Verificação de Artefatos de Diagramas de Classe UML através da Aplicação Testes de Design

Waldemar Pires Ferreira Neto

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

Franklin de Souza Ramalho

(Orientador)

Dalton Dário Serey Guerrero

(Orientador)

Campina Grande, Paraíba, Brasil

©Waldemar Pires Ferreira Neto, 17/05/2009

.
.
.
.
.
.
.
.
.
.
.
.
.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

F383v
2009 Ferreira Neto, Waldemar Pires.

Verificação de artefatos de diagramas de classe UML através da aplicação testes de design / Waldemar Pires Ferreira Neto. — Campina Grande, 2009.

127 f. : il.

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

Referências.

Orientadores: Prof. Dr. Franklin de Souza Ramalho, Prof. Dr. Dalton Dário Serey Guerrero.

1. Modelagem e Simulação de Sistemas. 2. Garantia de Qualidade de Programas. 3. Testes de Validação. 4. Administração de Desenvolvimento de Programas. I. Título.

CDU – 004.414.23(043)

Resumo

Neste trabalho investigamos e propomos uma técnica completamente automática para executar verificação de conformidade entre uma implementação de um sistema em Java e seu design expresso através de diagrama de classe UML. Essa verificação é realizada através de testes de design, um tipo de teste automático que verifica a conformidade de uma implementação em relação às regras de design expressadas através de código de teste. Definimos *templates* de testes de design para os artefatos mais usuais do diagrama de classe UML. E desenvolvemos a ferramenta UDT (*UML Design Tester*) baseada na abordagem MDA, capaz de aplicar esses *templates* sobre os elementos do diagrama de classe para gerar automaticamente os testes de design específicos para cada um dos elementos. Por fim, validamos nossa técnica através de um estudo de caso, o qual verifica a conformidade entre um sistema real (*Findbugs*) e o seu diagrama de classe gerado através de engenharia reversa. Adotamos essa abordagem para validação, pois não conseguimos identificar nenhum sistema com mais de 1000 classes que disponibilizasse o código do sistema e o seu design especificado em Diagramas de classes.

Abstract

In this work we propose and investigate a completely automatic technique to execute conformance verification between an implementation in Java and its design expressed by UML class diagram. This verification is performed through design tests, a kind of automatic test capable of verifing conformance of an implementation against design rules expressed by code. We define a design test templates to each most usual UML class diagram artifacts. And, we developed the UDT tool (UML Design Tester) 100% MDA-based, this tool is able to apply the templates over the class diagram elements in order to automatically generate the specific design tests for each of these elements. Concluding, we evaluated our technique using as case study that verifies the conformance between a real (Findbugs) system and its generated by reverse engineering.

Agradecimentos

Antes de mais nada a Deus, sem ele nada aqui existiria.

À minha família, por todo o apoido dado não somente durante o meu mestrado, mas por toda minha vida: Pepé, Del, Livinha, Narina, Tia Mima, Bruno JP, Vulcão, MárioIePeder, Vovó, Tio Sabino, Tia Gerusa, dentro muitos outros mais, tão importantes quanto os citados, mas não posso citá-los todos por limitação de espaço.

Aos meus orientadores, O Prof (Franklin) e Dalton, que me orientaram esplendidamente, ensinando-me não somente lições acadêmicas, mas também licões pessoais que eu as levarei por toda a minha vida.

Aos meu amigos que sempre me apoiaram: Opulus, Cleeeeerton, Hugo Parede, Ique, Lua, Tici, Tio Cavera, Jack, Amadinha, Madame Cartaxo, Dan, Verlinha, Makas, Lalinha, Danillo Cabeça, Tabira, Lucas Skywalker, Ana Stress, A Profa, Alisson Skywalker, Xico, dentre muitos outros mais.

Aos meus companheiros de Laboratório, esses grandes amigos que tiveram que me aguentar por todo esse o tempo do mestrado: Roberto Bit, Paulo Cabeça, Jemão, Pablito, Bel, Mirna 100m Rasos, Neto Wii, dentre tantos outros. E em especial a Singleton (Anderson) que me ajudou não somente na implementação mas como amigo de verdade.

Especialmente a duas pessoas, Aninha da Copin e Lilian do GMF, que meu ajudaram sempre e já me tiraram de maus bocados.

E a CAPES e FAPESQ, por todo o suporte financeiro que sem ele esse trabalho seria impossibilitado.

Conteúdo

1 Introdução **1**

1.1 *Overview* da Solução 4

1.2 Relevância 6

1.3 Estrutura da Dissertação 7

2 Fundamentação Teórica **9**

2.1 MDA 9

2.1.1 Meta-modelos 11

2.1.2 Transformações entre Modelos 13

2.1.3 Arquitetura Quatro Camadas da OMG 14

2.2 Diagrama de Classes da UML2 15

2.2.1 Classe 16

2.2.2 Operação 18

2.2.3 Atributo 19

2.2.4 Interface 20

2.2.5 Pacote 21

2.2.6 Associação 22

2.3 Teste de Design 23

2.4 DesignWizard 25

3 Verificação de Artefatos dos Diagrama de Classes UML 29

3.1 Template do Teste de Design 30

3.1.1 Classe 32

3.1.2 Operação 36

3.1.3 Atributo 40

3.1.4 Associação 42

3.1.5 Interface 46

3.1.6 Pacote 48

3.2 Exemplo de Aplicação dos Templates 51

3.2.1 Execução dos Testes Gerados 56

3.3 Considerações Finais 57

4 Geração Automática de Testes de Design 59

4.1 UML Design Tester (UDT) 60

4.1.1 Módulo de Pré-processamento 62

4.1.2 Módulo Gerador de Modelos 63

4.1.3 Módulo de Geração da Sintaxe Concreta 71

4.1.4 Módulo de Pós-Processamento 78

4.2 Considerações Finais 78

5 Avaliação 80

5.1 Metodologia dos Experimentos 81

5.1.1 Cenários 82

5.1.2 Métodos 85

5.2 Resultados e Avaliação 88

5.2.1 Precisão 88

5.2.2 Desempenho 94

5.3 Considerações Finais 96

6 Trabalhos Relacionados **98**

6.1 Verificação de Design 98

6.2 Considerações 101

7 Conclusões **102**

7.1 Contribuições 103

7.2 Trabalhos Futuros 105

A Templates de Teste de Design **113**

Lista de Figuras

1.1 Processo de desenvolvimento tradicional 2
1.2 Extensão do processo de desenvolvimento 3
1.3 *Overview* da ferramenta UDT 5

2.1 Simplificazccão Meta-modelo de UML para Classe 12
2.2 Exemplo de um modelo de uma classe de acordo com o Meta-modelo simplificado 12
2.3 Framework de MDA. Extraído e traduzido de Kleppe&Warner [31] . . . 16
2.4 Características de classe tratadas neste trabalho 17
2.5 Características de operações tratadas neste trabalho 18
2.6 Características de atributos tratadas neste trabalho 19
2.7 Características de interface tratadas neste trabalho 20
2.8 Características de pacotes tratadas neste trabalho 21
2.9 Características de associações tratadas neste trabalho 23
2.10 Arquitetura do DesignWizard 26

3.1 Processo geral para o uso de templates para a verificação de diagramas de classe 30
3.2 Classe de acesso ao banco de dados 51
3.3 Resultado dos testes de Design para a classe *DataHandler* 57

4.1 Principais atividades realizadas pela ferramenta UDT 61

4.2 Arquitetura para extensão da ferramenta UDT 64
4.3 Abordagem MDA do Módulo de Geração de Modelos de Testes de Design 65
4.4 Hierarquia das regras de transformações ATL 68
4.5 Abordagem MDA do Módulo de Geração da Sintaxe Concreta 72

5.1 Fórmula para Precisão 81

Lista de Tabelas

5.1 Ferramentas para Engenharia Reversa 83

5.2 Dados sobre os Testes de Design para Classe 88

5.3 Dados sobre os Testes de Design para Atributo 90

5.4 Dados sobre os Testes de Design para Método. 91

5.5 Dados sobre os Testes de Design para Pacote. 92

5.6 Dados sobre os Testes de Design para Interface. 93

5.7 Tempo de geração e execução dos testes de design para os projetos selecionados. 94

5.8 Tempo de geração e execução para cada tipo de artefato no projeto Findbugs. 95

6.1 Comparação entre trabalhos relacionados. 101

Lista de Códigos Fonte

2.1 Exemplo de Teste de Design 25

3.1 Trecho do Template do Teste de Design para Classe referente a assinatura do método de teste. 32

3.2 Trecho do Template do Teste de Design para Classe referente a verificação da existência da classe. 33

3.3 Trecho do Template do Teste de Design para Classe referente a verificação se a classe é abstrata. 33

3.4 Trecho do Template do Teste de Design para Classe referente a verificação da hierarquia de classes. 34

3.5 Trecho do Template do Teste de Design para Classe referente a verificação da visibilidade da classe. 34

3.6 Trecho do Template do Teste de Design para Classe referente a verificação da realização de interfaces. 35

3.7 Trecho do Template do Teste de Design para Operações referente assinatura do método de teste. 37

3.8 Trecho do Template do Teste de Design para Operações referente a verificação da existência da operação. 37

3.9 Trecho do Template do Teste de Design para Operações referente a verificação do tipo do retorno. 38

3.10 Trecho do Template do Teste de Design para Operações referente a verificação se a operação é abstrata. 38
3.11 Trecho do Template do Teste de Design para Operações referente a verificação se a operação é estática. 39
3.12 Trecho do Template do Teste de Design para Operações referente a verificação da visibilidade da operação. 39
3.13 Trecho do Template do Teste de Design para Atributos referente assinatura do método de teste. 40
3.14 Trecho do Template do Teste de Design para Atributos referente a verificação da existência do atributo. 40
3.15 Trecho do Template do Teste de Design para Atributos referente a verificação se o atributo é estático. 41
3.16 Trecho do Template do Teste de Design para Atributos referente a verificação da visibilidade do atributo. 41
3.17 Trecho do Template do Teste de Design para Atributos referente a verificação do tipo do atributo. 42
3.18 Trecho do Template do Teste de Design para cada Membro da Associação referente assinatura do método de teste. 43
3.19 Trecho do Template do Teste de Design para cada Membro da Associação referente a verificação do papel de cada membro. 43
3.20 Trecho do Template do Teste de Design para cada Membro da Associação referente a verificação do tipo de cada membro. 44
3.21 Trecho do Template do Teste de Design para cada Membro da Associação referente a verificação da visibilidade de cada membro. 45
3.22 Trecho do Template do Teste de Design para cada Membro da Associação referente a verificação dos membros de somente leitura. 46

3.23 Trecho do Template do Teste de Design para Interface referente assinatura do método de teste. 47
3.24 Trecho do Template do Teste de Design para Interface referente verificação da existência da Interface. 47
3.25 Trecho do Template do Teste de Design para Interface referente verificação da hierarquia da Interface. 48
3.26 Trecho do Template do Teste de Design para Pacote referente assinatura do método de teste. 49
3.27 Trecho do Template do Teste de Design para Pacote referente verificação da existência do pacote. 49
3.28 Trecho do Template do Teste de Design para Pacote referente verificação dos relacionamentos entre os pacotes. 50
3.29 Verificação da classe DataHandler. 52
3.30 Verificação do atributo self da classe DataHandler. 53
3.31 Verificação do método Construtor da classe DataHandler. 54
3.32 Verificação do método getInstance da classe DataHandler. 55
3.33 Implementação do *DataHandler* com o design alterado. 56
4.1 Transformação ATL para geração dos modelos de Teste de Design para classe. 69
4.2 Transformação para recuperação das informações sobre a declaração de Variável. 75
4.3 Transformação para a formatação da declaração de Variável. 77
A.1 Template do Teste de Design para Classe. 114
A.2 Template do Teste de Design para Atributo. 115
A.3 Template do Teste de Design para Operações. 116
A.4 Template do Teste de Design para Interface. 117
A.5 Template do Teste de Design para Pacote. 118

A.6 Template do Teste de Design para Associa ção. 119

Capítulo 1

Introdução

Uma atividade comum e fundamental a todo processo de software é o design do software. O design incorpora a descrição da estrutura do software, os dados que são manipulados pelo sistema, a descrição das interfaces entre os componentes do sistema, e algumas vezes o *algoritmo* usado [47]. O design realiza importantes decisões da arquitetura do software e possui um papel crucial no desenvolvimento, implantação e evolução do sistema [45]. Sendo assim, o design atua como o elo entre os requisitos e a implementação do software.

Contudo, nem sempre a implementação reflete o design proposto. De acordo com testemunhos de alguns gerentes de fábricas de software do Brasil, isso raramente acontece. Documentação errônea ou obsoleta é considerada uma das principais causas da baixa qualidade de software [14; 27]. Verificação de conformidade com o design é especialmente relevante em processos de desenvolvimento que promovem uma clara distinção entre artefatos de design e de implementação, como por exemplo, o RUP [55]. E, especialmente, se equipes diferentes desenvolverem os artefatos de design e os de implementação.

Porém, a verificação de conformidade entre design e implementação é ainda um ponto em aberto na engenharia de software com poucas ferramentas de automação dis-

poníveis. As principais ferramentas atualmente verificam o comportamento do sistema através de testes funcionais em detrimento à verificação da estrutura do sistema. Dentre as poucas que verificam estrutura são em sua maioria manuais.

Para ilustrar a importância da conformidade entre o design do sistema e o seu código, considere o processo de desenvolvimento ilustrado na Figura 1.1. Inicialmente, nesse processo, o arquiteto do software desenvolve o *projeto do sistema*, de forma que este contemple da melhor forma possível os *requisitos* do cliente. O *projeto do sistema* é um conjunto de artefatos que descrevem vários aspectos da forma como o sistema deve ser implementado. Dentre esses artefatos, um bastante usado para descrever a estrutura do sistema é o *design estrutural*, ele é responsável por exprimir a forma que a estrutura do código deve seguir. A etapa seguinte no processo de desenvolvimento é a etapa de *implementação*, é nessa etapa onde os programadores, baseados no *projeto do sistema*, implementam o código fonte desse sistema. Depois disso, vem a etapa de *teste*, onde diversos testes são executados, bugs podem ser descobertos e consertados. Por fim, a etapa de *implantação*, onde temos uma versão final do sistema pronta para ser usada. Um grande problema desse processo de desenvolvimento é que depois das diversas etapas intermediárias do processo, não há uma forma de garantir que o *projeto do sistema* ainda se mantém refletido no código final.

