representam o comportamento do *ATM System* para os casos de uso *query account*, *validate PIN*, *withdraw funds*, *transfer funds* e *deposit funds*, respectivamente. Adicionalmente, um diagrama de sequência, o qual é apresentado na Figura B.8 foi desenvolvido para representar o comportamento o sistema ATM *System* com relação ao lançamento/tratamento de exceções durante a execução de algumas das transações.

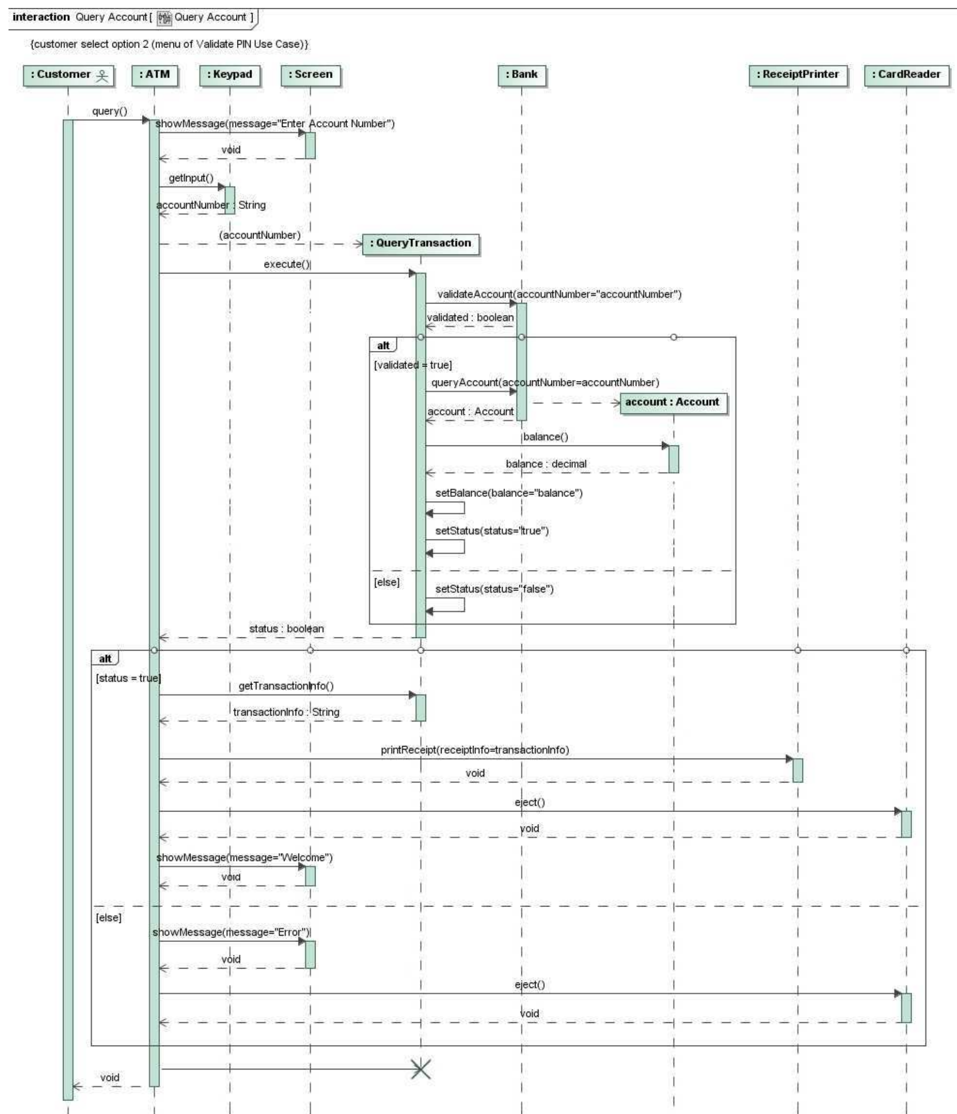


Figura B.3: Diagrama de sequência representando o caso de uso *Query Account*.

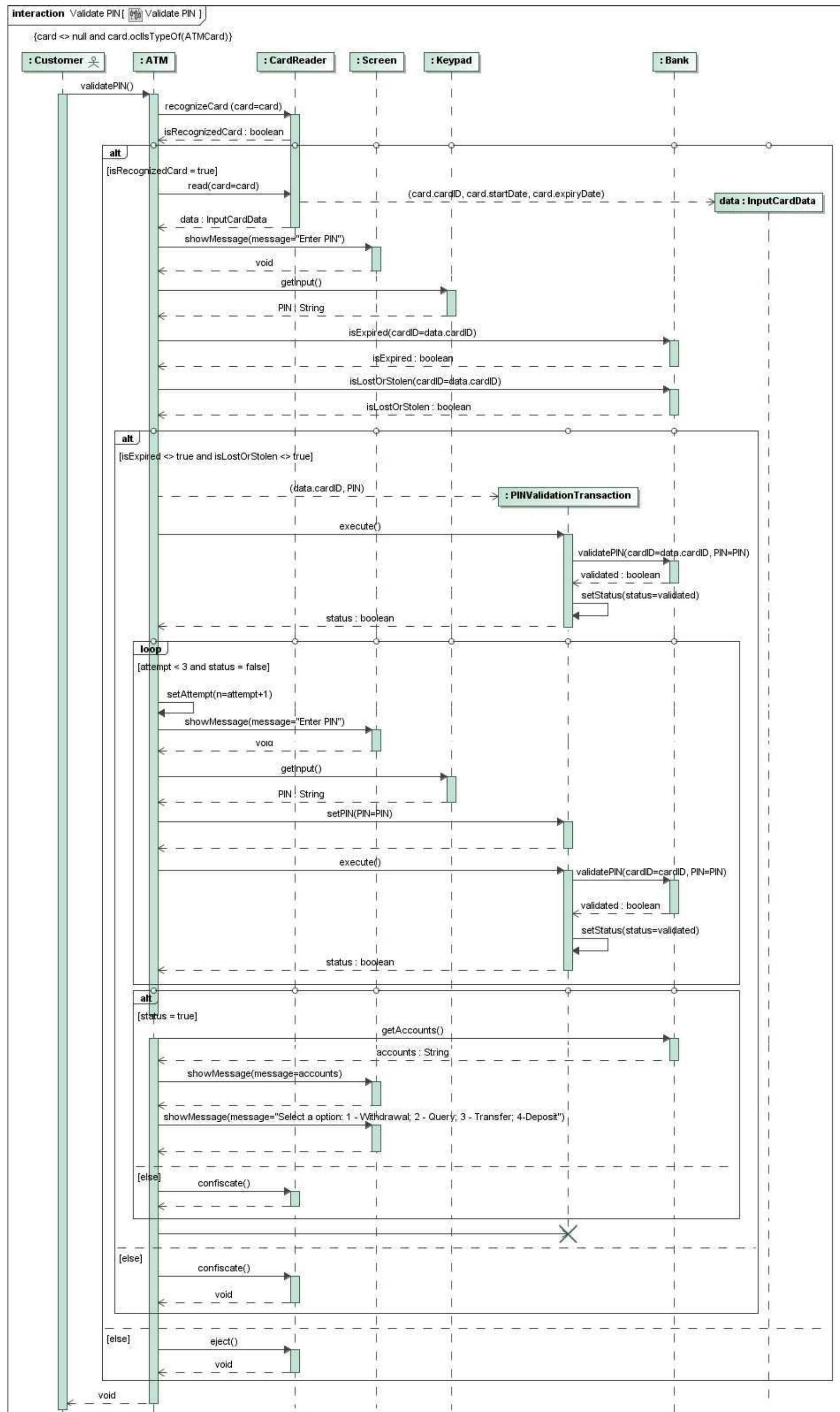


Figura B.4: Diagrama de sequência representando o caso de uso *Validate PIN*.

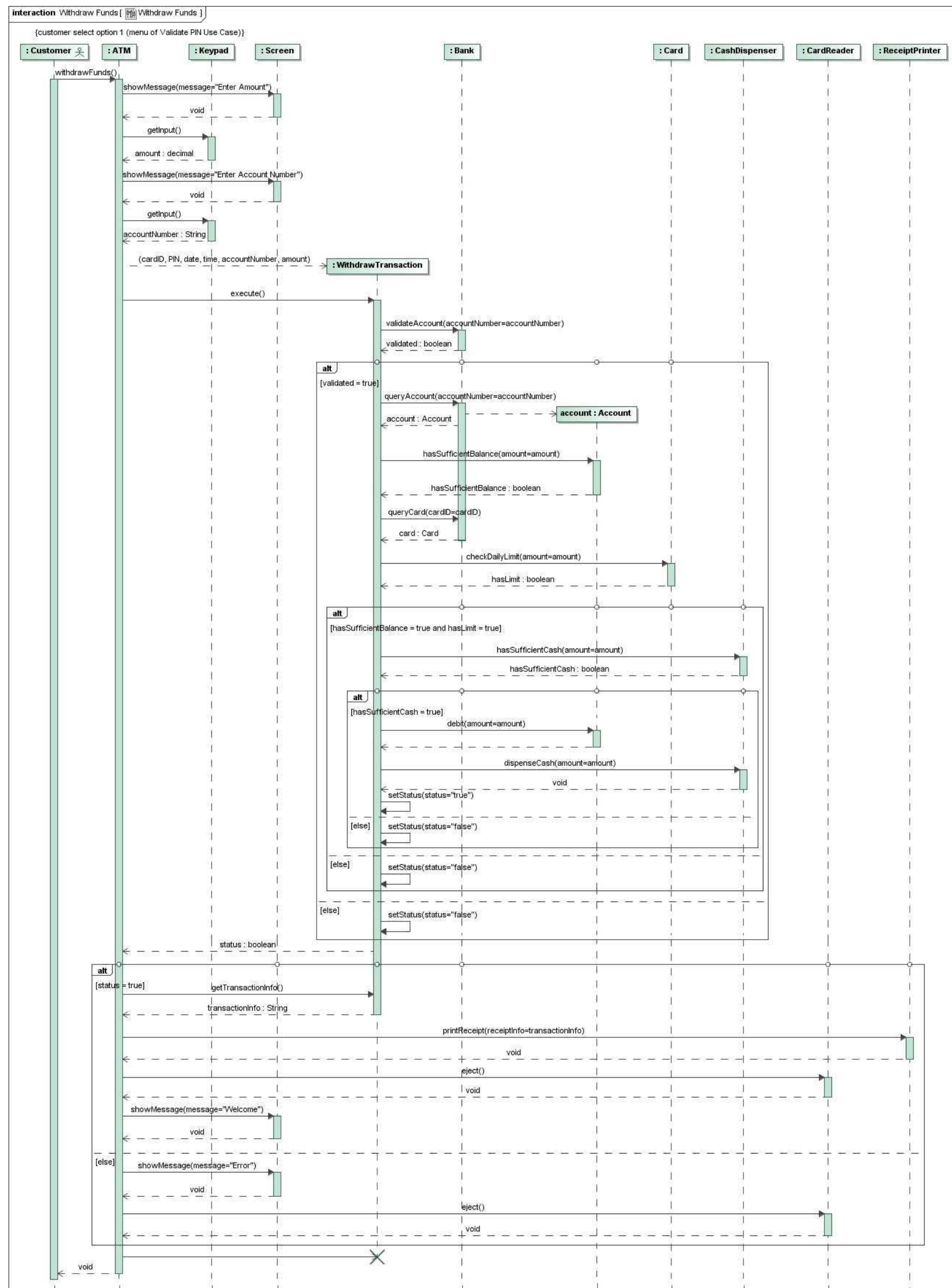


Figura B.5: Diagrama de sequência representando o caso de uso *Withdraw Funds*.