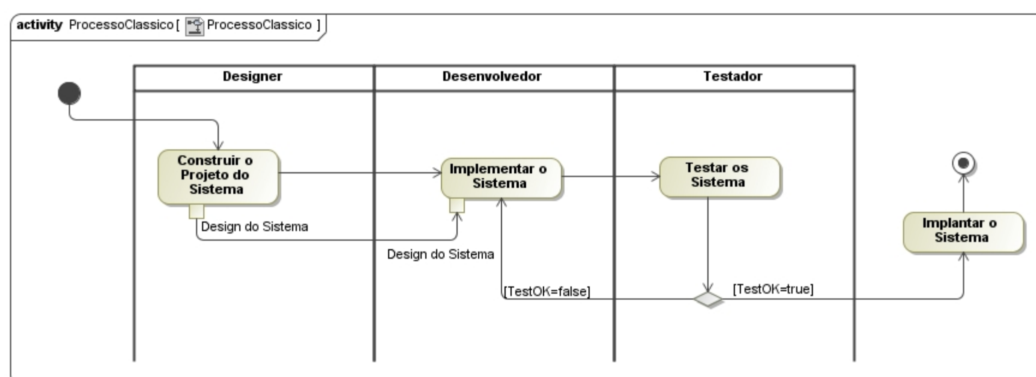

Figura 1.1: Processo de desenvolvimento tradicional

A Figura 1.2 exibe o mesmo processo de desenvolvimento tratado anteriormente, sendo que agora o processo foi estendido para a aplicação da nossa abordagem. Essa extensão teria um impacto mínimo no processo como um todo, tendo em vista que os arcos indicados com nome *auto* são ações completamente automáticas (ver Figura). Na prática esse processo de verificação é completamente manual, e com a nossa abordagem ele pode ser realizado completamente automático ou com o mínimo de revisão manual (considerando a verificação dos artefatos do diagrama de classe que não foram abordados nesse trabalho).

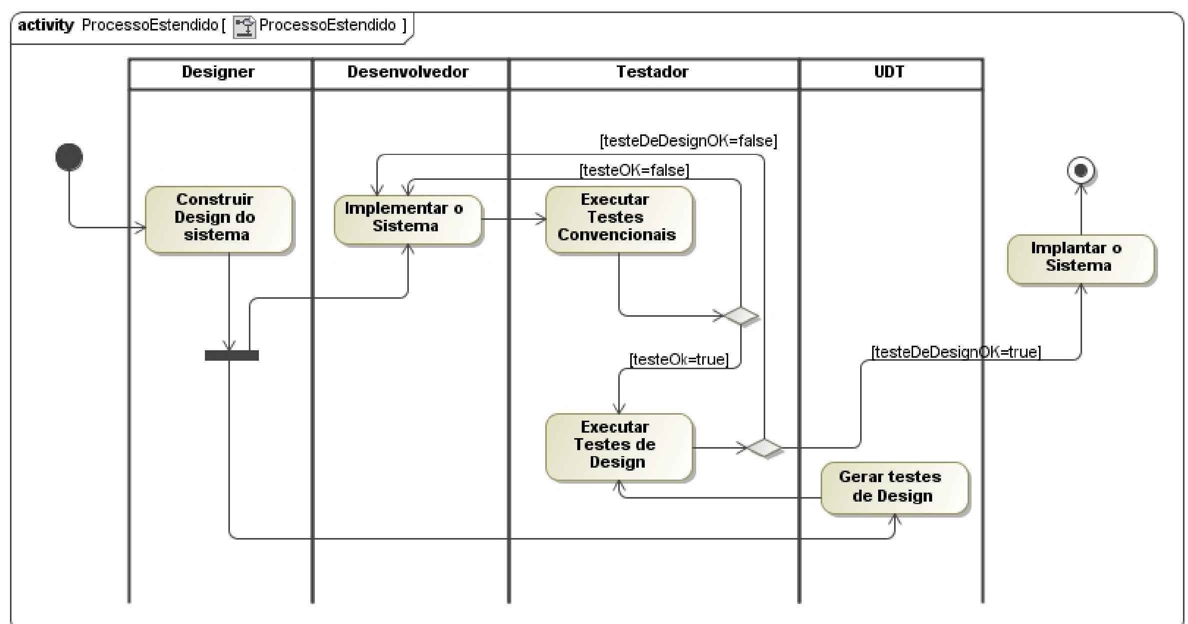

Figura 1.2: Extensão do processo de desenvolvimento

Atualmente, a notação UML 2.0 [40] é um formalismo padrão amplamente utilizado pela academia e pela indústria na especificação de design de software. Essa notação é usada para descrever tanto características estruturais quanto comportamentais do software.

Há ainda alguns trabalhos que propõem [15; 53] a verificação do design através

do conceito de teste de design. Teste de design é um tipo de teste automático que verifica a conformidade de uma implementação em relação a regras de design explícitas. Contudo, regras de design expressas através de código de teste não é uma forma usual para especificar design. Para tanto existe UML que é uma linguagem específica para esse fim.

Neste contexto, nós propomos neste trabalho uma abordagem baseada em templates[1] de testes de design para a verificação de artefatos de diagramas de classes UML contra código Java. Usando nossa abordagem, um designer que desenvolva o design estrutural do sistema baseado em modelos UML é capaz de derivar automaticamente os testes de design para verificar se a implementação está de acordo com o design proposto. Para a geração automática dos testes de design nós adotamos MDA, pois consideramos que essa abordagem consegue tratar todo o problema de aplicação de templates automaticamente com o uso de padrões internacionais, além do fato que um dos elementos do problema já envolve um dos padrões de MDA. Esta abordagem pode ser usada no desenvolvimento do software como ferramenta para manter a sincronia entre os documentos de design e a implementação.

1.1 *Overview* da Solução

Nossa solução para a verificação de artefatos do diagrama de classe UML fundamenta-se na criação de um catálogo de templates de testes de design. Cada template é responsável por criar os testes capazes de verificar um tipo de artefato abordado nessa dissertação.

Tendo em vista que a aplicação dos templates de teste de design manualmente seria uma tarefa muito dispendiosa, desenvolvemos a ferramenta UDT (*UML Design Tes-*

[1] Template (ou "modelo de documento´´) é um documento sem conteúdo, com apenas a apresentação visual (apenas cabeçalhos, por exemplo) e instruções sobre conteúdo de um documento a ser formado.

ter). Essa ferramenta é capaz de aplicar automaticamente os templates de testes sobre os artefatos do diagrama de classe, e assim gerar, também automaticamente, os testes específicos para o design expresso pelo diagrama de classe. A Figura 1.3 mostra uma *overview* do funcionamento do UDT. A implementação dessa ferramenta foi baseada nos conceitos de MDA [4] para a geração automática de testes de design.

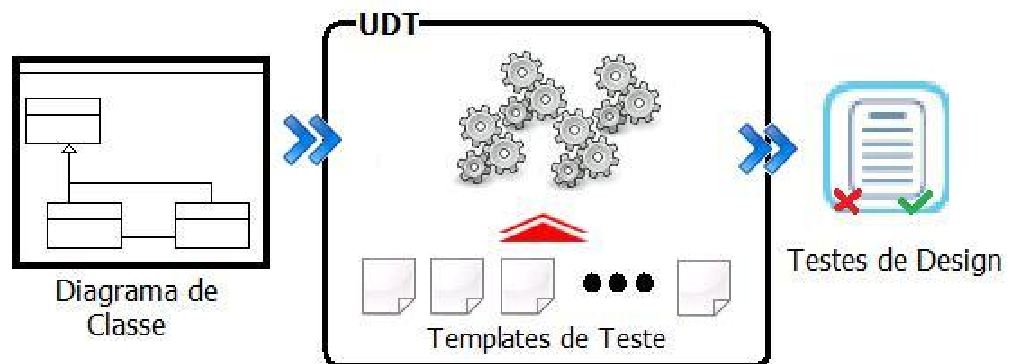

Figura 1.3: *Overview* da ferramenta UDT

A ferramenta UDT, além de servir como forma de aplicação automática dos templates de teste, serviu também como plataforma para avaliação de toda a abordagem criada. Essa avaliação foi realizada através de um estudo de caso verificando a conformidade entre o código de um sistema real e o seu diagrama de classe gerado através de engenharia reversa, maiores detalhes serão tratados no Capítulo 5. Utilizamos engenharia reversa para a geração do diagrama de classe UML, pois não encontramos disponível nenhum projeto de um sistema com o design especificado através de diagramas de classe UML e sua a implementação em Java.

1.2 Relevância

Uma das principais áreas onde esse trabalho pode afetar é na diminuição dos custos do software. Custos com a manutenção do software podem chegar até 90% do custo total do ciclo de vida do software [18]. Dentre esses custos destacamos o custo com a revisão de código. Um processo de revisão fortemente usado pelas empresas atualmente é o Design Peer Review [1]. Uma desvantagem desse processo é o fato de que normalmente quem realiza a revisão do código é um programador sênior, ou seja, um custo por hora mais alto é requerido. Com o nosso trabalho pode-se realizar parte da revisão de código de forma automática, e assim, reduzir custos.

Dentre os custos com a manutenção de software [14] destacamos os custos para a organização com a perda de mercado decorrente do envelhecimento prematura do software. Parnas [14] cita que uma das causas para o custo do software envelhecido é a diminuição da confiabilidade no código. Programas maiores e mais complexos com um projeto mal compreendido pela equipe levam à degradação do design do código, ocasionando numa complexidade desnecessária do código. Em outras palavras, quanto mais se altera um software sem ter conhecimento do projeto previsto para ele, maior será o esforço necessário para introduzir novas funcionalidades. Neste sentido nossa abordagem é importante por dois motivos. Em primeiro lugar, a criação de testes de design para as diversas entidades dos diagramas pode ajudar os desenvolvedores a compreender de uma forma mais clara como se deve implementar um sistema a partir dos diagramas construídos e como a implementação será verificada. Em segundo lugar, nossa proposta é um mecanismo capaz de manter uma constante verificação da conformidade com o design proposto, e assim promover uma maior confiabilidade ao código que está sendo desenvolvido.

Outro ponto onde nosso trabalho pode oferecer uma importante contribuição é no que se refere ao problema de manutenção do sincronismo entre a documentação e o código. Esse é um dos grandes problemas no desenvolvimento de software citado por

vários autores [14; 45; 31; 48]. Manter a documentação em conformidade com a implementação no desenvolvimento do software é uma tarefa muito difícil dado que os profissionais envolvidos trabalham sobre uma pressão muito grande quanto aos prazos, falta de entendimento do design, etc [21]. Sendo assim, o trabalho de manter todos os modelos, testes funcionais, teste de design e código em sincronia é bastante dispendioso, fazendo com que muitas vezes a documentação seja negligenciada, tornando-a obsoleta com relação ao código. Através da nossa abordagem, a maioria (senão todos) dos testes de design pode ser gerada automaticamente, provendo uma economia no esforço para a construção desses artefatos.

1.3 Estrutura da Dissertação

Este documento está estruturado da seguinte forma:

- Capítulo 2: Fundamentação Teórica. Nesse capítulo, detalharemos alguns conceitos que julgamos como essenciais para a compreensão desse trabalho: UML, MOF, MDA, testes de design e Design Wizard. A intenção é fornecer um embasamento teórico para o leitor, que lhe permita e facilite a leitura do restante da dissertação.

- Capítulo 3: Verificação de Artefatos do Diagrama de Classes UML. Nesse capítulo, mostraremos a forma como foram especificados os templates de teste para a criação dos testes de design para cara artefato do diagrama de classe UML abordado nessa dissertação.

- Capítulo 4: Geração Automática de Testes de Design: Neste capítulo, abordaremos a maneira de como a ferramenta UDT foi implementada para a aplicação automática dos templates de teste de design.

- Capítulo 5: Avaliação. Nesse capítulo, apresentaremos a maneira como foram dirigidos os experimentos sobre o estudo de caso para a avaliação da abordagem proposta neste trabalho. Adicionalmente, mostraremos os resultados obtidos e faremos uma análise sobre esses dados, obtidos no experimento.

- Capítulo 6: Trabalhos Relacionados. Nesse capítulo, mostraremos alguns trabalhos relacionados ao nosso que propõem técnicas de verificação de design. Além disso, compararemos esses trabalhos com nosso trabalho mostrado nessa dissertação.

- Capítulo 7: Conclusões. Neste capítulo, teceremos algumas conclusões sobre toda a abordagem proposta neste trabalho e sobre os resultados alcançados com ela. Por fim, faremos uma descrição das perspectivas de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

O objetivo deste capítulo é fornecer o embasamento teórico aos leitores acerca dos conceitos mais relevantes abordados ao longo de todo o trabalho. Os conceitos que julgamos como essenciais para a compreensão desse trabalho são: **MDA** (e **MOF**), a abordagem usada para construir a solução tratada nesse trabalho; **Diagramas de Classes UML**, um dos diagramas estruturais de UML (Unified Modeling Language) responsável por apresentar uma visão estática do projeto, mostrando uma coleção de elementos declarativos do modelo, tais como classes, tipos e seus relacionamentos; **Testes de Design**, um tipo de teste que verifica se um código implementado segue um design previamente estabelecido; e, por fim, **Design Wizard**, uma biblioteca capaz de fornecer informações estruturais sobre um programa Java.

2.1 MDA

Essa abordagem de desenvolvimento de software foi criada pela OMG (Object Management Group [39]) visando melhorar o processo de desenvolvimento de software, alterando o foco das ações onde se deve aplicar mais esforço durante o desenvolvimento de software.

MDA (Model Driven Architecure) representa uma visão de MDD (Model-Driven Development), sendo MDA a mais adotada até o presente momento. A idéia chave de MDA é mudar a ênfase de esforço e tempo empregados durante o ciclo de vida do software, focando-se na identificação de requisitos e organização do software através de modelos, ao invés de focar na implementação como é feito atualmente pelas metodologias tradicionais. Para alcançar esse objetivo, MDA sugere um conjunto de padrões de modelos:

- O CIM (*Computational Independent Model*) descreve os modelos de negócio. Eles devem ser abstratos o suficiente para permitir especificar os processos do negócio, os *stakeholders*, os departamentos, as dependências entre os processos, etc. Esses modelos não devem necessariamente tratar diretamente sobre sistema de software usado no negócio, mas devem especificar como e quais áreas do negócio são contempladas por esse sistema. Os modelos de negócio e os modelos do sistema de software em si podem ser bem diferentes, pois o principal objetivo dos modelos do negócio é fornecer os requisitos que o software deve alcançar, além de mostrar o domínio da aplicação.

- O PIM (*Platform Independent Model*) descreve como o sistema de software deve realizar a lógica do negócio da melhor forma possível. Ele é independente de plataforma pelo fato de que toda a descrição deve ser feita de uma forma tal que possa ser implementado por diversas tecnologias.

- Os PSM (*Platform Specific Model*) especificam como os PIMs devem ser implementados para as tecnologias envolvidas na solução. Um PIM pode ser traduzido em múltiplos PSM para diversas tecnologias. Por exemplo, um PIM pode descrever a arquitetura de sistema web cliente-servidor de múltiplas camadas. Já os PSM desse sistema podem descrever o software usando EJB, especificando as classes de controle de sessão, comunicação entre as camadas, etc. E outro PSM

descrevendo o banco de dados relacional especificando as tabelas em si com linhas, colunas, chaves estrangeiras, etc. Sendo assim, um PIM pode ser traduzido em um ou mais PSMs. Para cada plataforma especifica um PSM distinto será gerado. É fato que nos sistemas atuais uma vasta quantidade de tecnologias está sendo usada, dessa forma é comum encontrarmos vários PSMs para um único PIM.

- Código é o nível mais baixo no desenvolvimento de uma aplicação na abordagem MDA. Os códigos são gerados a partir de cada PSM das tecnologias escolhidas. Nesse nível acontece a realização dos modelos PSM. Por exemplo, nesse nível é que estão os códigos na linguagem de código escolhida ou os esquemas das tabelas do banco de dados.

2.1.1 Meta-modelos

O conceito de Meta-modelo é próximo ao das gramáticas BNF (Backus-Naur Form) [54] que descrevem um formalismo para definir se uma expressão pode ser considerada bem formada para uma determinada linguagem. Sendo que o contexto de Meta-modelagem é um pouco mais genérico, pois nem todas as entidades são baseadas estritamente em texto (como UML, que tem uma sintaxe gráfica). Além disso, Meta-modelos descrevem a sintaxe abstrata de um modelo, enquanto que gramáticas BNF descrevem a sintaxe concreta de uma linguagem.

Por exemplo, considere a declaração de classes nos diagramas UML. A Figura 2.1 mostra uma simplificação do Meta-modelo de UML referente à classe. Observando essa simplificação podemos notar que esse meta-modelo especifica que toda Classe (*Class*) em UML deve possuir: **Nome** (*Name*) do tipo *String* e possuir 0(zero) ou mais atributos (*Attribute*). A Figura 2.2 mostra um modelo concreto de uma classe em UML. Podemos perceber que esse modelo especifica: **Nome** da classe (*DataHandler*), e o atributo que

essa classe possui (*self*).

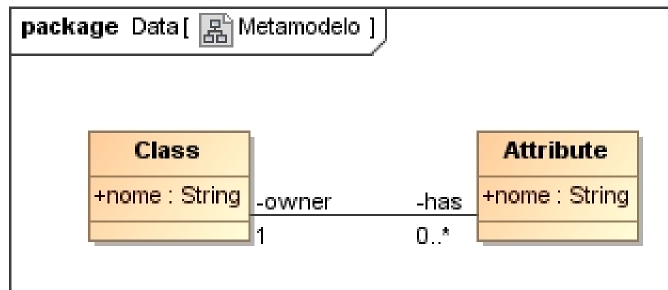

Figura 2.1: Simplificazccão Meta-modelo de UML para Classe

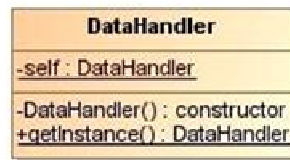

Figura 2.2: Exemplo de um modelo de uma classe de acordo com o Meta-modelo simplificado

Para ilustrar a aplicação de Meta-modelos, considere a UML. Com UML podemos descrever o modelo de uma aplicação qualquer. Por exemplo, podemos descrever um sistema web, especificando cada classe que trata a sessão no servidor, o acesso ao banco de dados, etc. Já o meta-modelo de UML descreve como os modelos UML devem ser constituídos. Especificando, por exemplo, que toda classe deve possuir um nome, pertencer a um pacote, etc.

Há várias linguagens propostas para descrever meta-modelos, chamadas meta-linguagens. Alguns exemplos de meta-linguagens são: MOF (*Meta Object Facility*)[22], Ecore [16], KM3 (*Kernel Meta Meta Model*)[20], etc. O padrão de meta-metalinguagem proposto pela OMG é MOF. Ela foi usada, por exemplo, para definir o meta-modelo de UML, OCL (Object Constraint Language) [42] e os outros Meta-modelos propostos pela OMG.

2.1.2 Transformações entre Modelos

MDA possui um alto potencial de automação alcançado através de transformações entre modelos, especialmente entre os modelos PIM, PSM e o código em sintaxe concreta. As transformações definem como os modelos de entrada podem ser mapeados para os modelos de saída, baseados nos seus meta-modelos. Por exemplo, a partir de um PIM de um sistema web, podemos executar algumas transformações sobre ele para gerarmos os PSMs descrevendo a implementação desse sistema usando EJB, outras transformações para criar o banco de dados relacional do sistema, e assim por diante. O mesmo acontece para as transformações textuais sendo que ao invés de modelos de saída, são gerados artefatos em sintaxe concreta.

A infra-estrutura para as transformações em MDA possuem basicamente 3 elementos:

- *Linguagem de Transformação*: assim como existem os meta-modelos que descrevem os modelos, existem as linguagens de transformação que definem como as transformações devem ser definidas. Elas podem ser classificadas em vários tipos: transformações declarativas, transformações imperativas, transformações por templates, transformações textuais, etc. Alguns exemplos de linguagens que podem ser usadas como linguagens de transformações são: QVT (*Queries / Views / Transformations*) [41], que são transformações declarativas e imperativas; ATL (*ATLAS Transformation Language*) [3], também declarativas e imperativas; linguagens de programação, que são transformações imperativas; XSLT (*XSL Transformations*) [57], que são transformações textual; JET [26], que são transformações por templates; etc. A linguagem de transformação padrão proposto pela OMG é QVT, sendo que ATL apesar de não ser o padrão da OMG, está em conformidade com todos os padrões propostos por ela, como UML, MOF e OCL.

- *Definição da Transformação*: são as transformações escritas nas linguagens ci-

tadas anteriormente que, baseadas nos meta-modelos dos modelos de entrada, geram os modelos de saída, também baseados em seus meta-modelos. É através dessas transformações que é definido como um PIM (por exemplo, o sistema web) vai gerar um PSM (a implementação do sistema em EJB), por exemplo.

- *Ferramentas de Transformação*: são as ferramentas que dão suporte à execução das transformações. Alguns exemplos que podemos citar são: para executar transformações em QVT pode ser usado o Borland Together [10]; para transformações ATL, o ATL-DT [8]; para transformações XSLT, o Xalan [56]; entre outros.

Transformações não se restringem somente a transformar de PIM para PSM e de PSM para o código em sintaxe concreta. Elas podem ser concebidas para transformar de qualquer nível de modelo para qualquer outro, inclusive para o mesmo nível (por exemplo, transformações de PSM em PSM executando algum refatoramento entre nesses modelos).

2.1.3 Arquitetura Quatro Camadas da OMG

A OMG usa uma arquitetura de quatro camadas para mostrar os papéis dos seus padrões em MDA: Metameta-modelo, Meta-modelo, Modelos e Transformações. Na terminologia da OMG estas camadas são chamadas M0, M1, M2 e M3, que são descritos a seguir:

- Camada M0: As Instâncias. Esta camada trata as instâncias reais no sistema, seja como objetos na memória, tuplas do banco de dados, etc. Por exemplo, em um momento de um sistema web poderia existir um cliente com o nome "Sr. Madruga" que mora na "Avenida Brasília" na cidade "Cidade do México".

- Camada M1: O Modelo do Sistema. Essa camada contém os modelos do sistema, ou seja, abstrações das instâncias reais do sistema. Por exemplo, considerando

o sistema web tratado anteriormente, no M1 estaria o conceito da classe *Cliente*, com as suas propriedades *nome*, *rua* e *cidade*.

- Camada M2: Os Modelos dos Modelos. Os elementos que existem na camada M1 (por exemplo, classes, atributos, etc.) são por si mesmos instâncias dos elementos da camada M2. Em outras palavras, M2 conteria o meta-modelo dos modelos da camada M1. No exemplo da camada M1, o M2 seria o meta-modelo de UML.

- Camada M3: O Modelo dos Meta-modelos. Seguindo a mesma linha de raciocínio podemos considerar que os Modelos da camada M2 são instâncias de modelos de mais alto nível, os meta-meta-modelos. No caso do meta-modelo de UML, o meta-meta-modelo seria o meta-modelo MOF ou qualquer outro formalismo capaz de descrever o meta-modelo de UML, como o meta-modelo KM3, ECORE, etc.

Na Figura 2.3 podemos observar o framework completo de MDA. Com todos os seus elementos: os modelos, meta-modelos e transformações; e os relacionamentos entre eles na forma como definimos anteriormente.

2.2 Diagrama de Classes da UML2

A UML possui uma vasta gama de diagramas capazes de descrever tanto a estrutura, quanto o comportamento de um sistema de software. Dentre esses diagramas um amplamente adotado para descrever a estrutura de sistemas é o Diagrama de Classes. Trata-se de uma representação gráfica de uma visão estática do sistema que mostra uma coleção declarativa de elementos, tais como classes, pacote, entre outros, além de seus conteúdos e relacionamentos. Um diagrama de classes pode possuir certos elementos comportamentais, tais como operações. Contudo, o comportamento desses elementos só é expresso através de outros diagramas, ou artefatos.

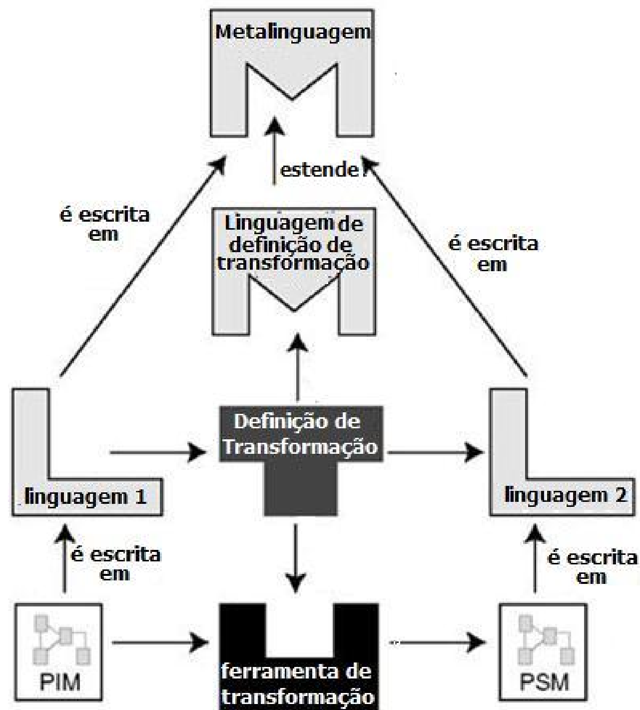

Figura 2.3: Framework de MDA. Extraído e traduzido de Kleppe&Warner [31]

Normalmente, são construídos vários diagramas de classes para mostrar a visão estática completa de todo um sistema. Diagramas de classes individuais necessariamente não significam divisões no modelo do sistema, mas podem somente mostrar divisões lógicas tais como pacotes ou diferentes perspectivas do sistema.

Dado o fato da UML2 ser bastante abrangente, o número de artefatos que podem ser usados no diagrama de classes, e as características desses artefatos, é extenso. Por esse motivo, destacaremos nesse trabalho os mais relevantes à nossa abordagem.

2.2.1 Classe

Classe representa um conceito com a qual a aplicação pode ser modelada. Uma classe é um descritor para um conjunto de objetos que compartilha os mesmos atributos, operações, relacionamentos e comportamentos. Uma classe pode modelar entidades físicas

(como aviões, carros, pessoas, etc.), entidades da lógica do negócio (como um acordo, pessoa jurídica, etc.), entidades lógicas (como algoritmo, etc.), entidades da própria aplicação (como uma sessão, um formulário, etc.), entidades computacionais (como uma tabela *hash*) ou entidades comportamentais (como uma tarefa).

A Figura 2.4 mostra um exemplo de uma classe, destacando as características pertencentes à classe que são tratadas nesse trabalho:

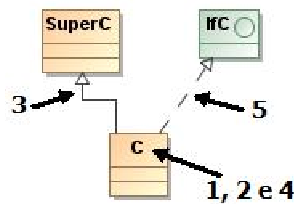

Figura 2.4: Características de classe tratadas neste trabalho

1. nome (*name*): toda classe deve possuir um nome único que a identifique. Esse nome é formado com a composição do nome da classe, junto com o nome da hierarquia de entidades na qual ela está contida, normalmente um pacote, mas pode ser outra classe também;

2. visibilidade (*visibility*): as classes possuem uma visibilidade relacionada à entidade na qual ela está contida. Essa visibilidade especifica como essa classe pode ser acessada pelas outras classes. Em UML existem quatro tipos de visibilidades predefinidas: **public**, significando que qualquer entidade que consiga alcançar o *container* da classe pode ver essa classe; **protected**, somente os elementos do próprio *container* e os dos *subcontainers* podem ver essa classe; **private**, somente a entidade proprietária desse elemento pode ter acesso a ele; **package**, somente os elementos do mesmo pacote ou dos subpacotes podem ver essa classe;

3. super classes (*superClass*): UML segue o paradigma de orientação a objeto, por-

tanto uma classe pode herdar de outra recebendo da classe pai toda a estrutura, relacionamentos e comportamentos dela. UML permite generalizações múltiplas;

4. classe abstrata: é uma característica que sinaliza se a classe pode ou não ser instanciada;

5. interfaces realizadas (*realizedInterface*): interface é um padrão de comportamento para algumas entidades do projeto. Uma classe pode realizar várias *interfaces* simultaneamente (interfaces são tratadas com mais detalhes na seção 2.2.4).

2.2.2 Operação

Uma operação especifica uma transformação no estado do objeto possuidor da operação (e possivelmente no estado do resto do sistema alcançável por esse objeto) ou uma consulta com o valor de um dado acessível através desse objeto.

A Figura 2.5 mostra um exemplo de uma operação, destacando as características pertencentes à operação que são tratadas nesse trabalho:

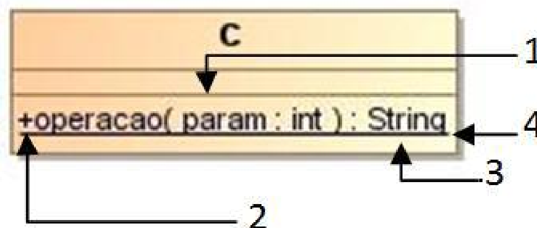

Figura 2.5: Características de operações tratadas neste trabalho

1. assinatura: toda operação possui um nome único formado por vários elementos dentre. Neste trabalho consideramos como assinatura formada por três elementos: nome da operação, o nome dos parâmetros e o nome dos tipos dos parâmetros da operação;

2. visibilidade: as operações possuem uma visibilidade que especifica quais outras entidades podem ter acesso a ela;

3. tipo do retorno: é o tipo do valor retornado quando essa operação é invocada. Uma operação pode retornar um tipo do próprio projeto do sistema, um tipo primitivo ou não retornar nada (*void*);

4. escopo: especifica a qual escopo essa operação pertence, ou seja, se ela pertence a qualquer instância de uma classe (*classifier*) ou somente a uma instância específica (*instance*).