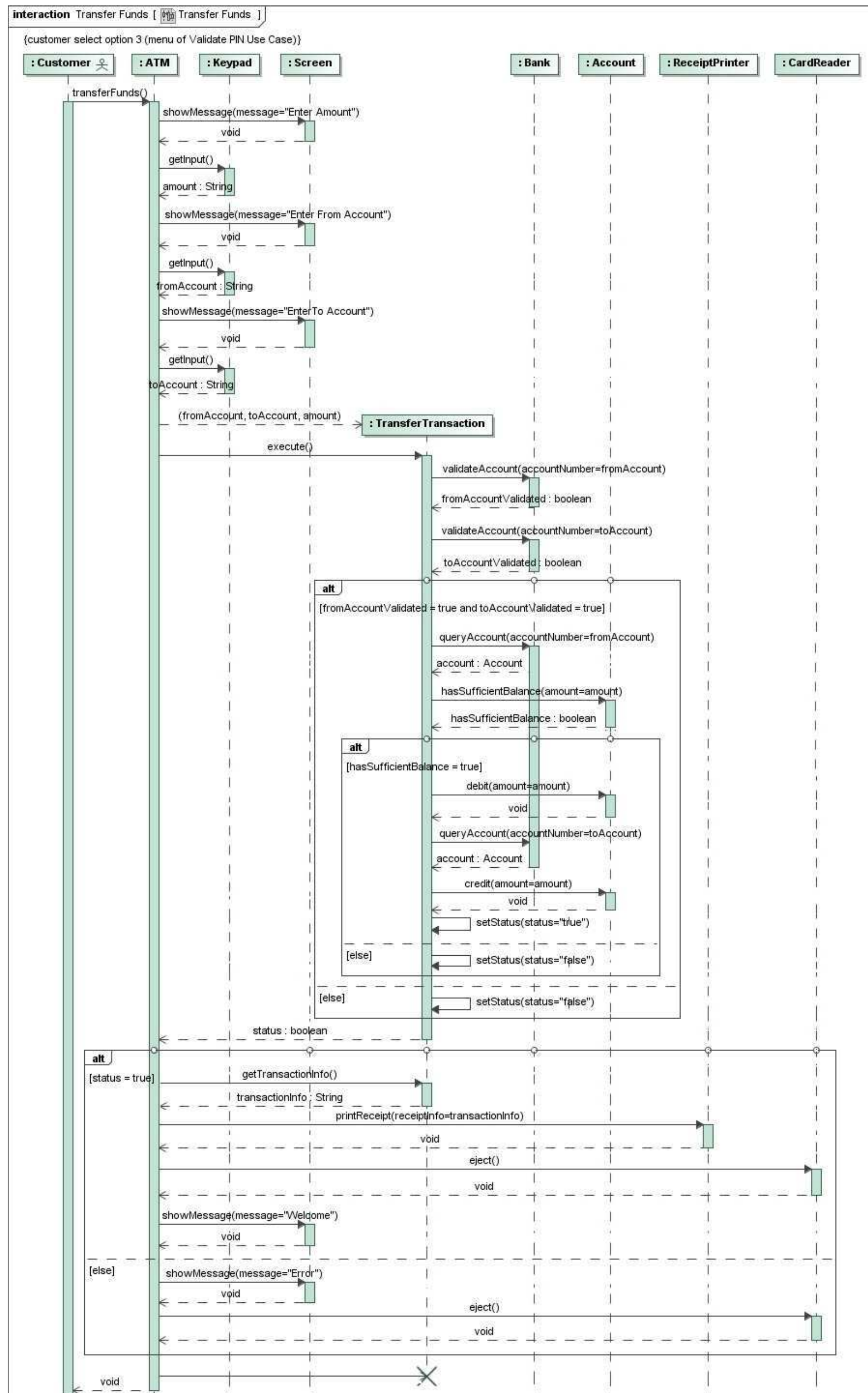


Figura B.6: Diagrama de sequência representando o caso de uso *Transfer Funds*.

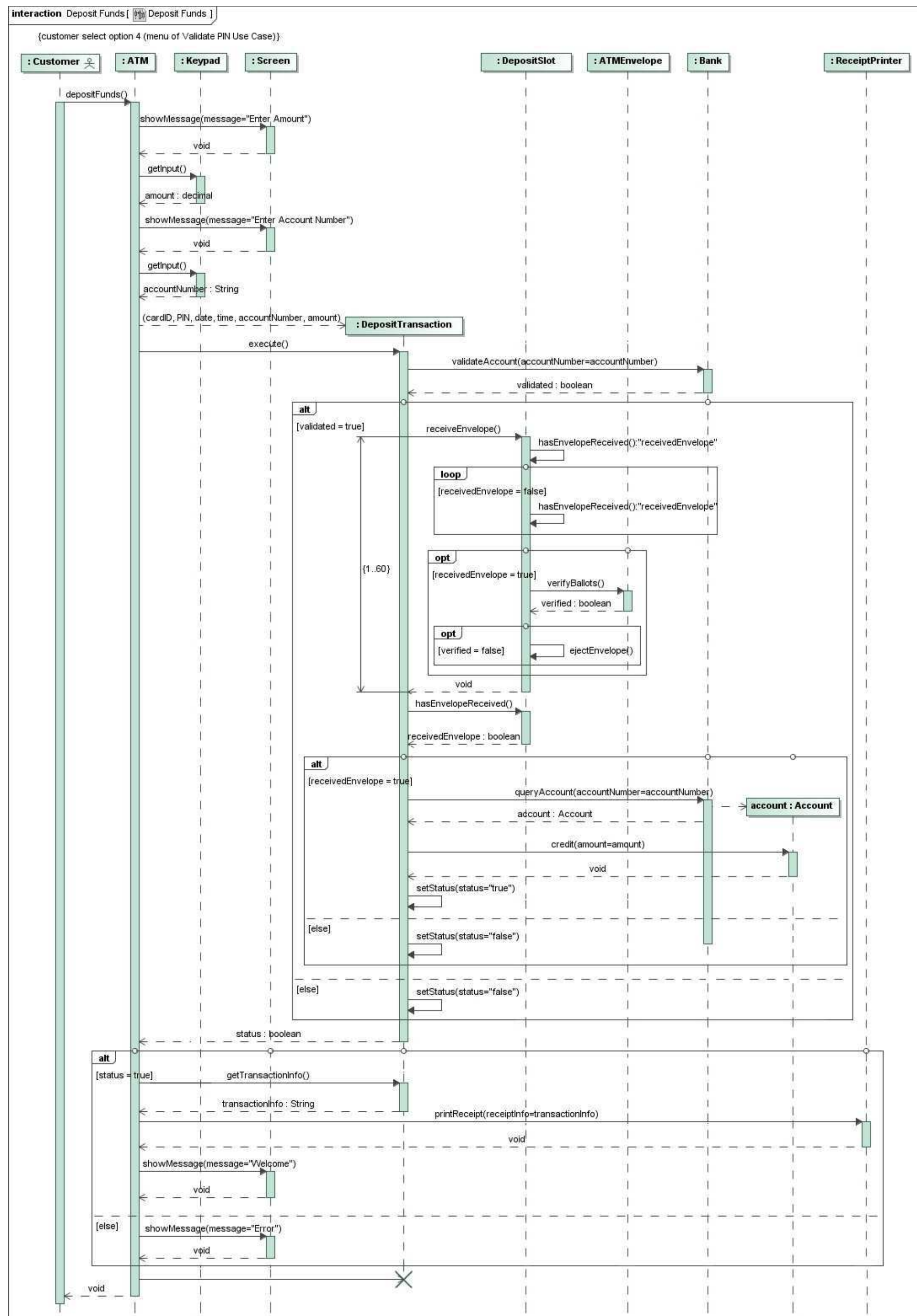


Figura B.7: Diagrama de sequência representando o caso de uso *Deposit Funds*.

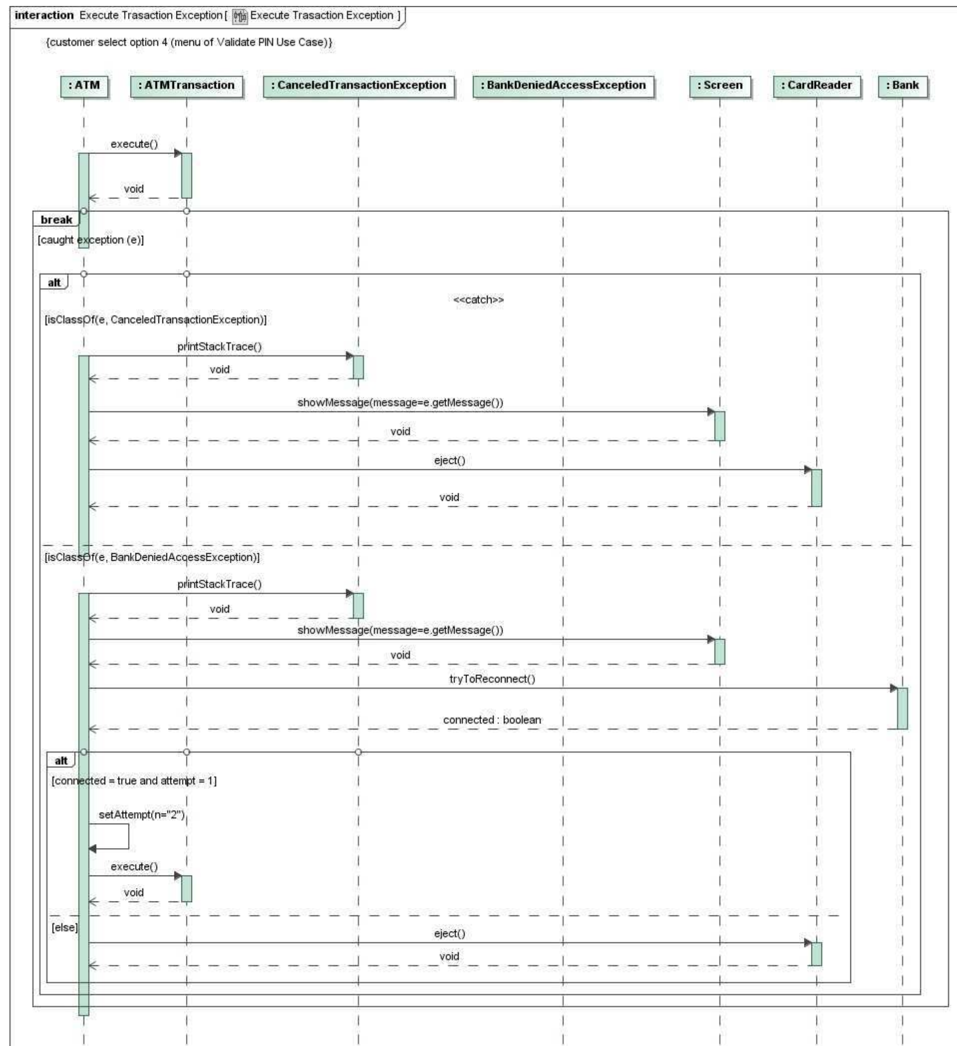


Figura B.8: Diagrama de sequência representando o comportamento do ATM *System* com relação às exceções.

B.5 Cenários de Integração

No contexto de teste de integração, problemas potenciais podem ser encontrados uma vez integradas as classes do sistema. Contudo, dependendo da complexidade do sistema, o número cenários de integração pode ser muito grande. Nesse contexto, foram escolhidos apenas alguns cenários críticos (em um total de 22 cenários) para serem utilizados durante o estudo de caso. Estes cenários foram separados por caso de uso (*use case*), uma vez que cada caso de uso está separado em um projeto diferente na ferramenta que foi utilizada para desenvolver os modelos de desenvolvimento. Alguns

desses cenários são apresentados a seguir (os demais podem ser encontrados no site http://www.dsc.ufcg.edu.br/∼camila/suporttool/casestudy1/scenarios).