2.2.3 Atributo

Um atributo representa uma propriedade que todas as instâncias de uma classe podem conter. Uma classe pode ter qualquer número de atributos, que podem ser resgatados ou capturados durante a execução de alguma operação.

A Figura 2.6 mostra um exemplo de um atributo, destacando as características pertencentes a atributo que são tratadas nesse trabalho:

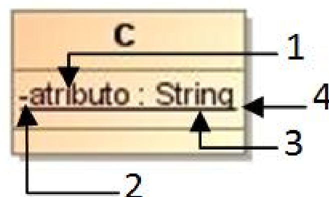

Figura 2.6: Características de atributos tratadas neste trabalho

1. nome: é um valor usado para identificar um atributo. Cada Atributo deve possuir um nome único aos moldes do que foi tratado em classe;

2. visibilidade: especifica quais outras entidades do projeto podem ter acesso ao atributo;

3. tipo: designa uma classe ou tipo de dado que o valor desse atributo deve obedecer;

4. escopo: especifica a qual escopo esse atributo pertence, ou seja, se ele pertence a qualquer instância de uma classe (*classifier*) ou somente a uma instância específica (*instance*).

2.2.4 Interface

Uma interface é uma coleção de operações que especificam serviços de uma classe ou componente, descrevendo o comportamento dessas entidades visível externamente. Raramente aparece sozinha, em geral são anexadas à classe ou ao componente que a realiza. Elas são completamente abstratas, e não especificam qualquer estrutura (não podem incluir atributos) nem qualquer implementação.

A Figura 2.7 mostra um exemplo de uma interface, destacando as características pertencentes a interface que são tratadas nesse trabalho:

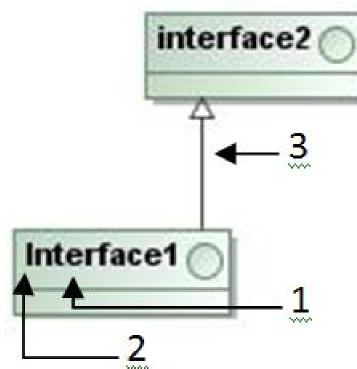

Figura 2.7: Características de interface tratadas neste trabalho

1. nome: toda interface deve possuir um nome único que a identifique. O padrão de nomes deve seguir o mesmo que foi descrito para classe, anteriormente;

2. visibilidade: especifica quais outras entidades do projeto podem realizar ou herdar dessa interface;

3. herança: UML permite que uma interface herde de outras interfaces fazendo com que a interface filha possua não só as operações contidas nela, mas também as contidas nas interfaces mãe. Sendo assim, se uma classe realizar a interface filha, ela deve, além de implementar os métodos da interface realizada, realizar os métodos das interfaces mãe.

2.2.5 Pacote

Um pacote é um agrupamento de elementos de um diagrama. Esses elementos podem ser quaisquer elementos de modelagem, incluindo outros pacotes. Ele tem a função de ser um mecanismo de propósito geral para organizar elementos semanticamente relacionados em grupos.

A Figura 2.8 mostra um exemplo de um pacote, destacando as características pertencentes a pacote que são tratadas nesse trabalho:

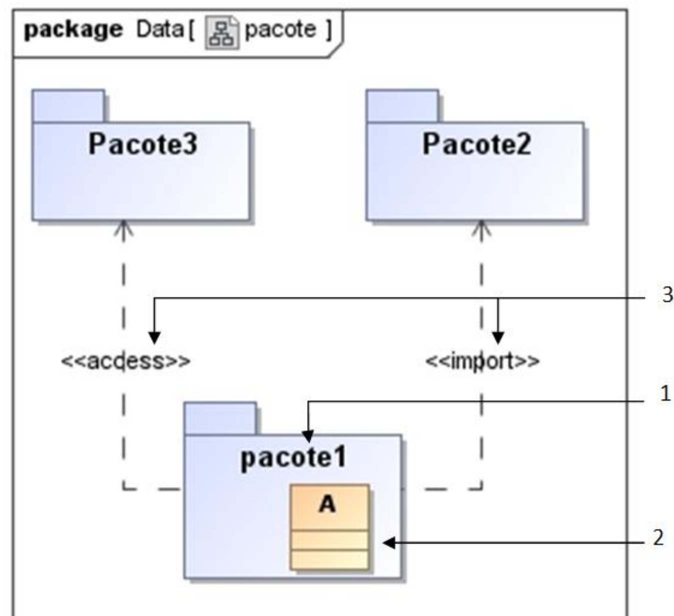

Figura 2.8: Características de pacotes tratadas neste trabalho

1. nome: um valor único usado para identificar o pacote. A hierarquia de nomes segue a mesma mostrada para a hierarquia de nomes de classes mostrado na seção de Classe (seção 2.2.1);

2. elementos: são os elementos que estão agrupados neste pacote;

3. relacionamentos: nesse trabalho tratamos especialmente de dois relacionamentos, «*access*» e «*import*». O relacionamento *access* significa que os elementos do pacote de origem podem acessar os serviços e estender (ou seja, especializar ou realizar) os elementos do pacote destino. O relacionamento «*import*» possue a mesma semântica do relacionamento «*access*» com o diferencial de que os elementos do pacote origem pode ainda acessar os serviços e estender os elementos dos pacotes que o pacote destino acessa ou importa.

2.2.6 Associação

Uma associação é um relacionamento entre duas ou mais classes que descreve uma conexão entre elas. Uma mesma classe pode participar de várias associações ou até mesmo de uma mesma associação várias vezes. A função da associação é interconectar o sistema que está sendo projetado de forma que o esse sistema funcione e faça sentido.

A Figura 2.9 mostra um exemplo de uma associação, destacando as características pertencentes a associação que são tratadas nesse trabalho:

1. nome: as associações possuem nomes para serem identificadas. A identificação de uma associação é através do seu nome e dos nomes dos elementos que fazem parte da associação;

2. membros: são os elementos (*MemberEnd*) que fazem parte da associação;

3. visibilidade: essa propriedade diz respeito a cada *MemberEnd* da associação. Ela especifica quais outras entidades do projeto podem acessar cada *memberEnd*;

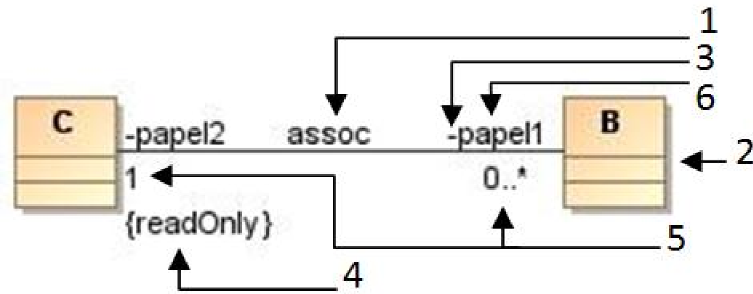

Figura 2.9: Características de associações tratadas neste trabalho

4. somente leitura: assim como a propriedade anterior, essa diz respeito a cada *MemberEnd* da associação. Ela especifica que os elementos que acessam essa associação podem somente ler o MemberEnd, contudo não podem modificá-lo;

5. multiplicidade: essa propriedade especifica a quantidade de elementos que podem estar envolvidos na associação;

6. papel: descreve o nome que como o qual cada membro da associação (*MemberEnd*) pode acessar outro membro (*MemberEnd*).

2.3 Teste de Design

Um teste é uma atividade executada para avaliar a qualidade de um produto, ou para melhorar esse produto identificando defeitos e problemas [33]. Existem vários tipos de testes, alguns que podemos citar são: testes de unidade, que verificam se um programa (ou parte dele) possui um comportamento esperado; testes de cobertura, que verificam o quão o código está testado; dentre outros.

Dentre vários tipos de teste existentes, um deles é são os **Testes de Design**. Eles são regras especificadas na forma de algoritmos que verificam propriedades do design de um código implementado [28; 53]. Essas propriedades devem ser providas a partir do design do software, seja a partir de um modelo conceitual presente somente na cabeça

do projetista do sistema, seja através de uma série de diagramas UML que descrevem o sistema.

Testes de design se assemelham a testes de integração, pois ambos verificam se um código implementado mantêm as características do *projeto do software*. Porém, os testes de integração verificam se o código desenvolvido mantêm uma determinada funcionalidade projetada. Já teste de design verifica se a estrutura desse código segue uma estrutura pré-determinada, sem considerar as funcionalidades que esse código possa ter.

Para desenvolver testes de design são necessários dois componentes chave: (1) Analisador da estrutura do código, uma ferramenta capaz de extrair e disponibilizar informações sobre as entidades presentes no código, e seus relacionamentos; (2) *Framework* de teste, um *framework* capaz de colher informações do extrator e verificar se elas condizem com o design esperado do software.

Nesse sentido, foi adotada uma infra-estrutura para a construção de testes de design para testar programas na linguagem Java. Para o analisador da estrutura do código adotamos uma biblioteca chamada Design Wizard [15]; detalharemos na próxima seção essa biblioteca. E como *framework* de teste foi adotado o JUnit [30]. A idéia central dos testes de design é usar a biblioteca Design Wizard para extrair a estrutura do código do sistema a ser testado, e usando o *framework* JUnit, verificar se a estrutura extraída possui a estrutura esperada. O Código Fonte 2.1 mostra um exemplo de teste de design, que verifica para dado o sistema *project* (linhas 2-3) existe uma classe chamada *A* (linhas 4-5) e ainda se a classe *A* não invoca diretamente nenhum método ou atributo da classe *B* (linhas 6-8).

```
1 public void testCommunication ( ) {
2         DesignWizard dw;
3         dw = new DesignWizard("project.jar");
4         designwizard.ui.Class clazz;
5         clazz = dw.getClass("A");
6         Set<String> usedBy;
7         usedBy = clazz.getClassesUsedBy();
8         assertFalse(usedBy.contains("B"));
9 }
```

Código Fonte 2.1: Exemplo de Teste de Design

2.4 DesignWizard

DesignWizard é uma biblioteca escrita em Java para dar suporte à realização de testes de design sobre aplicações também desenvolvidas em Java. Para a realização dos testes de design essa biblioteca atua como o analisador da estrutura do código [28; 15]. Ela é responsável por extrair informações sobre o código, organizar essas informações e expô-las através de uma API que ofereça métodos para a aquisição dessas informações.

A Figura 2.10 descreve a arquitetura do DesignWizard. Essa ferramenta é composta por um extrator, um tradutor, um gerenciador do design extraído e a API do DesingWizard.

O componente extrator usa o manipulador de bytecode ASM [7] que é uma ferramenta que usa análise estática do código para obter as instruções do bytecode da aplicação. O componente tradutor recebe as instruções do bytecode e as traduz para uma notação mais fácil de ser compreendida e manipulada por humanos.

As informações produzidas pelo tradutor são passadas para o gerenciador do design extraído. Este componente organiza essas informações na forma de um grafo, sendo os

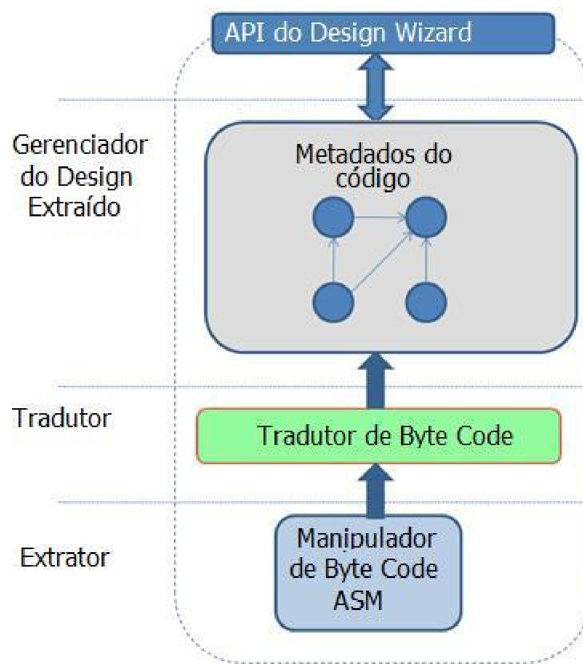

Figura 2.10: Arquitetura do DesignWizard

nós desses grafo as entidades presentes no código (classe, atributo, método, etc.) e as arestas os relacionamentos entre essas entidades. Por exemplo, se uma classe *C* possuir os atributos *a, b*, então serão criados 3 nós, um para cada entidade, além disso, serão criadas duas arestas, uma entre o nó da classe *C* e o atributo *a* e outra entre os nós de *C* e *b*. Por fim, as informações sobre essa estrutura de dados podem ser acessadas através da API do DesignWizard. Essa API funciona como fachada entre a representação do código e os testes de design usando o JUnit.

Destacamos algumas métodos da API do Design Wizard, mais informações podem ser encontradas no *JavaDoc* do Design Wizard [15].

A principal entidade da API do Design Wizard é a DesignWizard. Dentre os métodos dessa entidade destacamos:

- *new DesignWizard(caminhoDoProjeto:String)*. É um método construtor responsável por extrair a estrutura do código passado como parâmetro;

- *getClass(nomeDaClasse:String):ClassNode*. Recupera da estrutura extraída uma representação da classe com o nome passado por parâmetro.

- *getPackage(nomeDoPacote:String):PackageNode*. Recupera da estrutura extraída uma representação do Pacote com o nome passado por parâmetro.

Para representar uma classe ou interface o Design Wizard usa a entidade *ClassNode*. Dentre os métodos dessa entidade destacamos:

- *getSuperClass():ClassNode*. Se a entidade for uma classe, esse método extrai do código a super classe da classe a qual o método foi invocado;

- *getModifiers():Modifier*. Esse método é responsável por extrair do código a visibilidade da entidade;

- *isAbstract():boolean*. Esse método é responsável por expor se a entidade extraída é ou não abstrata;

- *isInterface():boolean*. Esse método é responsável por expor se a entidade extraída é uma classe ou uma interface;

- *getImplementedInterfaces():Set<ClassNode>*. Esse método é responsável por extrair do código todas as interfaces que são implementadas (se a entidade for uma classe) ou estendida (se a entidade for uma interface);

- *getMethod(nomeDoMetodo:String):MethodNode*. Esse método é responsável por extrair da classe, um método com a mesma assinatura passada por parâmetro.

Para representar um método o Design Wizard usa a entidade *MethodNode*. Dentre os métodos dessa entidade destacamos:

- *getReturnType():ClassNode*. Esse método é responsável por extrair do método o seu tipo de retorno;

- *isAbstract():boolean*. Esse método é responsável por expor se o método extraído é ou não abstrato;
- *isStatic():boolean*. Esse método é responsável por expor se o método extraído é ou não estático;
- *getModifiers():Modifier*. Esse método é responsável por extrair do código a visibilidade do método.