1. **Caso de Uso:** *Validate PIN Use Case*

   Descrição do Cenário: Neste cenário, o interesse é em integrar a classe *PINValidationTransaction*, a qual já foi testada individualmente e está disponível para testes. Para tanto, como se trata de uma subclasse de *ATMTransaction*, esta já foi devidamente testada e integrada. As demais classes não estão disponíveis ainda para testes. Logo, as classes que são dependências de *ATMTransaction*, para realizar o teste de integração, são Bank e Account. N. Nesse contexto, os seguintes estereótipos para as classes do sistema devem ser considerados:

   (a) «*NonIntegratedClass*»: *ATM*, *CardReader*, *ATMCard*, *Screen*, *Keypad*, *InputCardData*, *Bank*, *Account*, *ChequeAccount*, *SavingsAccount*, *Card* e *Customer*;

   (b) «*IntegratedClass*»: *ATMTransaction*;

   (c) «*ClassToBeIntegrated*»: *PINValidationTransaction*;

   (d) «*DependingClass*»: *Bank* e *Account* (*ChequeAccount* e *SavingsAccount*).

   Obs.: As classes que estão estereotipadas com *NonIntegratedClass* e não estão anotadas com *DependingClass*, não fazem parte do teste de integração. Adicionalmente, as classes que estão sem nenhum estereótipo também não fazem parte do teste para este cenário.

2. **Caso de Uso:** *Withdraw Funds Use Case*

   Descrição do Cenário: Neste cenário, o interesse é testar a classe ATM. Contudo, nem todas as classes que se relacionam com ATM, direta ou indiretamente, foram integradas. Nesse contexto, os seguintes estereótipos para as classes do sistema devem ser considerados:

   (a) «*NonIntegratedClass*»: *Screen*, *Keypad*, *CashDispenser*, *InputCardData*, *ChequeAccount* e *SavingsAccount*;

   (b) «*IntegratedClass*»: *CardReader*, *ATMCard*, *ReceiptPrinter*, *ATMTransaction*, *WithdrawTransaction*, *Bank*, *Account*, *Customer* e *Card*;

   (c) «*ClassToBeIntegrated*»: ATM;

   (d) «*DependingClass*»: Neste cenário, todas as classes fazem parte do teste de integração. Ou seja, todas as classes que têm relação de dependência com ATM, direta ou indiretamente, devem ser anotadas com este estereótipo. Contudo, só é preciso anotar aquelas que fazem parte do caso de uso em questão.

3. **Caso de Uso:** *Deposit Funds Use Case*

   Descrição do Cenário: Neste é um cenário deseja-se testar as classes envolvidas nas restrições de duração. Como são apenas duas classes (*DepositSlot* e *ATMEnvelope*), tem-se:

   (a) «*NonIntegratedClass*»: *ATMEnvelope*;

   (b) «*ClassToBeIntegrated*»: *DepositSlot*;

   (c) «*DependingClass*»: *ATMEnvelope*

   Obs.: Outro cenário similar é considerar a classe *ATMEnvelope* como o foco do teste.

4. **Caso de Uso:** *Exceptions Use Case*

   Descrição do Cenário: Neste cenário o interesse é testar a classe *ATM*, que realiza o tratamento das exceções lançadas pela classe *ATMTransaction*. Contudo, neste cenário todas as classes já foram integradas. Nesse contexto, os seguintes estereótipos para as classes do sistema devem ser considerados:

   (a) «*NonIntegratedClass*»: Nenhuma;

   (b) «*IntegratedClass*»: Todas as classes, com exceção da classe *ATM*;

   (c) «*ClassToBeIntegrated*»: *ATMTransaction*;

   (d) «*DependingClass*»: Todas as classes, com exceção da classe *ATM*.

Apêndice C

Especificação de Outros Sistemas

C.1 *Object Manager System*

O diagrama de classes para o sistema, conforme mostrado na Figura C.1 possui um total de quatro classes: *System*, *ObjectManager*, *MyObject* e *MyList*. Esta última, é uma classe abstrata. Além desse diagrama, o sistema possui dois diagramas de sequência, *Add Object* e *Remove Object*, conforme mostrado nas Figuras C.2 e C.3, respectivamente, representando casos de uso.

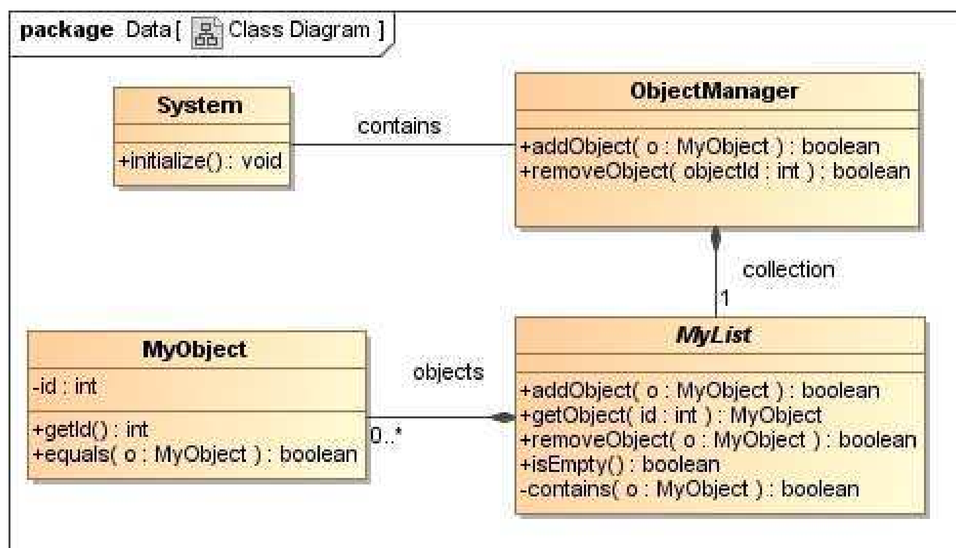


Figura C.1: Diagrama de classes do *Object Manager System*.

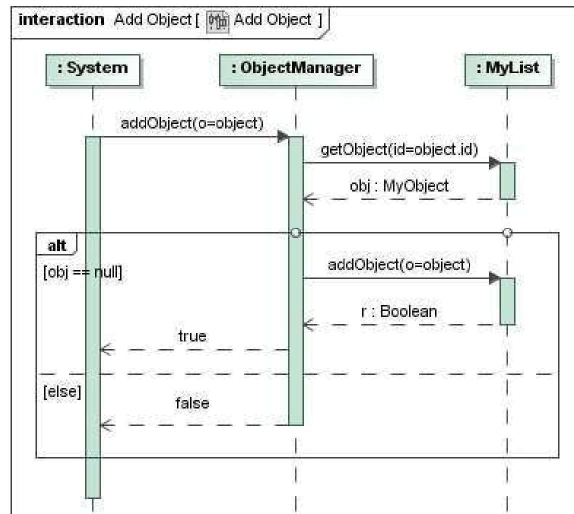


Figura C.2: Diagrama de sequência do *Object Manager System* (*Add Object*).

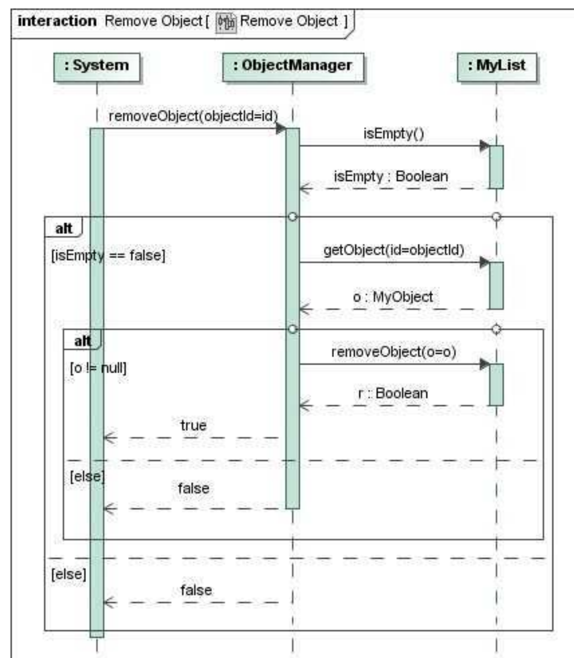


Figura C.3: Diagrama de sequência do *Object Manager System* (*Remove Object*).

C.2 *Simplified ATM System*

O diagrama de classes modelado para o sistema, conforme mostrado na Figura C.4 possui um total de doze classes: *Screen*, *Keypad*, *DepositSlot*, *CashDispenser*, *Account*, *Client*, *ATM*, *BankDatabse*, *Transaction*, *Withdrawal*, *BalanceInquiry* e *Deposit*. As classes *Account* e *Transaction* são classes abstratas. Além desse diagrama, o sistema possui três diagramas de sequência, *Authenticate Client*, *Withdraw Funds* e *Deposit Funds*, conforme mostrado nas

Figuras C.5, C.6 e C.7, respectivamente, representando casos de uso.

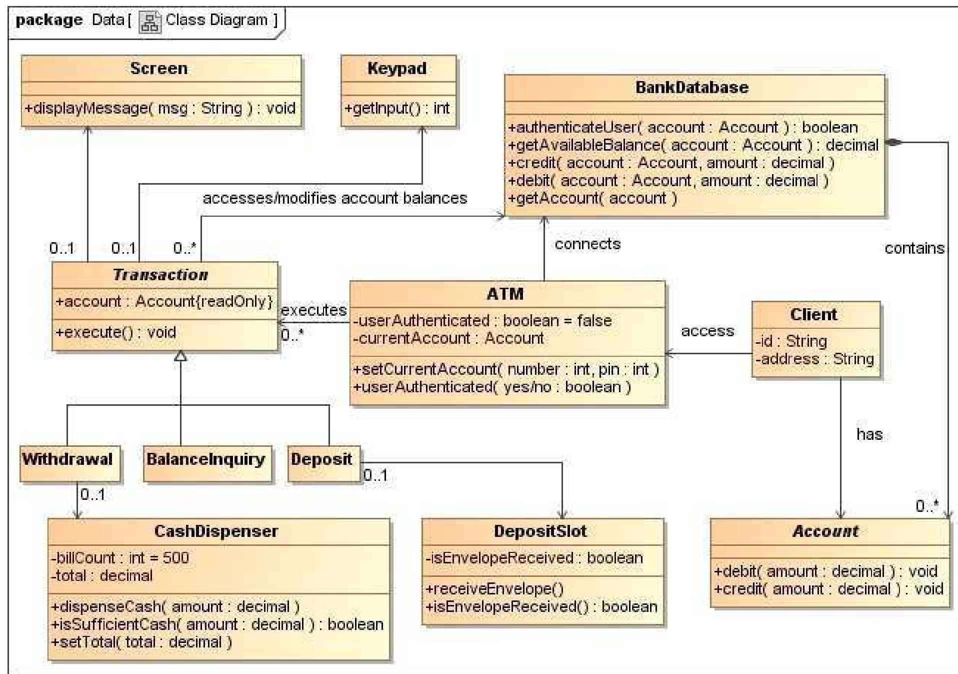


Figura C.4: Diagrama de classes do *Simplified ATM System*.

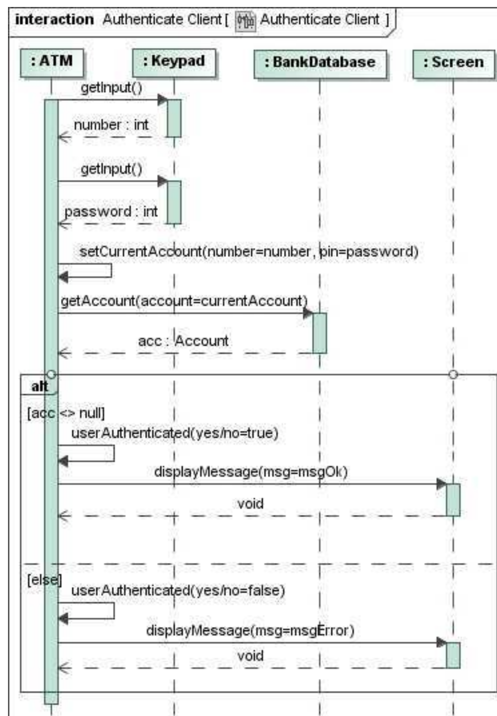


Figura C.5: Diagrama de sequência do *Simplified ATM System* (*Authenticate Client*).

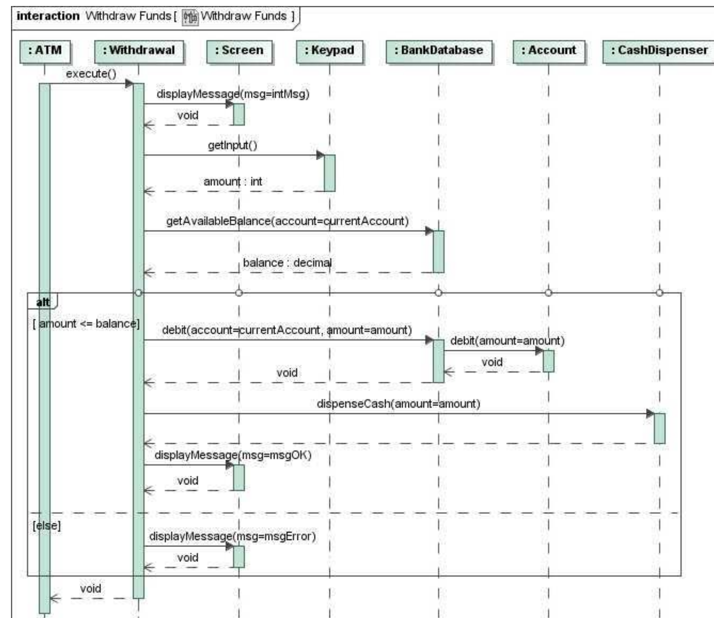


Figura C.6: Diagrama de sequência do *Simplified ATM System* (*Withdraw Funds*).

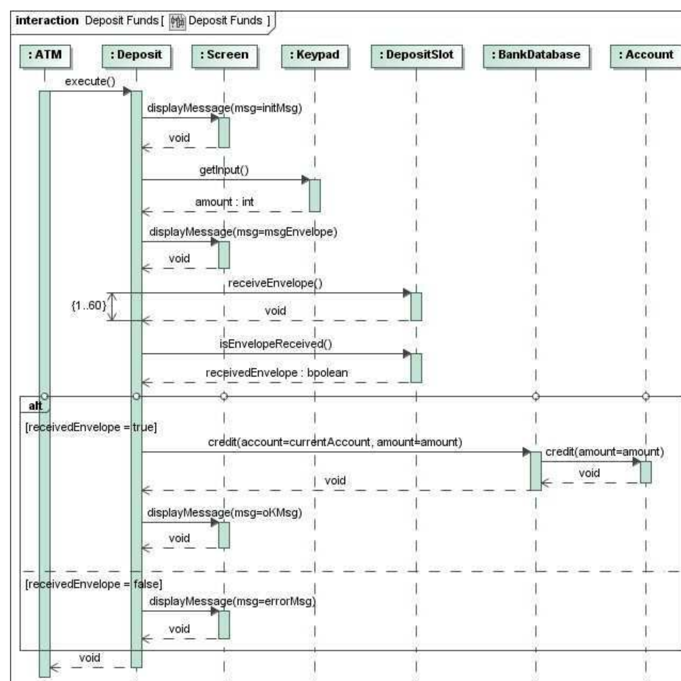


Figura C.7: Diagrama de sequência do *Simplified ATM System*(*Deposit Funds*).

C.3 *File Manager*

O diagrama de classes modelado para o sistema, conforme mostrado na Figura C.8 possui um total de seis classes, são elas: *System*, *FileManager*, *File*, *Exception*, *WrongFileNameException* e *AccessDeniedException*. Apesar de não ser mostrado na figura, o método *open()* da classe *File* lança as duas exceções *WrongFileNameException* e *AccessDeniedException*. Essa característica pode ser identificada, no XMI, pelo campo *raisedException*. Além desse diagrama, o sistema possui um diagrama de sequência, *Open File*, conforme mostrado na Figura C.9.

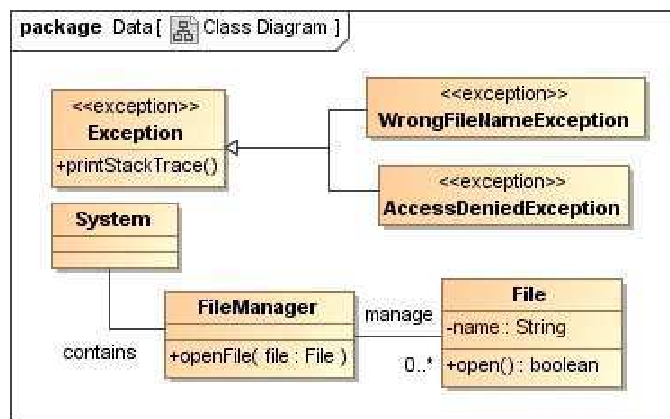


Figura C.8: Diagrama de classes do *File Manager*.

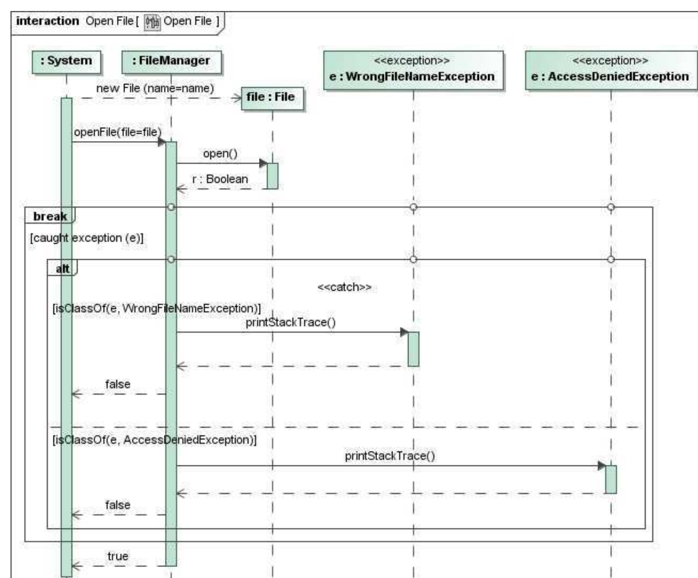


Figura C.9: Diagrama de sequência do *File Manager* (*Open File*).

Apêndice D

Resultados do Primeiro Estudo de Caso

Este Apêndice mostra, detalhadamente, os resultados obtidos com o estudo de caso realizado para avaliar a abordagem proposta nesse trabalho, incluindo os dados coletados para o cálculo dos valores das métricas e alguns dos casos de teste gerados. As Seções D.1, D.2 e D.2 apresentam os dados obtidos pelos primeiro, segundo e terceiro participantes, respectivamente.

D.1 Resultados do Primeiro Participante

Para uma melhor compreensão e interpretação dos dados coletados, foi pedido que cada participante registrasse, durante o estudo de caso, o tempo gasto e o número de casos de teste gerados por caso de uso (representados por um diagrama de sequência), para a obtenção dos valores das métricas com relação ao esforço. Após a execução do estudo, os casos de teste gerados foram analisados, um a um, para a identificação de possíveis casos de teste inválidos, e assim obter os valores da métrica com relação à corretude. Relembrando, um caso de teste é considerado inválido se: (i) é sintaticamente incorreto, de acordo com a modelagem/especificação do sistema; (ii) é semanticamente inconsistente com a aplicação, ou seja, denota um comportamento de sucesso ou falha que não condiz com o esperado pelos modelos; (iii) não pode ser executado, por estar incompletamente definido, ou por não cobrir a especificação do padrão; e (iv) não representa o cenário de integração proposto.

A Tabela D.1 mostra os dados coletados após a execução do estudo pelo primeiro participante, para o cálculo das métricas com relação ao esforço e à corretude. Nesta tabela

são apresentados o tempo gasto, em minutos, na aplicação da abordagem para a geração dos casos de teste, o número total de casos de teste gerados, o número total de casos de teste válidos e os padrões de teste aplicados (RTST - *Round-trip Scenario Test*, ACT - *Abstract Class Test*, TCWTS - *Test Case With Time Specification* e ES - *Error Simulator*), para cada caso de uso do sistema.

Tabela D.1: Dados coletados na execução do estudo (participante 1).

Caso de Uso	**Tempo Gasto (min)**	**CTs Gerados**	**CTs Válidos**	**Padrões**
Validate PIN	133 min	16	8	RTST
Withdraw Funds	133 min	11	9	RTST
Query Account	71 min	6	4	RTST
Transfer Funds	72 min	12	9	RTST
Deposit Funds	103 min	17	13	RTST, TCWTS
Exceptions	32 min	3	3	ES

Analisando os dados da tabela D.1, tem-se que o participante gerou um total de sessenta e cinco casos de teste, em um tempo total de quinhentos e quarenta e quatro minutos (nove horas e quatro minutos). Além disso, do total de casos de teste gerados, apenas quarenta e seis casos de teste foram considerados válidos, sendo dezenove considerados inválidos e descartados para a obtenção da métrica relacionada à eficácia. A Figura D.1 mostra um dos casos de teste gerados considerados inválidos. Este foi considerado inválido primeiramente pelo fato de não representar nenhum dos cenários de integração proposto (a classe *Bank* como SUT). É importante ressaltar que o fato de um caso de teste não representar um dos cenários propostos, não implica que o caso de teste não poderá nunca ser utilizado, ele apenas foi descartado nesse estudo de caso. Contudo, não só esse problema fez com que o caso de teste fosse considerado inválido, os seguintes defeitos (conforme pode ser visualizado em destaque na figura) também foram encontrados:

1. **Aplicação errada do estereótipo «*TestCompontent*»:** o estereótipo está aplicado a uma classe que já está integrada;

2. **Mensagem errada:** a primeira mensagem não era para existir, isso faz com que o caso de teste não possa ser executado corretamente;

3. **Veredito especificado incorretamente:** o veredito, representado pela última mensagem do diagrama, deve ser especificado na forma de uma auto-delegação ao objeto *TestContext* e não da forma que foi apresentado.

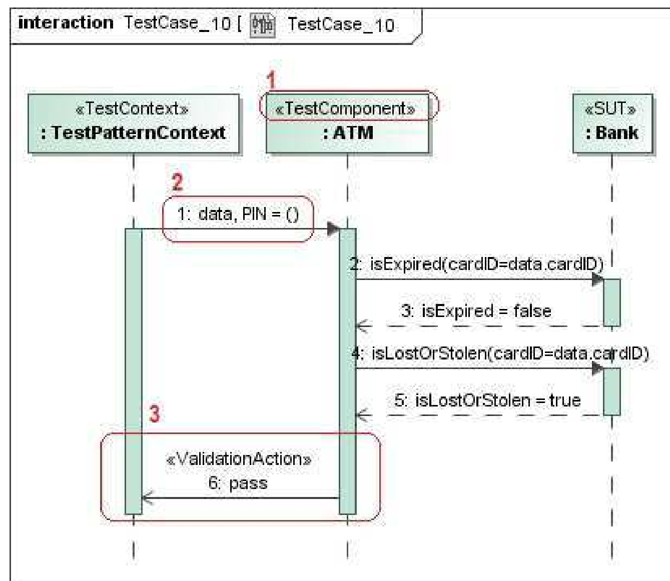


Figura D.1: Caso de teste inválido gerado pelo participante 1.