Para representar um atributo o Design Wizard usa a entidade *FieldNode*. Dentre os métodos dessa entidade destacamos:

- *getDeclaredType():ClassNode*. Esse método é responsável por extrair do atributo o seu tipo;
- *isAbstract():boolean*. Esse método é responsável por expor se o atributo extraído é ou não abstrato;
- *isStatic():boolean*. Esse método é responsável por expor se o atributo extraído é ou não estático;
- *getModifiers():Modifier*. Esse método é responsável por extrair do código a visibilidade do atributo.

Para representar um pacote o Design Wizard usa a entidade *PackageNode*. Dentre os métodos dessa entidade destacamos:

- *getAllClasses():Set<ClassNode>*. Esse método é responsável por extrair do pacote todas as entidades pertencentes a ele ou a seus subpacotes.

Capítulo 3

Verificação de Artefatos dos Diagrama de Classes UML

Dado o problema da verificação dos artefatos do diagrama de classes UML, a solução proposta nesse trabalho é a verificação desses artefatos através da criação de testes de design baseados em templates de código. Utilizamos templates, pois identificamos que os testes de design para a verificação dos artefatos do diagrama de classes seguem um padrão, e esse padrão consegue ser especificado através de *templates* de teste.

O uso de templates para geração de testes é utilizado há algum tempo para a geração de diversos tipos de testes [44]. Na nossa abordagem, propusemos um template de teste para cada tipo de artefato no diagrama de classe.

Neste capítulo, mostraremos a forma como foram adotados os templates de teste. Inicialmente, apresentaremos como adotamos templates para a verificação das classes do diagrama de classe. Em seguida, apontaremos os demais tipos dos artefatos do diagrama de classe e suas características que são cobertas pelo nosso trabalho. Mostraremos ainda alguns artefatos que não podem ser verificados, e o porquê desse feito. Depois disso, ilustraremos a aplicação da nossa abordagem para verificação das características referentes a um trecho do diagrama de classe de um sistema Web. Por fim,

teceremos alguns comentários sobre como essa abordagem alcança todos os artefatos tratados, e como nossa abordagem pode ser estendida para tratar outros artefatos.

3.1 Template do Teste de Design

A maneira como nós adotamos templates de testes para a verificação de artefatos dos diagramas de classe pode ser vista na Figura 3.1. Nessa figura podemos perceber que existem duas fases principais: a aplicação dos templates para diagrama de classe; e a execução dos testes pela aplicação dos templates.

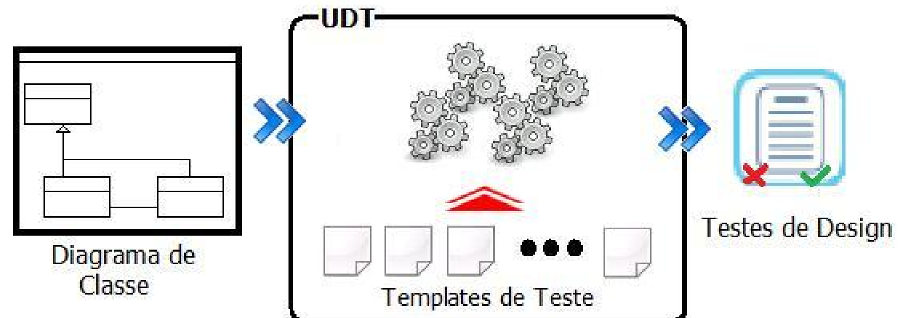

Figura 3.1: Processo geral para o uso de templates para a verificação de diagramas de classe

Inicialmente, definimos um catálogo de templates genéricos capazes de identificar cada tipo de artefato tratado neste trabalho: classe, operação, atributo, pacote, interface e associação. Esses templates são métodos genéricos formados por trechos estáticos e algumas *tags*. Os trechos estáticos, como o próprio nome sugere, identificam a parte comum a qualquer teste de uma instância de um determinado artefato. Os *tags*, sinalizados com “<” e “>”, são os trechos variáveis desses testes, e devem ser substituídos pelas especificidades das instâncias de cada artefato no diagrama de classe. Em outras palavras, cada instância de um artefato no diagrama de classe gera um método que a

testa, substituindo os *tags* do seu respectivo *template* pelas informações presentes no diagrama de classe.

Nas subseções a seguir, mostraremos os *templates* de testes de design para todos os artefatos tratados nesse trabalho. Todos esses *templates* compartilham algumas propriedades:

- **nome da entidade**: vários tags dos templates são referentes ao nome completo das entidades. O nome completo de uma entidade é o seu nome, seguido da hierarquia de nomes das outras entidades onde esta entidade está contida. Todos esses nomes são separados por ".". Por exemplo, a classe *Classe1* está contida no pacote *pacote2* que por sua vez está contido no pacote *pacote1*. Dessa forma, o nome completo da *Classe1* deve ser *pacote1.pacote2.Classe1*.

- **nome do teste**: os métodos para cada artefato específico seguem um padrão que identifica o tipo de verificação. Todo método começa com o "test" e seguidos do nome do artefato a ser testado, identificando qual o artefato que está sendo testado. Logo após, vem um diferenciador (um contador, por exemplo) que garante uma assinatura distinta para qualquer métodos, identificado pela *tag* <*dif*>. Por fim, vem o nome completo da entidade a ser testada, substituindo o "." por "_", pois nomes de métodos não podem possuir ".". Por exemplo, considere a classe mostrada no item anterior, *Class1*, o seu nome no método seria *pacote1_pacote2_Classe1*;

- **templates de mensagens explicativas**: as verificações realizadas através de uma assertiva (*assertTrue*, *assertFalse*, dentre outras.) possuem uma *string template* com uma mensagem que, caso a assertiva falhe, explica o porquê da falha do teste. Todas as mensagem estão em inglês somente[1], por uma questão de implementação e de mais fácil aceitação. E ainda, algumas dessas mensagens possuem a *tag*

[1] Estenderemos nossa solução para a adoção de um arquivo externo configurável para cada linguagem.

<*not*>, ela significa que de acordo com as informações do diagrama de classes, essas *tags* devem ou não ser substituídas pelo termo "*not*". Por exemplo, se a verificação for realizada para saber uma entidade não deve ser abstrata, então o *tag* <*not*> deve ser substituído. Caso contrário, ele não deve ser substituído por nada.

Os templates que serão mostrados estão divididos para ressaltar cada característica verificada. O código completo dos templates podem ser encontrados no apêndice A.

3.1.1 Classe

Inicialmente, ilustraremos a nossa técnica de criação de testes de design baseados em templates, mostrando como ela é usada para a verificação do artefato classe. O template para **classe** verifica as características de classe tratadas na seção 2.2.1: *nome, visibilidade, superClasse, escopo* e *interfaces realizadas*. O template está organizado da seguinte forma:

- Assinatura do teste. O código 3.1 exibe o trecho do método de teste que cria a assinatura do teste de design, identificando que se trata de um teste de design para a classe a qual o template está sendo aplicado.

```
1 public void testClass<dif><NomeDaClasse>()
2     throws IOException, InexistentEntityException{
3 ...
```

Código Fonte 3.1: Trecho do Template do Teste de Design para Classe referente a assinatura do método de teste.

- Existência da Classe. O código 3.2 exibe o método de teste substitui o *tag* <*NomeDaClasse*> pelo nome completo da classe. A verificação da existência da

classe testada se dá com a invocação do método *getClass* da biblioteca DesignWizard. Esse método retorna uma representação da classe (um *ClassNode*) presente no código-fonte que possui o mesmo nome passado por parâmetro. Se não existir nenhuma classe com esse nome no código, a exceção *InexistentEntityException* será lançada, informando que a classe com o nome indicado não existe.

```
1 ...
2   ClassNode aClass = dw.getClass("<NomeDaClasse>");
3 ...
```

Código Fonte 3.2: Trecho do Template do Teste de Design para Classe referente a verificação da existência da classe.

- Classe Abstrata. Se a classe no diagrama de classe for abstrata, o código 3.3 substitui o *tag* *<TrueFalse>* pelo termo "*True*", caso contrário, deve substituir por "*False*". Na mensagem explicativa o *tag* *<NomeDaClasse>* deve ser substituído pelo nome completo da classe testada;

```
1 ...
2   assert<TrueFalse>("The class <NomeDaClasse> must <not> be"
3       +" abstract",
4       aClass.isAbstract());
5 ...
```

Código Fonte 3.3: Trecho do Template do Teste de Design para Classe referente a verificação se a classe é abstrata.

- Herança de Classe. O código 3.4 exibe o trecho do método de teste que verifica se a classe testada deve herdar de outra classe do diagrama. Caso, no diagrama de classe, a classe testada não herde diretamente de nenhuma outra, essas linhas devem suprimidas do teste de design a ser criado. É fato que UML permite herança múltipla entre classes. Contudo, consideramos que todos os diagramas de

classe serão modelados de forma que somente existirão heranças simples. Isso se deve ao fato de somente consideramos que diagramas de classe modelam aplicações em Java, que não permite herança múltipla. As linhas 3-4 possuem uma *string template* para a formação de uma mensagem explicativa seguindo o padrão da mensagem do item anterior;

```
1 ...
2   ClassNode superClass = dw.getClass("<NomeDaSuperClasse>");
3   assertEquals("The class <NomeDaClasse> must extend the class "
4          +"<NomeDaSuperClasse>",superClass ,
5          aClass.getSuperClass());
6 ...
```

Código Fonte 3.4: Trecho do Template do Teste de Design para Classe referente a verificação da hierarquia de classes.

- Visibilidade da Classe. O código 3.5 exibe o trecho do método de teste que verifica se a classe no diagrama de classe possui a mesma visibilidade da classe mapeada no código. A linha 2 captura todos os modificadores da classe no código através do método *getModifiers* do DesignWizard. A linha 8 contem uma mensagem explicativa, formada da mesma forma que as dos itens anteriores. E as linhas 3-5 verificam se, dentre esses modificadores, está contido o modificador mapeado para a visibilidade da classe no diagrama;

```
1 ...
2   Collection <Modifier> modifs = aClass.getModifiers();
3   assertTrue("The visibility of class <NomeDaClasse> must be"
4          +" <visibility>",
5          modifs.contains(Modifier.< visibility >));
6 ...
```

Código Fonte 3.5: Trecho do Template do Teste de Design para Classe referente a

verificação da visibilidade da classe.

- Realização de Interfaces. No código 3.6, as linhas 2-3 instanciam um *array* de *strings* com os nomes completos de todas as interfaces que essa classe deve realizar, de acordo com o diagrama de classe. As linhas 4-11 executam um *for* que varre esse *array* de *strings*. No corpo desse *for* são realizadas as ações para a verificação: a linha 5 captura uma representação da interface no código (caso essa interface não exista, uma exceção será lançada); as linhas 6-7 capturam todas as interfaces que a classe testada realiza através do método *getImplementedInterfaces* da biblioteca DesignWizard. Por fim, as linhas 8-10 verificam se a interface capturada pela atual iteração do *for* está presente dentre as interfaces realizadas pela classe extraída do código. Caso não esteja presente, uma mensagem indicando o erro é criada.

```
1  ...
2    String[] superInterfaces =
3       {"<NomeDaInterface1\>" , "<NomeDaInterface2>" ,...};
4    for (String superInterface : superInterfaces) {
5      ClassNode classnodeInterface = dw.getClass(superInterface);
6      Set<ClassNode> interfacesExtend =
7          aClass.getImplementedInterfaces();
8      assertTrue("The class <NomeDaClasse> must realize the"
9          +" interface " + superInterface,
10         interfacesExtend.contains(classnodeInterface));
11   }
```

Código Fonte 3.6: Trecho do Template do Teste de Design para Classe referente a verificação da realização de interfaces.

Vale ressaltar também que o *template* para testar classe não verifica algumas características importantes de classe, tais como atributos, operações, associações, dentre outras.

Isso se deve ao fato de que a verificação dessas características é bastante complexa. E, sendo assim, decidimos que verificação dessas características deveria ser realizada através de templates específicos. Alguns outros fatores que nos motivaram a tomar essa decisão foram:

1. Código de teste mais limpos: dividindo em códigos de testes distintos, os testes criados são menores e mais fáceis de serem entendidos;

2. Identificação de vários erros simultaneamente: se a verificação de atributos, operações e associações fossem realizadas dentro da verificação de classe, quando ocorresse um erro, o método de teste iria parar e não identificaria erros em outras características que seriam verificadas posteriormente;

3. Reuso de teste: características comuns a vários artefatos são verificadas através de um mesmo template. Por exemplo, como uma interface é um tipo especial de classe com um estereótipo então ela também pode possuir operações. Sendo assim, para gerar os testes para verificar as operações tanto de classe e como de interface podem ser utilizados o mesmo *template* de teste.

3.1.2 Operação

O *template* para os testes de operações são capazes de criar testes para verificar todas as operações do diagrama. Ele não se restringe somente a operações de classes, alcança também operações interfaces, dentre outras. O template para operações verifica as características referentes às operações de UML tratadas na seção 2.2.2: *nome, tipo de retorno, visibilidade* e *escopo*. O template está organizado da seguinte forma:

- Assinatura do teste. O código 3.7 exibe o trecho do método de teste que cria a assinatura do teste de design, identificando que se trata de um teste de design para a operação a qual o template está sendo aplicado.

```
1 public void testMethod<dif><MethodName>()
2     throws IOException, InexistentEntityException{
3 ...
```

Código Fonte 3.7: Trecho do Template do Teste de Design para Operações referente assinatura do método de teste.

- Existência da Operação. O código 3.8 exibe o trecho do método de teste que recupera do código Java uma representação da classe com o mesmo nome da classe na qual a operação está contida (linha 2). Em seguida, extrai dessa representação um método com a mesma assinatura da operação que está sendo verificada (linha 3). O *tag* <MethodName> deve ser substituído pela assinatura no método. No design Wizard, a assinatura de um método é composta pelo *nome do método*, em seguida, entre parênteses, os nomes dos *tipos dos parâmetros* do método, separados por vírgula e na ordem como foram declarados. Dessa forma, verificando a assinatura do método, além de verificar se o nome do método está correto, verifica-se também se o tipo dos parâmetros desse método está correto, pois se algum dos parâmetros for declarado na implementação numa ordem diferente ou com tipos diferentes, o método com a assinatura esperada não será encontrado e o teste vai falhar indicando que o método não existe;

```
1 ...
2  ClassNode aClass = dw.getClass(\"<ClassName>");
3  MethodNode methodClass = aClass.getMethod("<MethodName>");
4 ...
```

Código Fonte 3.8: Trecho do Template do Teste de Design para Operações referente a verificação da existência da operação.

- Tipo de Retorno. Dada a existência da operação, o código 3.9 exibe o trecho do

método de teste que verifica se essa operação possui o mesmo tipo de retorno declarado no diagrama de classe. Inicialmente, o *tag* <*TypeName*> deve ser substituído pelo nome completo do tipo de retorno do método no diagrama. Caso a assertiva falhe, ela possui uma mensagem customizada explicando a causa da falha do teste. Essa mensagem explicativa é customizada substituindo as *tags* pelos seu valores correspondentes no diagrama. Caso o método testado seja um método construtor, essas linhas devem ser suprimidas do testes a ser criado;

```
1 ...
2  assertEquals("The return of the method <MethodName> "+
3       "from the class <ClassName>must be<TypeName>",
4       "<TypeName>", methodClass.getReturnType().getName());
5 ...
```

Código Fonte 3.9: Trecho do Template do Teste de Design para Operações referente a verificação do tipo do retorno.

- Método Abstrato. O código 3.10 exibe o trecho do método de teste que verifica se o método extraído do código deve ou não ser abstrato, de acordo com a forma como a operação foi declarada no diagrama. Se a operação no diagrama for abstrata, na linha 2 deve-se substituir a *tag* <*TrueFalse*> por "*True*", caso contrário, deve substituir por "*False*".

```
1 ...
2 assert<True False>("The method <MethodName> from "+
3       "the class <ClassName> must <not> be abstract",
4       methodClass.isAbstract());
5 ...
```

Código Fonte 3.10: Trecho do Template do Teste de Design para Operações referente a verificação se a operação é abstrata.

- Escopo do método. O código 3.11 exibe o trecho do método de teste que verifica o escopo da operação de acordo com a forma como a operação foi declarada no diagrama, ou seja, verifica se o método possui um escopo de classe (método estático) ou escopo de instância (método comum). Se a operação no diagrama possuir um escopo de classe, na linha 8 a *tag* *<TrueFalse>* deve ser substituído por "*True*", caso contrário, deve substituir por "*False*".

```
1 ...
2  assert<TrueFalse>("The method<MethodName> from the "+
3       "class <ClassName> must <not> be static",
4       methodClass.isStatic());
5 ...
```

Código Fonte 3.11: Trecho do Template do Teste de Design para Operações referente a verificação se a operação é estática.

- Visibilidade. O código 3.12 exibe o trecho do método de teste que verifica se a visibilidade do método extraído do código é a mesma do método no diagrama de classes. O padrão de substituição dos *tags* para a criação dessa verificação segue o mostrado para a visibilidade de classe na seção 3.1.1.

```
1 ...
2  Collection <Modifier> modifs = methodClass.getModifiers();
3  assertTrue("The visibility of The method<MethodName> from "+
4       "the class <ClassName> must be "+
5       "<visibility>",modifs.contains(Modifier.<visibility>));
6 }
```

Código Fonte 3.12: Trecho do Template do Teste de Design para Operações referente a verificação da visibilidade da operação.

3.1.3 Atributo

Nossos testes de design contemplam ainda os atributos dos elementos do diagrama de classes. A verificação mostrada aqui abrange as características dos atributos mostrados na seção 2.2.3. O *template* para atributo verifica as características: *nome, tipo de retorno, visibilidade* e *escopo*. Este *template* está organizado da seguinte forma:

- Assinatura do teste. O código 3.13 exibe o trecho do método de teste que cria a assinatura do teste de design, identificando que se trata de um teste de design para o atributo o qual o template está sendo aplicado.

```
1 public void testAttribute<dif><NomeDoAtributo> ()
2     throws IOException, InexistentEntityException {
3 ...
```

Código Fonte 3.13: Trecho do Template do Teste de Design para Atributos referente assinatura do método de teste.

- Existência do Atributo. O código 3.14 exibe o trecho do método de teste que verifica se existe uma classe com um atributo com os mesmos nomes das respectivas entidades testadas do diagrama. Se alguma dessa entidade não existir uma exceção será lançada e o teste falhará indicando a sua ausência;

```
1 ...
2   ClassNode aClass = dw.getClass("<NomeDaClasse>");
3   FieldNode attrClass = aClass.getField("<NomeDoAtributo>");
4 ...
```

Código Fonte 3.14: Trecho do Template do Teste de Design para Atributos referente a verificação da existência do atributo.

- Escopo. O código 3.15 exibe o trecho do método de teste que verifica o escopo do atributo de acordo com a forma como o atributo foi declarado no diagrama de

classe (escopo de classe ou escopo de instância). A criação desse teste segue o mesmo padrão do para a criação da verificação de escopo de operações, na seção 3.1.2;

```
1  ...
2    assertTrue("The attribute <NomeDoAtributo> of the class"
3       +" <NomeDaClasse> must <not> be static",
4       attrClass.isStatic());
5  ...
```

Código Fonte 3.15: Trecho do Template do Teste de Design para Atributos referente a verificação se o atributo é estático.

- Visibilidade. O código 3.16 exibe o trecho do método de teste que verifica se a visibilidade do atributo extraído do código é a mesma do atributo no diagrama. A criação desse teste segue o mesmo padrão do para a criação do testes de design mostrado nas seções 3.1.2 e 3.1.1;

```
1  ...
2    Collection<Modifier> modifs = attrClass.getModifiers();
3    assertTrue("The attribute <NomeDoAtributo> of the class"+
4       " <NomeDaClasse> must be <visibilidade>"
5       ,modifs.contains(Modifier.<visibilidade>));
6  ...
```

Código Fonte 3.16: Trecho do Template do Teste de Design para Atributos referente a verificação da visibilidade do atributo.

- Verificação de Tipo. O código 3.17 exibe o trecho do método de teste que verifica se o atributo possui o mesmo tipo do que foi declarado no diagrama. É importante frisar que essa verificação leva em consideração a multiplicidade do atributo. Se a

multiplicidade for 1 ou 0-1, o teste verifica se o tipo do atributo é o mesmo tipo do que está referenciado no diagrama de classe. Contudo, se a multiplicidade desse atributo for maior que 1, o teste verifica se o tipo desse atributo é uma *coleção*. O tipo da coleção a ser verificada é definido de acordo com as características do atributo: se o atributo for único (*isUnique*) e ordenado (*isOrdered*) o teste vai verificar se o tipo do atributo é *java.util.SortedSet*; se o atributo for somente único, o tipo verificado é *java.util.Set*; se o atributo for somente ordenado o tipo verificado é *java.util.List*; por fim, se o atributo não possuir nenhuma das características tratadas anteriormente, o tipo verificado é *java.util.Collection*;

```
1 ...
2   assertEquals("The attribute <NomeDoAtributo> of the class"
3      +" <NomeDaClasse> must return the type "+
4      "<TipoDoAtributo>", <TipoDoAtributo>,
5      attrClass.getDeclaredType().getName());
6 }
```

Código Fonte 3.17: Trecho do Template do Teste de Design para Atributos referente a verificação do tipo do atributo.

3.1.4 Associação

Associações são verificadas no código seguindo a abordagem proposta por *Akehurst et al.* [13], que considera que todos os membros de uma associação devem possuir um atributo representando cada *memberEnd* navegável dessa associação. Esses membros devem possuir ainda métodos *get* e *set* para manipular esses atributos. Nem todas as características abordadas por *Akehurst et al.* podem ser verificadas, pois a análise estática do código não fornece informação suficiente. Por exemplo, o máximo e mínimo de elementos na associação não podem ser tratados, pois esse tipo de informação só poderia se extraída monitorando o número de instâncias de uma classe em tempo de execução.

O template para associação deve ser aplicado para a criação de testes de design para cada *MemberEnd* navegável pertencente à associação. Para cada membro da associação é gerado um teste de design com as seguintes características:

- Assinatura do teste. O código 3.18 exibe o trecho do método de teste que cria a assinatura do teste de design, identificando que se trata de um teste de design para o cada membro da associação o qual o template está sendo aplicado;

```
1 public void testAssociation<dif><NomeFonte><NomeAlvo> ()
2           throws InexistentEntityException , IOException{
3 ...
```

Código Fonte 3.18: Trecho do Template do Teste de Design para cada Membro da Associação referente assinatura do método de teste.

- Papel na associação. O código 3.19 exibe o trecho do método de teste que verifica se a classe que é membro da associação possui um atributo com o mesmo nome do papel dos outros membros navegáveis da associação. Esse teste ainda verifica se essa classe possui os métodos *get* e *set* para esse atributo (linhas 4 e 5, respectivamente). Nesse teste, o *tag* *<NomeTipoFonte>* deve ser substituído pelo nome completo da classe membro da associação que está sendo testada. O *tag* *<papel>* deve ser substituído pelo nome do papel dos outros *memberEnd* navegáveis dessa associação;

```
1 ...
2   ClassNode c1 = dw.getClass("<NomeTipoFonte>");
3   FieldNode f1 = c1.getField("<papel>");
4   MethodNode getAssoc1 = c1.getMethod("get<papel>()");
5   MethodNode setAssoc1 = c1.getMethod("set<papel>()");
6 ...
```

Código Fonte 3.19: Trecho do Template do Teste de Design para cada Membro da Associação referente a verificação do papel de cada membro.

- Tipo na Associação. O código 3.20 exibe o trecho do método de teste que verifica se o atributo mapeado para a associação e seus métodos *get* e *set* utilizam o tipo esperado, ou seja, a *tag* <*NomeTipoAlvo*>. Dependendo das características da associação, se a multiplicidade for 0-1 ou 1, o *tag* <*NomeTipoAlvo*> deve ser substituído pelo nome completo do tipo do outro *memberEnd* navegável da associação. No entanto, se a multiplicidade for superior a 1, o tipo esperado é uma coleção especializada, logo, o *tag* <*NomeTipoAlvo*> deve ser substituído pelo mesmo padrão dos tipos das coleções explicadas na seção 3.1.3 (tipo de atributo). A substituição dos demais *tags* segue o mesmo padrão dos itens anteriores. As linhas 1-5 e 6-8 são referentes a verificação do tipos dos métodos *get* (tipo de retorno) e *set* (parâmetro do método), respectivamente. Por fim, As linhas 9-12 são referentes a verificação do tipo do atributo o qual a associação foi mapeado;

```
1  ...
2    assertEquals("the method get<papel> of the class "+
3       "<NomeTipoFonte> must return the type "+
4       "<NomeTipoAlvo>", "<NomeTipoAlvo>",
5       getAssoc1.getReturnType().getName());
6    assertEquals("the method set<papel> of the class "+
7       "<NomeTipoFonte> must return the type void",
8       "void", setAssoc1.getReturnType().getName());
9    assertEquals("the attribute <papel> of the class "+
10      "<NomeTipoFonte> must return the type "+
11      "<NomeTipoAlvo>", "<NomeTipoAlvo>",
12      f1.getDeclaredType().getName());
13 ...
```

Código Fonte 3.20: Trecho do Template do Teste de Design para cada Membro da Associação referente a verificação do tipo de cada membro.

- Visibilidade. De acordo com a abordagem de *Akehurst et al.*, o atributo que repre-

senta o *memberEnd* associado deve ser privado, e a visibilidade para esse *memberEnd* deve ser refletida nos métodos *get* e *set* de cada membro da associação. Sendo assim, o código 3.21 exibe o trecho do método de teste que verifica se o atributo da associação possui a visibilidade privada (linhas 2-5) e se a visibilidade dos métodos *get* (linhas 7-9) e *set* (linhas 10-13) estão de acordo com a visibilidade de cada membro navegável descrito no diagrama;

```
1  ...
2    Collection <Modifier> modifs = f1.getModifiers();
3    assertTrue("the attribute <papel> of the class "+
4       "<NomeTipoFonte> must be private"
5       ,modifs.contains(Modifier.PRIVATE));
6    modifs = getAssoc1.getModifiers();
7    assertTrue("the method get<papel> of the class "+
8       "<NomeTipoFonte> must be <visibilidade>"
9       ,modifs.contains(Modifier.<visibilidade>));
10   modifs = setAssoc1.getModifiers();
11   assertTrue("the method set<papel> of the class "+
12      "<NomeTipoFonte> must be <visibilidade>"
13      ,modifs.contains(Modifier.<visibilidade>));
14 ...
```

Código Fonte 3.21: Trecho do Template do Teste de Design para cada Membro da Associação referente a verificação da visibilidade de cada membro.

- Somente leitura. Essa característica é difícil de ser verificada através de análise estática, pois só é possível certificar se um artefato está sendo somente lido ou se está sendo alterado através da monitoração do software em tempo de execução. Porém, podemos verificar essa característica em associações com multiplicidade superior a 1 da seguinte forma: o teste verifica se o método *get* para o atributo o qual *memberEnd* foi mapeado retorna uma coleção através do método *Collec-*

tions.unmodifiable da API para coleções de Java. Se a associação não for se somente leitura ou se não tiver uma multiplicidade superior a 1, essas linhas devem ser suprimidas. O código 3.22 exibe o trecho do método de teste que verifica essa característica.

```
1  ...
2    Collection<MethodNode> methodsGetAssoc =
3        getAssoc1.getCalledMethods();
4    boolean isReadOnly = false;
5    for( MethodNode method: methodsGetAssoc ){
6        if(method.getName().equals
7                ("java.util.Collections.unmodifiable")){
8          isReadOnly = true;
9          break;
10       }
11   }
12   assertTrue("The method set<papel> must returns a "+
13                  "java.util.Collections.unmodifiable",
14       isReadOnly);
15 }
```

Código Fonte 3.22: Trecho do Template do Teste de Design para cada Membro da Associação referente a verificação dos membros de somente leitura.

3.1.5 Interface

Outro artefato que nossa solução também testa é interface. Todas as características de interface da seção 2.2.4 são verificadas pelos testes propostos neste trabalho. O template de Interface verifica as características: *nome* e *hierarquia de interfaces*. O template está organizado da seguinte forma:

- Assinatura do teste. O código 3.23 exibe o trecho do método de teste que cria a

assinatura do teste de design, identificando que se trata de um teste de design para a interface a qual o template está sendo aplicado.

```
1 public void testInterface<dif><NomeDaInterface>()
2         throws IOException, InexistentEntityException{
3 ...
```

Código Fonte 3.23: Trecho do Template do Teste de Design para Interface referente assinatura do método de teste.

- Existência da interface. O código 3.24 exibe o trecho do método de teste que verifica se existe uma interface na implementação com o mesmo nome da interface especificada no diagrama de classe. O nome que é verificado é o nome completo da interface, ou seja, o nome da interface seguindo a hierarquia de entidades do projeto. Diferente das verificações anteriores, onde a verificação da existência da entidade se dá quando a entidade é extraída do DesignWizard (*getClass*, *getAttribute*, dentre outras.). Para realizar a verificação de interface é necessário uma assertiva a mais, pois uma interface é extraída do código pelo DesignWizard como um *ClassNode* comum. A assertiva adicional (linhas 3-4) verifica se o *ClassNode* extraído realmente representa uma interface, através do método *isInterface*;

```
1 ...
2   ClassNode aClass = dw.getClass("<NomeDaInterface>");
3   assertTrue("The class <NomeDaInterface> must be a interface",
4     aClass.isInterface());
5 ...
```

Código Fonte 3.24: Trecho do Template do Teste de Design para Interface referente verificação da existência da Interface.