Para a medição da capacidade de detecção de defeitos dos casos de teste foram escolhidos nove tipos de defeitos de integração existentes indicados por Binder [11]. São eles: [11]:

1. Falta, sobreposição ou conflito de funcionalidades;
2. Tratamento de erros (exceções) incorreto;
3. Chamadas a funções erradas devido a erro no código ou exceções inesperadas;
4. Mensagens enviadas de uma classe cliente que violam pré-condições da classe servidora;
5. Mensagens enviadas de uma classe cliente que violam restrições seqüenciais da classe servidora;
6. Objetos incorretos associados a mensagens;
7. Parâmetros errados ou valores de parâmetros incorretos;

8. Conflitos entre componentes (por exemplo, colisão de *threads*);

9. Bloqueio de funções/tarefas.

A maioria dos padrões de teste, incluindo os considerados nesse trabalho, têm um modelo de faltas associado. Esse modelo de faltas descreve os possíveis tipos de defeitos que poderiam ser encontrados pelos casos de teste gerados de acordo com a estratégia do padrão. Cada um dos padrões utilizados pela abordagem são capazes de detectar algum dos tipos de defeitos acima listados. Fazendo um mapeamento entre os tipos de defeitos e os modelos de faltas de cada padrão tem-se:

- ***Round-trip Scenario Test*:** Um cenário é um caminho no diagrama de sequência. Cada interface de um objeto que participa do cenário deve ser fisicamente correta e deve prover uma implementação correta de suas funcionalidades. Nesse contexto, os defeitos de integração 1, 3, 4, 5, 6 e 7 listados acima, se existirem, serão capturados por pelo menos um caminho *round-trip* gerado, ou seja, por pelo menos um dos casos de teste gerados para o cenário.

- ***Abstract Class Test*:** Uma vez que esse padrão reusa o padrão *Round-trip Scenario Test*, na intenção de gerar casos de teste para verificar as interações entre as classes do sistema com classes abstratas (ou interfaces), e consequentemente as sub-classes existentes, todos os defeitos de integração listados no item anterior, se existirem, também serão capturados pelos casos de teste gerados com esse padrão.

- ***Error Simulator*:** Este padrão tem o intuito de testar o tratamento de exceções de um sistema. Várias faltas podem acontecer relacionadas ao tratamento de exceções, como por exemplo, exceções inapropriadas passadas de um servidor a um cliente, exceções propagadas fora do escopo, tratadores de exceção que criam efeitos colaterais indesejados, entre outros. Nesse contexto, defeito de integração 2 (tratamento de erros (exceções) incorretos), se existir, será capturado pelos casos de teste gerados com esse padrão.

- ***Test Case With Time Specification*:** Este padrão detecta alterações de código que resulta em efeitos colaterais negativos no desempenho do sistema. Dentre os defeitos

que podem ser detectados que implicam no desempenho são, por exemplo, o conflito de componentes e o bloqueio de funções/tarefas (defeitos de integração 8 e 9).

No intuito de avaliar a eficácia da abordagem realizada no estudo pelo participante, os casos de teste foram agrupados por cenário de integração especificado e por padrão de teste, conforme mostrado na Tabela D.2. Nesta tabela, em cada coluna é mostrado o número de casos de teste gerados com relação ao número mínimo de casos de teste que deveriam ter sido construídos (para garantir a cobertura). É possível perceber, com esse dados, que um dos padrões, o *Abstract Class Test* não foi aplicado, embora elementos dos modelos indicassem que sua aplicação era possível. Além disso, para muitos dos cenários desenvolvidos não foram construídos todos os casos de teste necessários. No total, foram contabilizados trinta e nove casos de teste de acordo com o padrão *Round-trip Scenario Test*, quatro de acordo com o padrão *Test Case With Time Specification* e três de acordo com o padrão *Error Simulator*. Após esse agrupamento, os casos de teste foram analisados com respeito aos tipos de defeitos que poderiam detectar.

A análise dos casos de teste de acordo com cada padrão de teste foi realizada da seguinte forma:

- ***Round-trip Scenario Test***: Conforme mostrado no Capítulo 5, com a aplicação deste padrão, seis tipos de defeitos de integração, dentre os nove listados, poderiam ser detectados por pelo menos um caso de teste gerado para um caminho do diagrama de sequência. Contudo, para garantir isso, é necessário que os casos de teste gerados, para um diagrama, cubram todos os caminhos. Analisando os casos de teste gerados por cenário de integração constatou-se que foram gerados casos de teste que cobrem todos os caminhos para nove cenários. Para os demais cenários, alguns (ou todos) casos de teste gerados foram inválidos ou não foram construídos. Dessa forma, para esses cenários, embora os casos de teste que foram gerados possam capturar esses tipos de defeitos, não é possível garantir que eles seriam necessariamente capturados por esse conjunto de casos de teste, uma vez que ele não está completo. Logo, foi contabilizado pelo menos um defeito para cada um dos seis tipos identificados para esses nove cenários, totalizando em cinquenta e quatro defeitos;

- ***Abstract Class Test***: Como este padrão não foi aplicado, os tipos de defeitos de

Tabela D.2: Casos de teste válidos por cenário de acordo com os padrões de teste (participante 1).

Casos de Uso e Cenários		**RTST**	**ACT**	**TCWTS**	**ES**
Validate PIN	Cenário 2	2 de 2	-	-	-
	Cenário 2	2 de 3	-	-	-
	Cenário 3	2 de 4	-	-	-
	Cenário 4	2 de 4	-	-	-
Withdraw Funds	Cenário 5	4 de 4	-	-	-
	Cenário 6	1 de 1	-	-	-
	Cenário 7	2 de 4	-	-	-
	Cenário 8	2 de 4	-	-	-
Query Account	Cenário 9	2 de 2	-	-	-
	Cenário 10	0 de 1	-	-	-
	Cenário 11	0 de 2	-	-	-
	Cenário 12	2 de 2	-	-	-
Transfer Funds	Cenário 13	3 de 3	-	-	-
	Cenário 14	1 de 1	-	-	-
	Cenário 15	2 de 3	-	-	-
	Cenário 16	3 de 3	-	-	-
Deposit Funds	Cenário 17	4 de 6	-	-	-
	Cenário 18	1 de 1	-	-	-
	Cenário 19	0 de 6	-	-	-
	Cenário 20	4 de 6	-	-	-
	Cenário 21	-	-	4 de 4	-
Exceptions	Cenário 22	-	-	-	3 de 3

integração que poderiam ser capturados pelos casos de teste gerados pelo padrão não foram contabilizados;

- ***Test Case With Time Specification*:** Conforme mostrado no Capítulo 5, este padrão detectaria defeitos que implicam no desempenho relacionados ao conflito de componentes e o bloqueio de funções/tarefas (defeitos de integração 8 e 9). Como este

padrão foi corretamente aplicado, os quatro casos de teste gerados pelo participante poderiam capturar os defeitos, totalizando em oito defeitos;

- ***Error Simulator*:** Conforme mostrado no Capítulo 5, este padrão detectaria defeitos de integração do tipo 2 (tratamento de erros/xceções incorreto). Como o padrão foi corretamente aplicado, os três casos de teste gerados pelo participante poderiam capturar o defeito, totalizando em três defeitos.

A Tabela D.3 apresenta os possíveis defeitos de integração que podem ser detectados pelos casos de teste gerados de acordo com a análise feita.

Tabela D.3: Tipos de defeitos de integração que poderiam ser detectados pelos casos de teste (participante 1).

Padrão de Teste	**Número de Defeitos por Tipo**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Round-trip Scenario Test	9		9	9	9	9	9	-	-
Abstract Class Test	0	-	0	0	0	0	0	-	-
Error Simulator	-	3	-	-	-	-	-	-	-
Test Case With Time Specification	-	-	-	-	-	-	-	4	4
Total	9	3	9	9	9	9	9	4	4

D.2 Resultados do Segundo Participante

A Tabela D.4 mostra os dados coletados após a execução do estudo pelo segundo participante, para o cálculo das métricas com relação ao esforço e à corretude. Nesta tabela são apresentados o tempo gasto, em minutos, na aplicação da abordagem para a geração dos casos de teste, o número total de casos de teste gerados, o número total de casos de teste válidos e os padrões de teste aplicados, para cada caso de uso do sistema.

Analisando os dados da tabela D.4, tem-se que, o participante gerou um total de setenta e três casos de teste, em um tempo total de hum mil e quarenta e cinco minutos (dezessete horas e vinte e cinco minutos). Além disso, do total de casos de teste gerados, sessenta e dois casos de teste foram considerados válidos, sendo onze inválidos e descartados para

Tabela D.4: Dados coletados na execução do estudo (participante 2).

Caso de Uso	Tempo Gasto (min)	CTs Gerados	CTs Válidos	Padrões
Validate PIN	259 min	11	5	RTST
Withdraw Funds	207 min	17	17	RTST, ACT
Query Account	142 min	10	10	RTST, ACT
Transfer Funds	158 min	16	16	RTST, ACT
Deposit Funds	217 min	16	14	RTST, TCWTS
Exceptions	62 min	3	0	ES

a obtenção da métrica relacionada à eficácia. A Figura D.2 mostra um dos casos de teste gerados considerados inválidos. Este foi considerado inválido primeiramente pelo fato ser sintaticamente incorreto, de acordo com a modelagem/especificação do sistema (conforme pode ser visualizado em destaque na figura, indicado pelo número 3). Além disso, os seguintes defeitos também foram encontrados:

1. **Anotação de classes com os estereótipos de integração:** nos modelos de teste, as classes do sistema ainda estão anotadas com os estereótipos do perfil IOP, o que não devia acontecer, conforme a especificação da abordagem;

2. **Falta de emuladores:** as classes que não foram integradas, de acordo com o cenário de integração, não foram substituídas por seus respectivos emuladores, conforme a especificação da abordagem.

Para a medição da capacidade de detecção de defeitos dos casos de teste gerados pelo participante, no intuito de avaliar a eficácia da abordagem realizada no estudo, os casos de teste foram agrupados por cenário de integração especificado e por padrão de teste, conforme mostrado na Tabela D.5. Nesta tabela, em cada coluna é mostrado o número de casos de teste gerados com relação ao número mínimo de casos de teste que deveriam ter sido construídos (para garantir a cobertura).

É possível perceber, com esse dados, que o padrão *Error Simulator* não foi aplicado corretamente, por isso os casos de teste foram considerados inválidos. Além disso, para muitos dos cenários desenvolvidos não foram construídos todos os casos de teste necessários. No total, foram contabilizados trinta e oito casos de teste de acordo com o padrão *Round-trip*

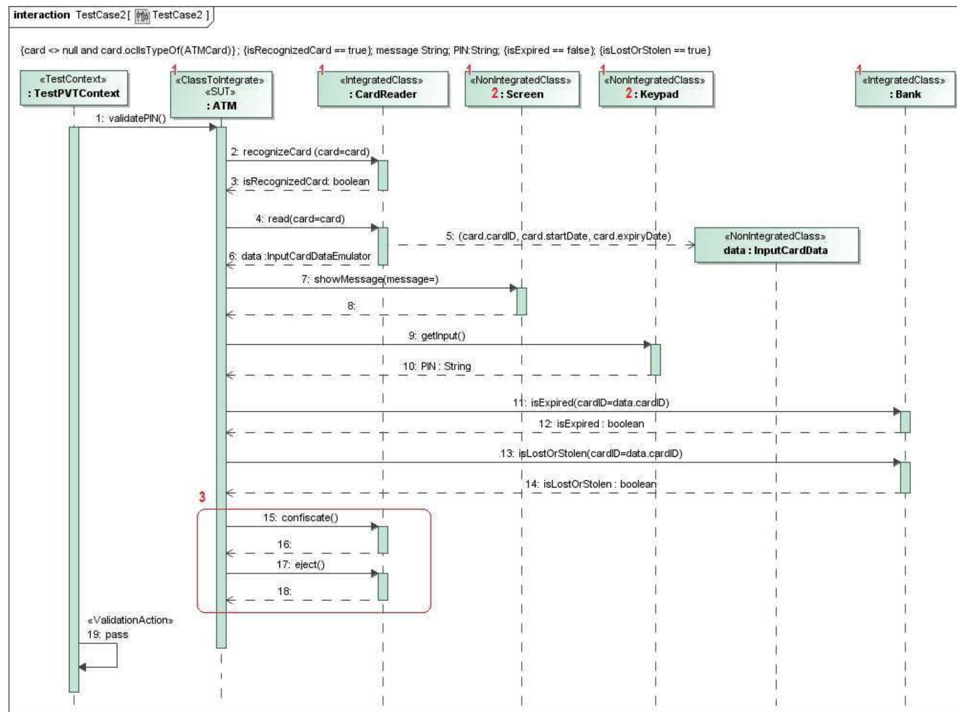


Figura D.2: Caso de teste inválido gerado pelo participante 2.

Scenario Test, vinte e quatro de acordo com o padrão *Abstract Class Test* e três de acordo com o padrão *Test Case With Time Specification*. Após esse agrupamento, os casos de teste foram analisados com respeito aos tipos de defeitos que poderiam detectar. A seguir, é apresentada a análise dos casos de teste de acordo com cada padrão de teste:

- ***Round-trip Scenario Test*:** Analisando os casos de teste gerados por cenário de integração constatou-se que foram gerados casos de teste que cobrem todos os caminhos para dez cenários. Para os demais cenários, alguns (ou todos) casos de teste gerados foram inválidos ou não foram construídos. Dessa forma, para esses cenários, embora os casos de teste que foram gerados possam capturar esses tipos de defeitos não é possível garantir que eles seriam capturados, uma vez que o conjunto de casos de teste não está completo. Logo, foi contabilizado pelo menos um defeito para cada um dos seis tipos identificados para esses dez cenários, totalizando em sessenta defeitos;

- ***Abstract Class Test*:** Analisando os casos de teste gerados por cenário de integração constatou-se que foram gerados casos de teste que cobrem todos os caminhos que envolvem classes abstratas para seis cenários. Logo, foi contabilizado pelo menos um

Tabela D.5: Casos de teste válidos por cenário de acordo com os padrões de teste (participante 2).

Casos de Uso e Cenários		**RTST**	**ACT**	**TCWTS**	**ES**
Validate PIN	Cenário 1	1 de 2	-	-	-
	Cenário 2	1 de 3	-	-	-
	Cenário 3	2 de 4	-	-	-
	Cenário 4	1 de 4	-	-	-
Withdraw Funds	Cenário 5	3 de 4	3 de 3	-	-
	Cenário 6	1 de 1	-	-	-
	Cenário 7	3 de 4	2 de 3	-	-
	Cenário 8	3 de 4	2 de 6	-	-
Query Account	Cenário 9	2 de 2	1 de 1	-	-
	Cenário 10	1 de 1	-	-	-
	Cenário 11	2 de 2	1 de 1	-	-
	Cenário 12	2 de 2	1 de 2	-	-
Transfer Funds	Cenário 13	3 de 3	2 de 2	-	-
	Cenário 14	1 de 1	-	-	-
	Cenário 15	3 de 3	2 de 2	-	-
	Cenário 16	3 de 3	2 de 4	-	-
Deposit Funds	Cenário 17	1 de 6	1 de 3	-	-
	Cenário 18	1 de 1	-	-	-
	Cenário 19	2 de 6	2 de 3	-	-
	Cenário 20	2 de 6	2 de 6	-	-
	Cenário 21	-	-	3 de 4	-
Exceptions	Cenário 22	-	-	-	0 de 3

defeito para cada um dos seis tipos identificados para esses seis cenários, totalizando em trinta e seis defeitos;

- ***Test Case With Time Specification*:** Como este padrão foi corretamente aplicado, os três casos de teste gerados pelo participante poderiam capturar os defeitos relacionados ao conflito de componentes e o bloqueio de funções/tarefas (8 e 9), totalizando em seis

defeitos;

- ***Error Simulator*:** Como este padrão não foi corretamente aplicado, os tipos de defeitos de integração que poderiam ser capturados pelos casos de teste gerados pelo padrão não foram contabilizados.

A Tabela D.6 apresenta os possíveis defeitos de integração que podem ser detectados pelos casos de teste gerados de acordo com a análise feita.

Tabela D.6: Tipos de defeitos de integração que poderiam ser detectados pelos casos de teste (participante 2).

Padrão de Teste	**Número de Defeitos por Tipo**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Round-trip Scenario Test	10	-	10	10	10	10	10	-	-
Abstract Class Test	6	-	6	6	6	6	6	-	-
Error Simulator	-	0	-	-	-	-	-	-	-
Test Case With Time Specification	-	-	-	-	-	-	-	3	3
Total	16	0	16	16	16	16	16	3	3

D.3 Resultados do Terceiro Participante

Da mesma forma que os outros dois participantes, a Tabela D.7 mostra os dados coletados após a execução do estudo pelo primeiro participante, para o cálculo das métricas com relação ao esforço e à corretude. Nesta tabela são apresentados o tempo gasto, em minutos, na aplicação da abordagem para a geração dos casos de teste, o número total de casos de teste gerados, o número total de casos de teste válidos e os padrões de teste aplicados, para cada caso de uso do sistema.

No caso deste participante, o número de casos de teste válidos é igual ao número de casos de teste gerados. Isso acontece porque ele realizou a abordagem automática, utilizando a ferramenta de suporte para a geração dos casos de teste. Consequentemente, os casos de teste gerados são sintaticamente corretos, são semanticamente consistentes com a aplicação, podem ser executados e representam os cenários de integração propostos. Além disso, eles

Tabela D.7: Dados coletados na execução do estudo (participante 3).

Caso de Uso	Tempo Gasto (min)	CTs Gerados	CTs Válidos	Padrões
Validate PIN	10 min	13	13	RTST
Withdraw Funds	17 min	25	25	RTST, ACT
Query Account	10 min	11	11	RTST, ACT
Transfer Funds	12 min	18	18	RTST, ACT
Deposit Funds	13 min	35	35	RTST, ACT, TCWTS
Exceptions	7 min	3	3	ES

cobrem todos os caminhos dos diagramas de sequência dos modelos, de acordo com a aplicação do padrão *Round-trip Scenario Test*. Em suma, foram gerados cento e cinco casos de teste válidos em um tempo de aproximadamente sessenta e nove minutos. Vale ressaltar que para este participante, o tempo gasto referiu-se basicamente à aplicação do perfil IOP aos diagramas e à configuração da ferramenta. O tempo de geração dos casos de teste, neste caso, não foi medido pelo fato de ser insignificante (poucos segundos) e não atribuir valor à métrica.

Para a medição da capacidade de detecção de defeitos dos casos de teste gerados pelo participante, no intuito de avaliar a eficácia da abordagem realizada no estudo, os casos de teste foram agrupados por cenário de integração especificado e por padrão de teste, conforme mostrado na Tabela D.8. Nesta tabela, em cada coluna é mostrado o número de casos de teste gerados com relação ao número mínimo de casos de teste que deveriam ter sido construídos (para garantir a cobertura). É possível perceber, com esse dados, que todos os padrões foram aplicados. Além disso, todos os casos de teste necessários para todos os cenários foram construídos. No total, foram contabilizados sessenta e dois casos de teste de acordo com o padrão *Round-trip Scenario Test*, trinta e seis de acordo com o padrão *Abstract Class Test*, quatro de acordo com o padrão *Test Case With Time Specification* e três de acordo com o padrão *Error Simulator*. Após esse agrupamento, os casos de teste foram analisados com respeito aos tipos de defeitos que poderiam detectar.

A seguir, é apresentada a análise dos casos de teste de acordo com cada padrão de teste:

- ***Round-trip Scenario Test*:** Como o padrão foi corretamente aplicado e foram gerados casos de teste que cobrem todos os caminhos dos cenários, foi contabilizado pelo

Tabela D.8: Casos de teste válidos por cenário de acordo com os padrões de teste (participante 3).

Casos de Uso e Cenários		**RTST**[1]	**ACT**[2]	**TCWTS**[3]	**ES**[4]
Validate PIN	Cenário 1	2 de 2	-	-	-
	Cenário 2	3 de 3	-	-	-
	Cenário 3	4 de 4	-	-	-
	Cenário 4	4 de 4	-	-	-
Withdraw Funds	Cenário 5	4 de 4	3 de 3	-	-
	Cenário 6	1 de 1	-	-	-
	Cenário 7	4 de 4	3 de 3	-	-
	Cenário 8	4 de 4	6 de 6	-	-
Query Account	Cenário 9	2 de 2	1 de 1	-	-
	Cenário 10	1 de 1	-	-	-
	Cenário 11	2 de 2	1 de 1	-	-
	Cenário 12	2 de 2	2 de 2	-	-
Transfer Funds	Cenário 13	3 de 3	2 de 2	-	-
	Cenário 14	1 de 1	-	-	-
	Cenário 15	3 de 3	2 de 2	-	-
	Cenário 16	3 de 3	4 de 4	-	-
Deposit Funds	Cenário 17	6 de 6	3 de 3	-	-
	Cenário 18	1 de 1	-	-	-
	Cenário 19	6 de 6	3 de 3	-	-
	Cenário 20	6 de 6	6 de 6	-	-
	Cenário 21	-	-	4 de 4	-
Exceptions	Cenário 22	-	-	-	3 de 3

menos um defeito para cada um dos seis tipos identificados para todos os vinte cenários nos quais esse padrão podia ser aplicado, totalizando em cento e vinte defeitos;

- ***Abstract Class Test*:** Como este padrão foi corretamente aplicado para todos os doze casos que requeriam a sua aplicação, os tipos de defeitos de integração que poderiam ser capturados pelos casos de teste gerados pelo padrão foram contabilizados para cada

um desses casos, totalizando em sessenta defeitos;

- ***Test Case With Time Specification*:** Como este padrão foi corretamente aplicado, os quatro casos de teste gerados pelo participante poderiam capturar os defeitos relacionados ao conflito de componentes e o bloqueio de funções/tarefas (8 e 9), totalizando em oito defeitos;

- ***Error Simulator*:** Como este padrão foi corretamente aplicado, os três casos de teste gerados pelo participante poderiam capturar o defeito do tipo 2 (tratamento de erros/exceções incorreto), totalizando em três defeitos.

A Tabela D.9 apresenta os possíveis defeitos de integração que podem ser detectados pelos casos de teste gerados de acordo com a análise feita.

Tabela D.9: Tipos de defeitos de integração que poderiam ser detectados pelos casos de teste (participante 3).

Padrão de Teste	**Número de Defeitos por Tipo**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Round-trip Scenario Test	20	-	20	20	20	20	20	-	-
Abstract Class Test	12	-	12	12	12	12	12	-	-
Error Simulator	-	3	-	-	-	-	-	-	-
Test Case With Time Specification	-	-	-	-	-	-	-	4	4
Total	32	3	32	32	32	32	32	4	4

Apêndice E

Resultados do Segundo Estudo de Caso

Este Apêndice mostra, detalhadamente, os resultados obtidos com o estudo de caso realizado para avaliar a ferramenta de suporte desenvolvida. Nesse estudo, foram utilizadas as modelagens dos quatro sistemas descritos nos Apêndices B e C. Os modelos foram desenvolvidos usando a ferramenta de modelagem *MagicDraw* e exportados para o formato XMI, para que pudessem ser utilizados na ferramenta. Os cenários de integração considerados nesse estudo de caso para todos os sistemas foram aqueles onde todas as classes são necessárias para o teste, ou seja, já foram testadas individualmente e deseja-se integrá-las.

E.1 Precisão

Para medir a precisão, primeiramente, a ferramenta foi executada com os modelos de todos os sistemas, para verificar se os padrões que deveriam ser aplicados, foram realmente aplicados. Em seguida, foram realizadas algumas mutações nos modelos, para ver o comportamento da ferramenta para outras situações.