- Hierarquia de interfaces. O código 3.25 exibe o trecho do método de teste que verifica se uma interface presente no código estende de outras interfaces, de acordo

com a hierarquia de interfaces presente no diagrama de classe. As linhas 1-2 especificam os nomes completos das interfaces que a interface testada herda de acordo com o diagrama de classe. As linhas 4-11 iteram sobre esses nomes verificando se a interface no código também herda das outras interfaces com os mesmos nomes;

```
1  ...
2    String[] superInterfaces =
3       {"<superInterface1>","<superInterface2>",",...};
4    for (String superInterface : superInterfaces) {
5      ClassNode cnInterface = dw.getClass(superInterface);
6      Set<ClassNode> interfacesExtend =
7          aClass.getImplementedInterfaces();
8      assertTrue("The interface <NomeDaInterface>"+
9           " must extends from "+ superInterface,
10          interfacesExtend.contains(cnInterface));
11   }
12 }
```

Código Fonte 3.25: Trecho do Template do Teste de Design para Interface referente verificação da hierarquia da Interface.

3.1.6 Pacote

Por fim, nossa abordagem é capaz de criar testes de design para verificar os pacotes do diagrama de classe. O template para pacote verifica as características: *nome, hierarquia de pacote* e *relacionamentos entre pacote*. O template está organizado da seguinte forma:

- Nome do teste. O código 3.26 exibe o trecho do método de teste que cria a assinatura do teste de design, identificando que se trata de um teste de design para o pacote o qual o template está sendo aplicado.

```
1 public void testPackage<dif><NomeDoPacote>() throws java.io.
      IOException, InexistentEntityException {
2 ...
```

Código Fonte 3.26: Trecho do Template do Teste de Design para Pacote referente assinatura do método de teste.

- Existência do pacote. O teste de design gerado verifica se existe um pacote na implementação com o mesmo nome do pacote no diagrama de classe. O código 3.27 exibe o trecho do método de teste responsálvel por verificar essa característica. Nesse trecho o *tag* *<NomeDoPacote>* deve ser substituído pelo nome completo do pacote, ou seja, o nome do pacote seguindo a hierarquia de entidades do projeto, da mesma forma como foi explicado para a verificação do nome de classes (seção 3.1). Dessa forma, esse trecho além de testar a existência do pacote ainda verifica se a hierarquia de pacotes no código condiz com a hierarquia especificada no diagrama de classe;

```
1 ...
2   PackageNode thePackage = dw.getPackage("<NomeDoPacote>");
3 ...
```

Código Fonte 3.27: Trecho do Template do Teste de Design para Pacote referente verificação da existência do pacote.

- Relacionamento entre pacote (*«Access»* e *«Import»*). O código 3.28 exibe o trecho do método de teste responsável por verificar se os elementos do pacotes se relacionam de acordo com as diretrizes do relacionamento *import* e *access*. As linhas 2-3 especificam quais outros pacote esse pacote se relaciona (de acordo com as diretrizes de relacionamento entre pacotes, elucidada na seção 2.2.5). A linha 4 cria uma coleção com todas as entidades do pacote testado. As linhas 5-9

criam um coleção com todas as classes que podem ser extensíveis pelas classes dos pacote testado (ou seja, as classe internas ao próprio pacote e as classes dos pacotes apontados na linha 3). Por fim, as linhas 10-18 verificam se todos os elementos do pacote testado (*internalEntyties*) estendem (herança de classe - linhas 11-13, realização de interface - linhas 14-17) dos elementos do próprio pacote ou dos pacotes acessados.

```
1  ...
2    String[] importedPackages =
3            {"<PackageImp1>", "<PackageImp2>",... };
4    Set<ClassNode> internalEntyties = thePackage.getAllClasses();
5    Set<ClassNode> extentableEntiites = internalEntyties;
6    for (String aPackage : importedPackages) {
7       extentableEntiites.addAll(dw.getPackage(aPackage)
8          .getAllClasses());
9    }
10   for (ClassNode aEntity : internalEntyties) {
11     assertTrue(aEntity+" cannot extends the class " +
12         aEntity.getSuperClass(),
13         extentables.contains(aEntity.getSuperClass()));
14    for(ClassNode aInterface:aEntity.getImplementedInterfaces()){
15        assertTrue(aEntity+" cannot implements the interface"
16           + aInterface,extentableEntiites.contains(aInterface));
17     }
18   }
19 }
```

Código Fonte 3.28: Trecho do Template do Teste de Design para Pacote referente verificação dos relacionamentos entre os pacotes.

3.2 Exemplo de Aplicação dos Templates

Para exemplificar como nossa abordagem pode ser aplicada em um caso real, considere o desenvolvimento de um sistema Web, onde o designer desse sistema toma uma decisão estratégica de que o objeto que manipula os dados, *DataHandler*, deve seguir o padrão *Singleton*, para facilitar a sincronização do acesso aos dados. De acordo com o padrão Singleton, o objeto deve possuir: (1) seus construtores sendo privados, (2) um atributo estático, privado e com o tipo igual ao tipo da própria classe e (3) um método chamado *getInstance*, que deve retornar uma instância única desse objeto (ou seja, o atributo estático privado). Dessa forma, todas as classes que tratarem dados no sistema vão tratar somente com uma única instância. A Figura 3.2 exibe a classe do diagrama que projeta o acesso aos dados.

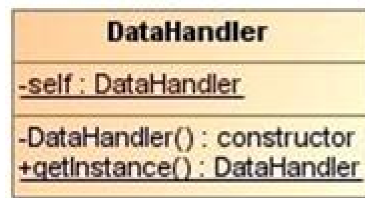

Figura 3.2: Classe de acesso ao banco de dados

Aplicando os templates de testes mostrados anteriormente para verificar a implementação dessa classe serão criados os códigos: Código 3.29 (verificação da classe *DataHandler*), Código 3.30 (verificação do atributo *self*), Código 3.31 e Código 3.32 (verificação dos métodos *construtor* e *getInstance*, respectivamente). Considere que a seleção e aplicação dos templates de testes de design deve ser feito manualmente pela equipe de desenvolvimento.

O código 3.29 é referente à verificação da existência da classe *DataHandler* (linha 3) da forma como ela foi especificada: concreta (linhas 4-5) e pública (linhas 6-8).

```
1 public void testClass1DataHandler () {
2   DesignWizard dw = new DesignWizard("/Project");
3   ClassNode aClass = dw.getClass("DataHandler");
4   assertFalse("The class DataHandler must not be abstract",
5      aClass.isAbstract());
6   Collection <Modifier> modifs = aClass.getModifiers();
7   assertTrue("The visibility of the class DataHandler must be public"
8      , modifs.contains(Modifier.PUBLIC));
9 }
```

Código Fonte 3.29: Verificação da classe DataHandler.

O código 3.30 é referente à verificação do atributo *self* da classe *DataHandler*. O teste verifica se o tipo desse atributo é o mesmo tipo da classe onde ele está contido (linhas 5-7) e se ele é um atributo estático (linhas 8 e 9) e privado (linhas de 10-13), conforme especificado na Fig. 3.2.

O código 3.31 é referente à verificação do construtor (especificado pelo DesignWizard por <*init*>) da classe *DataHandler* (linha 4). O restante do método de teste verifica as demais características do método testado. Em especial, ele verifica se esse construtor possui a visibilidade privada (linhas 9-11), que é uma característica própria do *design pattern Singleton*.

Por fim, o código 3.32 é referente à verificação do método *getInstance* da classe *DataHandler* (linhas 3 e 4). O restante do método de teste verifica as demais características do método testado. Em especial, verifica se ele retorna um tipo igual ao da classe que ele pertence (linhas 5-6), e se ele é um método estático (linhas 9-10), que é são características próprias do *design pattern Singleton*.

```
1  public void testAttribute1Self() {
2    DesignWizard dw = new DesignWizard("/Project");
3    ClassNode aClass = dw.getClass("DataHandler ");
4    FieldNode attrClass = aClass.getField("self");
5    assertEquals("The attribute self from the class DataHandler must be DataHandler",
6       "DataHandler",attrClass.getDeclaredType().getName());
7    assertTrue("The attribute seflfrom the class DataHandler must be static",
8       attrClass.isStatic());
9    Collection <Modifier> modifs = attrClass.getModifiers();
10   assertTrue("The visibility of the attribute self from the class DataHandler must be private",
11      modifs.contains(Modifier.PRIVATE));
12 }
```

Código Fonte 3.30: Verificação do atributo self da classe DataHandler.

```java
public void testMethod1construc() {
  DesignWizard dw = new DesignWizard("/Project");
  ClassNode aClass = dw.getClass("DataHandler ");
  MethodNode methodClass =  aClass.getMethod("<init>()");
  assertFalse("The method <init>() from the class DataHandler must not be abstract",
     methodClass.isAbstract());
  assertFalse("The method <init>() from the class DataHandler must not be static",
     methodClass.isStatic());
  Collection <Modifier> modifs = methodClass.getModifiers();
  assertTrue("The visibility of The method <init>() from the class DataHandler must be private"
    ,modifs.contains(Modifier.PRIVATE));
}
```

Código Fonte 3.31: Verificação do método Construtor da classe DataHandler.

```
1  public void testMethod2getInstance() {
2    DesignWizard dw = new DesignWizard("/Project");
3    ClassNode aClass = dw.getClass("DataHandler ");
4    MethodNode methodClass =  aClass.getMethod("getInstance()");
5    assertEquals("The return of the method getInstance from the class DataHandler must be "+
6      "DataHandler", "DataHandler", methodClass.getReturnType().getShortName());
7    assertFalse("The method getInstance from the class DataHandler must not be abstract",
8       methodClass.isAbstract());
9    assertTrue("The method getInstance from the class DataHandler must be static",
10      methodClass.isStatic());
11   Collection <Modifier> modifs = methodClass.getModifiers();
12   assertTrue("The visibility of The method getInstance from the class DataHandler must be public",
13      modifs.contains(Modifier.PUBLIC));
14 }
```

Código Fonte 3.32: Verificação do método getInstance da classe DataHandler.

3.2.1 Execução dos Testes Gerados

Nesta seção vamos mostrar como os testes de design funcionam para identificar erros na implementação de um sistema. Inicialmente, considere a classe mostrada na seção anterior (Seção 3.2), *DataHandler*, e os testes gerados para esta classe. Considere agora que esta classe foi implementada de uma maneira simplificada, como pode ser vista no código 3.33. Implementada assim, essa classe quebra completamente a decisão estratégica de que o acesso aos dados seja realizado através de um *Singleton*, pelo fato de possuir um construtor público.

```
1  ...
2  public class DataHandler {
3          public DataHandler(){
4                  ...
5          }
6          ...
7  }
```

Código Fonte 3.33: Implementação do *DataHandler* com o design alterado.

A Figura 3.3 mostra os erros no design apontados pelos testes de design, quando executados sobre a classe *DataHandler* mostrada no código 3.33. Abaixo do *label* Failure Trace são mostrados os *traces* resultantes da execução de cada um dos métodos de teste de design mostrado anteriormente. O primeiro *trace* é referente à verificação do método *construtor*. Perceba que ele falha, indicando que a visibilidade do método *construtor* deve ser privada: *"The visibility of The method <init>() from the class DataHandler must be private´´*. E o segundo e terceiro *traces* são referentes à verificação do método *getInstance* e do atributo *self*, respectivamente. Além disso, eles falham apontando que o método e o atributo não existem na implementação do *DataHandler*.

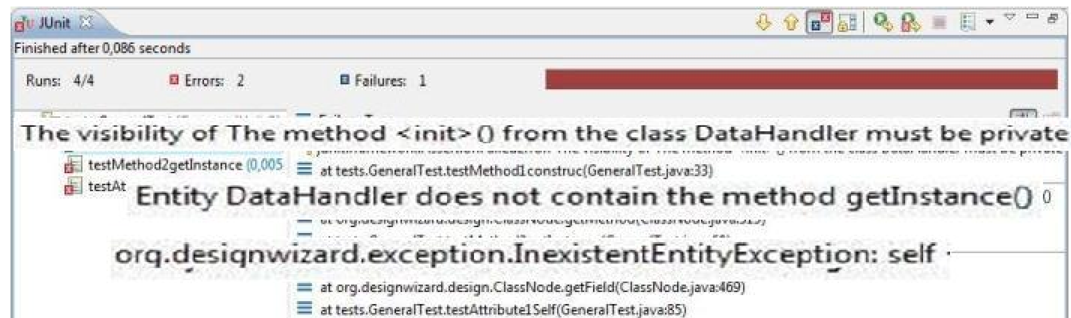

Figura 3.3: Resultado dos testes de Design para a classe *DataHandler*

3.3 Considerações Finais

Neste capítulo apresentamos uma abordagem para verificação de conformidade entre o design de um sistema especificado através de digramas de classes UML e o seu código fonte implementado em Java. A abordagem consiste basicamente de templates de testes de design para cada artefato da especificação de diagrama de classes proposto pela OMG.

Os templates de testes de design que foram mostrados podem ser aplicados para verificar características dos seguintes artefatos do diagrama de classe de UML: classe, atributo, operação, associação, interface e pacote. Esses artefatos foram selecionados, pois se mostram como os mais usuais para a descrição de modelos usando diagrama de classes. Exemplificamos a aplicação desses templates para a criação de testes de design para a verificação de um trecho de uma aplicação Web para acesso aos dados da aplicação.

Sobre o escopo das características verificadas, é fato que, segundo a especificação da OMG, classes possuem outras características além das tratadas anteriormente. Contudo, julgamos que essas características cobrem as mais usuais para a modelagem de diagramas de classe. Outro fato relevante nesse sentido é que várias características desses diagramas são muito difíceis de serem verificadas por análise estática do código. Por exemplo, o número máximo de instâncias de uma classe só poderia ser verificada

monitorando o código em tempo de execução e verificando se o número de instâncias de uma classe não ultrapassa o número de instâncias projetadas no diagrama de classe do sistema. Dado que nossa abordagem se baseia em templates de código, consideramos que propor novos templates ou estender os já existentes não é tão custoso, de modo que nossa abordagem mostra-se bastante flexível e extensível.

Capítulo 4

Geração Automática de Testes de Design

A abordagem proposta no capítulo anterior de criação de testes de design a partir de *templates* de teste possibilita a verificação de um design expresso através do diagrama de classe sobre a sua implementação em Java. Contudo, para a adoção em um ambiente real, ele possui alguns problemas práticos devido aos seguintes fatos:

1. Necessidade do aprendizado de uma nova API. Os testadores terão que conhecer a biblioteca do DesignWizard para poder aplicá-la bem, e especialmente estenderem os testes de design;

2. Testes muito detalhados. Devido à fina granularidade dos testes de design para cada artefatos do diagrama, a criação dos testes para estes artefatos pode ser muito custosa;

3. Criação manual. A aplicação dos templates sobre cada artefato específico do diagrama de classe teria que ser realizada manualmente. Essa ação, além de sobrecarregar o processo de desenvolvimento, aumenta o trabalho dos testadores, e fica bastante propícia a erros humanos na sua criação.