Object Manager System

Esse sistema corresponde ao sistema apresentado na Seção C.1. Para esse estudo, todas as classes, com exceção da classe *ObjectManager*, foram anotadas com os estereótipos *DependingClass* e *IntegratedClass* do perfil IOP. A classe *ObjectManager* foi escolhida como a *ClassToBeIntegrated*.

Para este cenário inicial, os padrões de teste que deveriam ser aplicados para a geração

dos casos de teste são o *Round-Trip Scenario Test* por causa do diagramas de sequência *Add Object* e *Remove Object*, e o *Abstract Class Test* por causa da classe abstrata *MyLyst*. É importante ressaltar que a aplicação do primeiro padrão só é necessária porque a classe *ObjectManager*, que se deseja integrar, faz parte dos diagramas em questão. A aplicação do segundo padrão, por sua vez, só é necessária porque a classe *MyLyst* também faz parte do diagrama *Remove Object*, caso contrário, não seria preciso.

O Código E.1 mostra uma instância do meta-modelo *ApplicableTestPatterns* (em XMI) gerada pela ferramenta com a execução do módulo de identificação, no qual é possível perceber que os dois padrões foram identificados corretamente para as situações necessárias. Logo tem-se que N_{pi} = *3* e N_{pa} = *3*, onde N_{pi} é o número de vezes que os padrões de teste foram identificados para a aplicação, e N_{pa} é o número de vezes que os padrões de teste deveriam ser aplicados de acordo com as características do sistema.

Código Fonte E.1: XMI para o cenário inicial do *Object Manager System*.

```
1  <xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns="ApplicableTestPatterns">
2      <TestPattern name="Round-trip Scenario Test">
3          <involvedElement name="Remove Object" type="UML2!Collaboration"/>
4      </TestPattern>
5      <TestPattern name="Round-trip Scenario Test">
6          <involvedElement name="Add Object" type="UML2!Collaboration"/>
7      </TestPattern>
8      <TestPattern name="Abstract Class Test">
9          <involvedElement name="MyList" type="UML2!Class"/>
10     </TestPattern>
11 </xmi:XMI>
```

Após a execução da ferramenta para o cenário acima, foram feitas as seguintes mutações nos modelos, a fim de verificar outras situações de aplicabilidade dos padrões:

1. Situação onde, dado o cenário inicial, a classe *MyList* passou a ser uma classe concreta. Dessa forma, o padrão *Abstract Class Test* não precisa mais ser aplicado;

2. Situação onde, dado o cenário inicial, a classe *MyObject* passou a ser a classe a ser integrada (*ClassToBeIntegrated*). Neste caso, o padrão *Round-Trip Scenario Test* não precisa ser aplicado.

Os Códigos E.2 e E.3 mostram instâncias do meta-modelo *ApplicableTestPatterns* (em XMI) geradas pela ferramenta com a execução do módulo de identificação para as duas

situações de mutação descritas acima, respectivamente. É possível perceber, no primeiro código, que o único padrão identificado foi o *Round-Trip Scenario Test*, para os dois diagramas de sequência. Logo, para este caso, tem-se que N_{pi} = N_{pa} = *2*. Já no segundo código, o único padrão identificado foi o *Abstract Class Test*, conforme previsto. Logo, tem-se que N_{pi} = N_{pa} = *1*.

Código Fonte E.2: XMI para o primeiro cenário de mutação do *Object Manager System*.

```
1 <xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns="ApplicableTestPatterns">
2     <TestPattern name="Round-trip Scenario Test">
3         <involvedElement name="Remove Object" type="UML2!Collaboration"/>
4     </TestPattern>
5     <TestPattern name="Round-trip Scenario Test">
6         <involvedElement name="Add Object" type="UML2!Collaboration"/>
7     </TestPattern>
8 </xmi:XMI>
```

Código Fonte E.3: XMI para o segundo cenário de mutação do *Object Manager System*.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns="ApplicableTestPatterns">
3     <TestPattern name="Abstract Class Test">
4         <involvedElement name="MyList" type="UML2!Class"/>
5     </TestPattern>
6 </xmi:XMI>
```

Simplified ATM System

Esse sistema corresponde ao sistema apresentado na Seção C.2. Para esse estudo, todas as classes, com exceção da classe *BankDatabse*, foram anotadas com os estereótipos *DependingClass* e *IntegratedClass* do perfil IOP. A classe *BankDatabse* foi escolhida como *ClassToBeIntegrated*.

Para este cenário inicial, os padrões de teste que deveriam ser aplicados para a geração dos casos de teste são o *Round-Trip Scenario Test* por causa do diagramas de sequência *Authenticate Client*, *Withdraw Funds* e *Deposit Funds* e o *Abstract Class Test* por causa da classe abstrata *Account*. É importante ressaltar que a aplicação do primeiro padrão só é necessária porque a classe *BankDatabse*, que se deseja integrar, faz parte dos diagramas em questão. A aplicação do segundo padrão, por sua vez, só é necessária porque a classe *Account* também faz parte dos diagramas, caso contrário, não seria preciso. Ainda, o padrão

Abstract Class Test não é aplicável para a classe abstrata *Transaction*, pois ela não participa de nenhum diagrama de sequência, e o padrão *Test Case With Time Specification* também não é aplicável porque a restrição de duração (*DurationConstraint*) no diagrama de sequência *Deposit Funds* não restringe a *ClassToBeIntegrated*.

O Código E.4 mostra uma instância do meta-modelo *ApplicableTestPatterns* (em XMI) gerada pela ferramenta com a execução do módulo de identificação, no qual é possível perceber que os três padrões foram identificados corretamente para as situações necessárias. Logo tem-se que N_{pi} = N_{pa} = *4*.

Código Fonte E.4: XMI para o cenário inicial do *Simplified ATM System*.

```
1  <xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns="ApplicableTestPatterns">
2       <TestPattern name="Round-trip Scenario Test">
3            <involvedElement name="Deposit Funds" type="UML2!Collaboration"/>
4       </TestPattern>
5       <TestPattern name="Round-trip Scenario Test">
6            <involvedElement name="Authenticate Client" type="UML2!Collaboration"/>
7       </TestPattern>
8      <TestPattern name="Round-trip Scenario Test">
9           <involvedElement name="Withdraw Funds" type="UML2!Collaboration"/>
10     </TestPattern>
11     <TestPattern name="Abstract Class Test">
12          <involvedElement name="Account" type="UML2!Class"/>
13     </TestPattern>
14 </xmi:XMI>
```

Após a execução da ferramenta para o cenário acima, foram feitas as seguintes mutações nos modelos, a fim de verificar outras situações de aplicabilidade dos padrões:

1. Situação onde, dado o cenário inicial, a classe *Account* passou a ser uma classe concreta e a ser integrada (*ClassToBeIntegrated*). Dessa forma, o padrão *Abstract Class Test* não precisa mais ser aplicado e o padrão *Round-Trip Scenario Test* só deve ser aplicado para os diagramas de sequência *Deposit Funds* e *Withdraw Funds*;

2. Situação onde, dado o cenário inicial, a classe *DepositSlot* passou a ser a *ClassToBeIntegrated*). Neste caso, o padrão *Test Case With Time Specification* deve ser aplicado, o padrão *Round-Trip Scenario Test* só deve ser aplicado para o diagrama *Deposit Funds* e padrão *Abstract Class Test* continua sendo aplicável.

Os Códigos E.5 e E.6 mostram instâncias do meta-modelo *ApplicableTestPatterns* (em

XMI) geradas pela ferramenta com a execução do módulo de identificação para as duas situações de mutação descritas acima, respectivamente. É possível perceber, no primeiro código, que o padrão identificado foi apenas o *Round-Trip Scenario Test* (para os diagramas *Deposit Funds* e *Withdraw Funds*). Logo, para este caso, tem-se que $N_{pi} = N_{pa} = 2$. Já no segundo código, os três padrões previstos foram corretamente aplicados. Logo, tem-se que $N_{pi} = N_{pa} = 3$.

Código Fonte E.5: XMI para o primeiro cenário de mutação do sistema *Simplified ATM System*.

```
1 <xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns="ApplicableTestPatterns">
2     <TestPattern name="Round-trip Scenario Test">
3         <involvedElement name="Deposit Funds" type="UML2!Collaboration"/>
4     </TestPattern>
5     <TestPattern name="Round-trip Scenario Test">
6         <involvedElement name="Withdraw Funds" type="UML2!Collaboration"/>
7     </TestPattern>
8 </xmi:XMI>
```

Código Fonte E.6: XMI para o segundo cenário de mutação do sistema *Simplified ATM System*.

```
1 <xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns="ApplicableTestPatterns">
2     <TestPattern  name="Round-trip Scenario Test">
3         <involvedElement name="Deposit Funds" type="UML2!Collaboration"/>
4     </TestPattern>
5     <TestPattern name="Abstract Class Test">
6         <involvedElement name="Account" type="UML2!Class"/>
7     </TestPattern>
8     <TestPattern name="Test Case With Time Specification">
9         <involvedElement name="1..60" type="UML2!DurationConstraint"/>
10   </TestPattern>
11 </xmi:XMI>
```

Sistema *File Manager*

Esse sistema corresponde ao sistema apresentado na Seção C.3. Para esse estudo, todas as classes, com exceção da classe *FileManager*, foram anotadas com os estereótipos *DependingClass* e *IntegratedClass* do perfil IOP. A classe *FileManager* foi escolhida como *ClassToBeIntegrated*.

Para este cenário inicial, os padrões de teste que deveriam ser aplicados para a geração dos casos de teste são o *Round-Trip Scenario Test* por causa do diagramas de sequência *Open File* e o padrão *Error Simulator* por causa do método *open()* da classe *File* que lança exceções. É importante ressaltar que a aplicação do primeiro padrão só é necessária porque a classe *FileManager*, que se deseja integrar, faz parte do diagrama em questão. Ainda, o padrão *Error Simulator* só é aplicável para o diagrama de sequência *Open File* porque ele possui as características necessárias (o tratamento das exceções).

O Código E.7 mostra uma instância do meta-modelo *ApplicableTestPatterns* (em XMI) gerada pela ferramenta com a execução do módulo de identificação, no qual é possível perceber que os dois padrões foram identificados corretamente para as situações necessárias. Logo, para este caso, tem-se que $N_{pi} = N_{pa} = 2$.

Código Fonte E.7: XMI para o cenário inicial do sistema *File Manager System*.

```
1 <xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns="ApplicableTestPatterns">
2     <TestPattern name="Round-trip Scenario Test">
3         <involvedElement name="Open File" type="UML2!Collaboration"/>
4     </TestPattern>
5     <TestPattern name="Error Simulator">
6        <involvedElement name="File" type="UML2!Class"/>
7     </TestPattern>
8 </xmi:XMI>
```

Após a execução da ferramenta para o cenário acima, foram feitas as seguintes mutações nos modelos, a fim de verificar outras situações de aplicabilidade dos padrões:

1. Situação onde, dado o cenário inicial, a classe *Exception* passou a ser integrada (*ClassToBeIntegrated*). Dessa forma, o padrão *Round-Trip Scenario Test* não deve ser mais aplicado;

2. Situação onde, dado o cenário anterior, método *open()* da classe *File* não lança mais as exceções (campo raisedException vazio). Dessa forma, nenhum padrão deve ser mais aplicado.

O Código E.8 mostra uma instância do meta-modelo *ApplicableTestPatterns* (em XMI) gerada pela ferramenta com a execução do módulo de identificação para a primeira situação de mutação descritas acima. É possível perceber, no código, que o padrão identificado foi

apenas o *Error Simulator*. Logo, para este caso, tem-se que N_{pi} = N_{pa} = *1*. Já para a segunda situação de mutação, o resultado da ferramenta foi um arquivo XMI vazio, ou seja, nenhum padrão foi aplicado, conforme previsto. Logo, tem-se que N_{pi} = N_{pa} = *0*.

Código Fonte E.8: XMI para o primeiro cenário de mutação do sistema *File Manager System*.

```
1 <xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns="ApplicableTestPatterns">
2     <TestPattern name="Error Simulator">
3         <involvedElement name="File" type="UML2!Class"/>
4     </TestPattern>
5 </xmi:XMI>
```

ATM System

Esse sistema corresponde ao sistema apresentado no Apêndice B. Para esse estudo, todas as classes, com exceção da classe *ATM*, foram anotadas com os estereótipos *DependingClass* e *IntegratedClass* do perfil IOP. A classe *ATM* foi escolhida como *ClassToBeIntegrated*.

Para este cenário inicial, os padrões de teste que deveriam ser aplicados para a geração dos casos de teste são o *Round-Trip Scenario Test* para todos os seis diagramas de sequência, o *Abstract Class Test* por causa das classes abstratas *Account* e *ATMTransaction* e o padrão *Error Simulator* por causa do método *execute()* da classe *ATMTransaction* que lança exceções. É importante ressaltar que a aplicação do primeiro padrão só é necessária porque a classe *ATM*, que se deseja integrar, faz parte do diagrama em questão. Ainda, o padrão *Error Simulator* só é aplicável para o diagrama de sequência *Execute Transaction Exception* porque ele possui as características necessárias (o tratamento das exceções).

O Código E.9 mostra uma instância do meta-modelo *ApplicableTestPatterns* (em XMI) gerado pela ferramenta com a execução do módulo de identificação no qual, é possível perceber que os três padrões foram identificados corretamente para as situações necessárias. Logo, para este caso, tem-se que N_{pi} = N_{pa} = *9*.

Código Fonte E.9: XMI para o cenário inicial do *ATM System*.

```
1 <xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns="ApplicableTestPatterns">
2     <TestPattern name="Round-trip Scenario Test">
3         <involvedElement name="Deposit Funds" type="UML2!Collaboration"/>
4     </TestPattern>
5     <TestPattern name="Round-trip Scenario Test">
6         <involvedElement name="Query Account" type="UML2!Collaboration"/>
7     </TestPattern>
```

```
8        <TestPattern name="Round-trip Scenario Test">
9            <involvedElement name="Transfer Funds " type="UML2!Collaboration"/>
10       </TestPattern>
11       <TestPattern name="Round-trip Scenario Test">
12           <involvedElement name="Validate PIN" type="UML2!Collaboration"/>
13        </TestPattern>
14        <TestPattern name="Round-trip Scenario Test">
15            <involvedElement name="Withdraw Funds" type="UML2!Collaboration"/>
16        </TestPattern>
17        <TestPattern name="Round-trip Scenario Test">
18           <involvedElement name="Execute Transaction Exception" type="UML2!Collaboration"/>
19        </TestPattern>
20        <TestPattern name="Abstract Class Test">
21           <involvedElement name="Account" type="UML2!Class"/>
22        </TestPattern>
23       <TestPattern name="Abstract Class Test">
24           <involvedElement name="ATMTransaction" type="UML2!Class"/>
25       </TestPattern>
26      <TestPattern name="Error Simulator">
27          <involvedElement name="ATMTransaction" type="UML2!Class"/>
28      </TestPattern>
29 </xmi:XMI>
```

Após a execução da ferramenta para o cenário acima, foram feitas as seguintes mutações nos modelos, a fim de verificar outras situações de aplicabilidade dos padrões:

1. Situação onde, dado o cenário inicial, as classes *Account* e *ATMTransaction* passaram a ser classes concretas e a classe *Account* passou a ser a *ClassToBeIntegrated*. Dessa forma, o padrão *Abstract Class Test* não precisa mais ser aplicado e o padrão *Round-Trip Scenario Test* não deve ser aplicado para os diagramas de sequência *Validate PIN* e *Execute Transaction Exception*;

2. Situação onde, dado o cenário anterior, a classe *DepositSlot* passou a ser a *ClassToBeIntegrated* e o método *execute()* da classe *ATMTransaction* não lança mais as exceções (campo raisedException vazio). Neste caso, o padrão *Test Case With Time Specification* deve ser aplicado, o padrão *Round-Trip Scenario Test* só deve ser aplicado para o diagrama *Deposit Funds* e os padrões *Abstract Class Test* e *Error Simulator* não devem ser aplicados.

Os Códigos E.10 e E.11 mostram instâncias do meta-modelo *ApplicableTestPatterns* (em XMI) geradas pela ferramenta com a execução do módulo de identificação para as duas

situações de mutação descritas acima, respectivamente. É possível perceber, no primeiro código, que os padrões identificados foram apenas o *Round-trip Scenario Test* (para quatro diagramas de sequência) e o *Error Simulator*. Logo, para este caso, tem-se que N_{pi} = N_{pa} = *5*. Já no segundo código, os padrões identificados foram apenas o *Round-trip Scenario Test* (para o diagrama *Deposit Funds*) e o padrão *Test Case With Time Specification*, conforme previsto. Logo, tem-se que N_{pi} = N_{pa} = *2*.

Código Fonte E.10: XMI para o primeiro cenário de mutação do *ATM System*.

```
1  <xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns="ApplicableTestPatterns">
2      <TestPattern name="Round-trip Scenario Test">
3          <involvedElement name="Deposit Funds" type="UML2!Collaboration"/>
4      </TestPattern>
5      <TestPattern  name="Round-trip Scenario Test">
6          <involvedElement name="Query Account" type="UML2!Collaboration"/>
7      </TestPattern>
8      <TestPattern name="Round-trip Scenario Test">
9          <involvedElement name="Transfer Funds " type="UML2!Collaboration"/>
10     </TestPattern>
11      <TestPattern name="Round-trip Scenario Test">
12          <involvedElement name="Withdraw Funds" type="UML2!Collaboration"/>
13      </TestPattern>
14    <TestPattern name="Error Simulator">
15        <involvedElement name="ATMTransaction" type="UML2!Class"/>
16    </TestPattern>
17 </xmi:XMI>
```

Código Fonte E.11: XMI para o segundo cenário de mutação do *ATM System*.

```
1  <xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns="ApplicableTestPatterns">
2      <TestPattern name="Round-trip Scenario Test">
3          <involvedElement name="Deposit Funds" type="UML2!Collaboration"/>
4      </TestPattern>
5      <TestPattern name="Test Case With Time Specification">
6          <involvedElement name="{1..60}" type="UML2!DurationConstraint"/>
7      </TestPattern>
8  </xmi:XMI>
```

Resultados Gerais para os Sistemas

A Tabela E.1 resume os dados obtidos de todos os sistemas para o cálculo da precisão. Analisando esses dados, é possível perceber que todos os padrões foram devidamente identificados nas situações em que a aplicação era requerida.

Tabela E.1: Resumo dos dados coletados de todos os sistemas para o cálculo da precisão.

Sistemas e Cenários		N_{pa}	N_{pi}
Sistema 1[1]	Cenário Inicial	3	3
	Cenário de Mutação 1	2	2
	Cenário de Mutação 2	1	1
Sistema 2[2]	Cenário Inicial	4	4
	Cenário de Mutação 1	3	3
	Cenário de Mutação 2	2	2
Sistema 3[3]	Cenário Inicial	2	2
	Cenário de Mutação 1	1	1
	Cenário de Mutação 2	0	0
Sistema 4[4]	Cenário Inicial	9	9
	Cenário de Mutação 1	5	5
	Cenário de Mutação 2	2	2

E.2 Escalabilidade

Para medir a escalabilidade da ferramenta, foram executados o módulo de filtragem dos modelos e o modulo de geração de casos de teste, considerados os mais críticos para essa medição. Para a obtenção dos dados, foi cronometrado o tempo de execução de cada módulo da ferramenta para cada sistema. A Tabela E.2 mostra os dados obtidos, especificamente o tempo gasto, em segundos, na execução de cada módulo e o tempo total gasto, para os quatro sistemas.

Tabela E.2: Dados coletados de todos os sistemas para o cálculo da escalabilidade.

	Sistema 1[1]	**Sistema 2**[2]	**Sistema 3**[3]	**Sistema 4**[4]
Filtragem dos modelos	2	4	2	17
Geração de casos de teste	10	30	5	148
Total (T_{exec})	12	34	7	165

[1] *Object Manager System*

[2] *Simplified ATM System*

[3] *File Manager System*

[4] *ATM System*

Apêndice F

Questionários de Avaliação

Questionário 1 - Abordagem Manual

1. Como você classificou o grau de dificuldade da aplicação da abordagem?

( ) Muito Fácil. ( ) Fácil. ( ) Mediano. ( ) Difícil. ( ) Muito Difícil.

2. Na sua opinião, quais aspectos da abordagem tornam sua aplicação fácil/difícil de usar?

3. Você gerou casos de teste utilizando apenas os padrões de teste documentados?

( ) Sim. Utilizei apenas os padrões de teste documentados.

( ) Não. Utilizei também conhecimento próprio.

4. Caso você tenha gerado casos de teste com auxílio de outro tipo de conhecimento, por exemplo, conhecimento do domínio ou experiência em técnicas/estratégias para a geração de casos de teste a partir de modelos, identifique para que casos de teste isso ocorreu e que tipo de conhecimento você utilizou.

5. Como o uso dessa nova abordagem auxiliou na geração dos casos de teste?

( ) Negativamente. A abordagem atuou como um obstáculo. O meu desempenho teria sido melhor se eu não tivesse utilizado-a.

( ) Neutro. Acho que construiria os mesmos casos de teste, ou até casos de teste mais eficientes/eficazes, se não tivesse utilizado essa abordagem.

( ) Positivamente. A abordagem me ajudou a construir casos de teste que são importantes. Talvez não tivesse pensado em construir alguns dos casos de testes propostos pelos padrões de teste.

6. Você utilizaria a abordagem para outros projetos?

( ) Sim. ( ) Talvez. Justifique. ( ) Não. Justifique.

7. A abordagem é efetiva no que se propõe?Justifique.

( ) Sim.   ( ) Neutro.   ( ) Não. Justifique.

8. A abordagem é relevante para o contexto ao qual está inserida?

( )Sim.   ( ) Neutro.   ( ) Não. Justifique.

9. Na sua opinião, como a abordagem poderia ser melhorada?

10. Por favor, registre quaisquer comentários que julgar pertinente.

Questionário 2 - Abordagem Automática

1. Como você classificou o grau de dificuldade da aplicação da abordagem?

( ) Muito Fácil.   ( ) Fácil.   ( ) Mediano.   ( ) Difícil.   ( ) Muito Difícil.

2. Na sua opinião, quais aspectos da abordagem tornam sua aplicação fácil/difícil de usar?

3. Como você classificou o grau de dificuldade da utilização da ferramenta?

( ) Muito Fácil.   ( )Fácil.   ( )Mediano.   ( ) Difícil.   ( ) Muito Difícil.

4. Como o uso da ferramenta auxiliou na geração dos casos de teste?

( ) Negativamente. A ferramenta atuou como um obstáculo. O meu desempenho teria sido melhor se eu não tivesse utilizado-a.

( ) Neutro. Acho que construiria os mesmos casos de teste, ou até casos de teste mais eficientes/eficazes, se não tivesse utilizado a ferramenta, ou outro processo (manual ou automático).

( ) Positivamente. A ferramenta gerou casos de teste que são importantes. Talvez não tivesse gerado alguns dos casos de testes propostos pelos padrões de teste se tivesse utilizado qualquer outro processo (manual ou automático).

5. Você utilizaria a ferramenta/abordagem para outros projetos?

( ) Sim.   ( ) Talvez. Justifique.   ( ) Não. Justifique.

6. A abordagem é efetiva no que se propõe? Justifique.

( ) Sim.   ( ) Neutro.   ( ) Não. Justifique.

7. A abordagem é relevante para o contexto ao qual está inserida?

( ) Sim.   ( ) Neutro.   ( ) Não. Justifique.

8. Na sua opinião, como a abordagem poderia ser melhorada? E a ferramenta?

9. Por favor, registre quaisquer comentários que julgar pertinente.