Para solucionar os problemas citados, propomos uma ferramenta chamada UDT (*UML Design Tester*) que visa gerar automaticamente testes de design capazes de verificar a conformidade entre a implementação em Java de um sistema e o seu design expresso através de diagramas de classe UML.

Neste capítulo, inicialmente faremos um *overview* sobre a arquitetura adotada para a ferramenta UDT. Em seguida, detalharemos a implementação de cada módulo específico da ferramenta, destacando os dois principais: o módulo de geração de modelos e o módulo de geração da sintaxe concreta. Por fim, teceremos alguns comentários sobre todo o assunto tratado nesse capítulo.

4.1 UML Design Tester (UDT)

UDT é uma ferramenta capaz de gerar automaticamente testes de design para verificar os artefatos do diagrama de classes. Esses testes são gerados para todos os templates mostrados no capítulo anterior. A UDT foi implementada em Java, a sequência de atividades que essa ferramenta realiza para gerar dos testes de design estão ilustradas na Fig. 4.1.

As 4 atividades princiapais do UDT são realizadas por 4 módulos distintos:

- **Módulo de Pré-processamento**: responsável por executar todas as ações necessárias para a configuração do ambiente de geração dos testes de design. Por exemplo, verificar a consistência dos arquivos.

- **Módulo de Geração de Modelos**: executa as ações necessárias para geração dos modelos dos testes de design para o diagrama de classe a ser testado, seguindo os templates de testes de design.

- **Módulo de Geração da Sintaxe Concreta**: responsável por, baseado nos modelos dos testes de design, gerar o arquivo com a classe Java final, compilável e

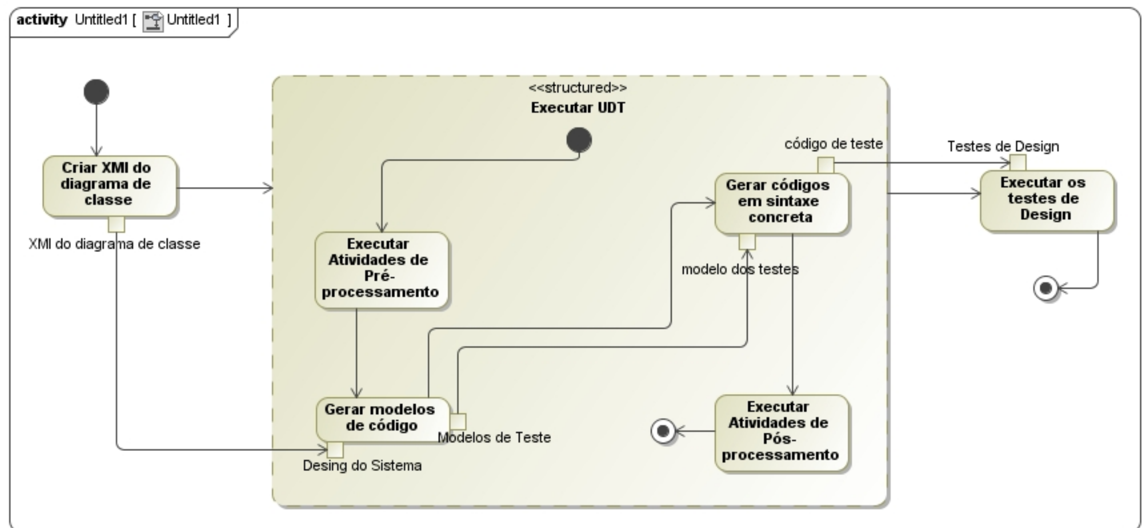

Figura 4.1: Principais atividades realizadas pela ferramenta UDT

executável;

- **Módulo de Pós-processamento**: finaliza corretamente a ferramenta UDT, excluindo todos os artefatos intermediários para a geração do teste de design final.

A abordagem adotada para a geração automática dos testes de design foi o padrão MDA. Há vários fatores que nos levaram a escolher MDA para dar suporte a nossa solução [9], como:

- **Alinhamento com os padrões da OMG**. A OMG propôs MDA de forma que fosse capaz de facilitar o processo de desenvolvimento de software, alinhado com seus próprios padrões;

- **Reuso de transformações**. Dado que todas as transformações são baseadas em meta-modelos, então domínios que compartilhem os mesmos meta-modelos podem reusar suas (uma parte ou completamente) transformações. Por exemplo, considere dois projetos que geram modelos de código Java usando o mesmo meta-modelo. Para gerar a sintaxe concreta desses modelos um projeto pode reusar as

transformações do outro e somente implementar as características que não são cobertas pelas transformações do primeiro projeto;

- **Separação evidente entre representação e lógica**. A representação do sistema, as transformações de cada nível e a implementação do sistema, são todos implementados por artefatos distintos;

- **Separação de *Concerns***. Para cada artefato tratado neste trabalho, existe um catálogo específico de transformações responsável por gerar seus testes de design.

4.1.1 Módulo de Pré-processamento

Nesse módulo estão todas as ações necessárias para a configuração do ambiente de geração dos testes de design. As ações *default* do sistema são:

1. **Verificação dos arquivos de entrada**: ela verifica se os parâmetros inseridos como entrada para a ferramenta realmente existem, se o arquivo XMI (padrão da OMG para persistir em arquivo diagramas UML) está bem formado, e se o software testado corresponde a um *jar*;

2. **Criação de um diretório temporário**: ela cria o diretório onde todos os arquivos auxiliares, criados durante a geração dos testes, serão criados (*sand box*);

3. **Ajuste de meta-modelo**: ela converte o meta-modelo do XMI passado como parâmetro para o meta-modelo adotado pela ferramenta, o proposto pela OMG. Isso se deve ao fato de que nossa ferramenta deve ser capaz de tratar XMIs gerados por diversas ferramentas. Contudo, algumas dessas ferramentas geram o XMI resultante com um meta-modelo diferente do padrão, sendo assim, necessitando ser alterado. Algumas ferramentas que persistem adequadamente os arquivos XMI são, por exemplo, MagicDraw [32] e o OMONDO [43];

4. **Configuração da localização do projeto testado**: ela insere o caminho do *jar* da aplicação testada nos arquivos de configuração. Essa informação é importante para posteriormente ser usada na geração dos testes de design com esse caminho especificado.

Outras ações podem ser incorporadas ao sistema. Para tanto, essas novas ações devem estender da interface *java.lang.Runnable* da API de Java [5], e elas devem executar suas tarefas no corpo do método *run*. Além disso, deve-se adicionar as novas ações à lista de ações a serem executadas (*preActions*) contida na classe *br.edu.gmf.udt.preProcess.Preprocessor*.

4.1.2 Módulo Gerador de Modelos

Esse módulo é responsável por criar todos os modelos dos códigos dos testes de design produzidos pela ferramenta UDT. Com o intuito de que nossa ferramenta seja capaz de executar outras transformações, além das apresentadas neste trabalho, adotamos a micro-arquitetura mostrada na Figura 4.2. Nesta arquitetura adotamos o *design pattern Abstract Factory* [17] que se deve prover um conjunto de interfaces capaz de abstrair a forma como diferentes família de produto sejam implementadas. Para a UDT, esse conjunto de interfaces abstraem a execução das transformações que geram os testes de design. Para tanto, definimos duas interfaces *M2M_Seed* e *M2M_Machine* [1] e uma fábrica (*M2M_Factory*) responsável por selecionar as realizações dessas interfaces. Os objetos que implementam a *M2M_Machine* são responsáveis por executar diretamente do código da ferramenta as transformações que geram os modelos de testes. Os objetos que implementam a *M2MSeed* possuem duas funções: (1) selecionar sua M2M_Machine correspondente no método *createMachine* da *M2M_Factory*; (2) passar os parâmetros

[1] M2M é uma abreviação para **Modelo para Modelo**. Esta é uma abreviação adotada na implementação para nos referirmos ao Módulo de Gerador de Modelos.

necessários para a execução de sua *M2M_Machine* correspondente, como por exemplo, o caminho do modelo de entrada e saída, etc.

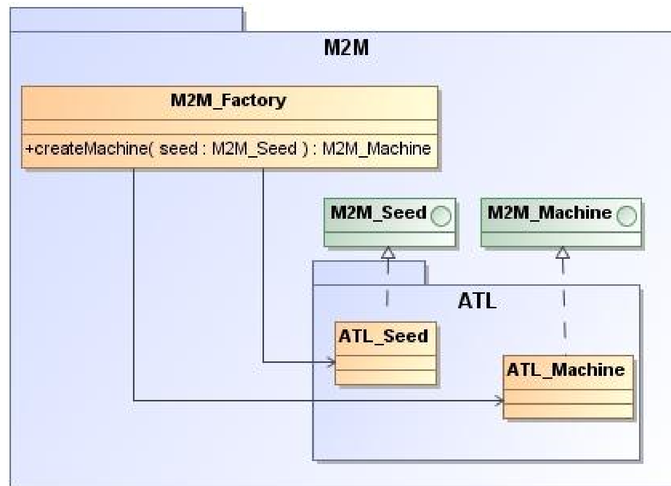

Figura 4.2: Arquitetura para extensão da ferramenta UDT

Módulo de Geração de Modelos de Testes de Design

Dado que esse módulo é responsável por gerar os modelos do testes de design a partir dos diagramas de classe UML e que adotamos o *framework* MDA para a geração desses testes, esse módulo é responsável por transformar modelos PIM em PSM. O diagrama de classe é considerado como PIM, visto que ele é genérico o suficiente para que possa ser implementado em qualquer linguagem de programação orientada a objeto. E os modelos de testes de design são PSM, pois são modelos específicos da linguagem Java.

A Fig. 4.3 mostra a abordagem MDA adotada para esse módulo. Nesta figura, encontramos os seguintes artefatos:

- **Meta-modelos**: como esse módulo gera modelos de código Java a partir de modelos UML, então adotamos dois meta-modelos: i) o origem sendo o da UML 2.0, provido pelo consórcio OMG; e ii) o destino como sendo o da linguagem

Java - JAS (Java Abstract Syntax) [24], uma representação completa do código Java, adotada pelo Eclipse, por exemplo;

- **Modelos**: os modelos de entrada são os diagramas de classe UML contendo o design estrutural do software, enquanto os de saída são os modelos do código dos testes de design, criados de acordo com os *templates* de teste mostrados no capítulo anterior. Ambos os modelos de entrada e saída devem estar em conformidade com seus respectivos meta-modelos;

- **Transformações**: as transformações foram feitas seguindo a linguagem ATL. Escolhemos essa linguagem, pois ela possui um *framework* de criação e execução amplamente utilizado, e está alinhada com os atuais padrões MDA.

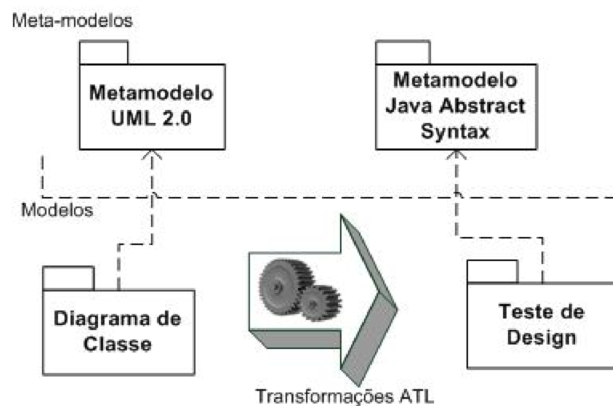

Figura 4.3: Abordagem MDA do Módulo de Geração de Modelos de Testes de Design

Visto que especificamos em ATL as transformações que geram os modelos dos testes de design, então implementamos ainda a *ATL_Seed* e *ATL_Machine* para a execução automática dessas transformações, de acordo com a arquitetura mostrada na seção anterior (Figura 4.2).

Transformações em ATL

Os testes de design são gerados seguindo os templates de testes de design. Contudo, esses *templates* só existem conceitualmente no UDT. Na ferramenta as transformações de PIM para PSM são a realização da aplicação dos *templates*, ou seja, são essas transformações que capturam as informações relevantes do diagrama de classe e geram os modelos do código de teste para cada artefato.

Para a realização das transformações nesse nível utilizamos a linguagem ATL. Apesar do padrão proposto pela OMG para esse tipo de transformação ser QVT. Decidimos usar ATL, pois, diferente de QVT, essa linguagem possui um *framework* de criação e execução amplamente utilizado. E ainda, assim como QVT, estas transformações estão completamente alinhadas com os atuais padrões da OMG.

As transformações para a geração dos testes de design foram organizadas hierarquicamente para facilitar a compreensão, identificação e extensão dessas transformações. Dividimos essa hierarquia de regras em 4 níveis. A Figura 4.4 mostra a aplicação de cada um desse níveis para gerar uma linha de código que verifica se uma classe é abstrata.

1. *R1 - Regras de artefato do código*. Desenvolvemos uma *macth rule* ATL [2] para cada artefato do diagrama de classe abordado neste trabalho. Essa *match rule* é responsável por criar o teste de design, segundo o *template* desse artefato. Essa regra cria o método que testa o artefato (método vazio) e chama as *called rules* ATL[3] que criam a verificação de cada característica desse artefato. A Figura 4.4 mostra que as transformações referentes à Classe são executadas sobre a *Classe C*;

2. *R2 - Regras de característica de artefato*. Essas *called rules* criam os trechos do teste de design que testam cada característica específica de um artefato. Essas regras, por sua vez, chamam a *called rule* responsável por criar cada linha do *template* que verificam essa característica. A Figura 4.4 mostra a transformação R2 gerando o trecho do testes de design referente a verificação da existência da *Classe C*. Apesar da Figura 4.4 mostrar que essa transformação gera uma linha, ela poder gerar várias outras, dependendo do código necessário para gerar o trecho do teste que verifica a característica;

3. *R3 - Regra de linha de código*. Essas regras criam as entidades específicas de uma linha de comando. Depois disso chama uma *called rule* que recebe essas especificidades e cria a linha de comando final. Considere como exemplo a declaração simples de uma variável. As regras desse nível vão criar o nome da variável e o seu tipo, e em seguida chama uma regra que monta a linha de código final. A Figura 4.4 mostra a transformação R3 gerando as entidades específicas para a criação da linha do testes de design que verificam a existência da *Classe C*;

[2]*Macth Rule* - é uma forma de transformação provida pela ATL onde os programadores podem declarar como os elementos dos modelos de entrada podem ser mapeados nos elementos dos modelos de saída. Trata-se de regras declarativas.

[3]*Called rule* - é um tipo de regra provido pela ATL que podem ser invocadas sem a necessidade haver mapeamento com algum elemento. Trata-se de regras imperativas.

4. *R4 - Regra de comando genérico de Java*. Essa regra recebe as especificidades da linha de comando e a cria de acordo com meta-modelo da linguagem Java. Por exemplo, considere a declaração simples de uma variável mostrada no item anterior, a regra em R4 vai receber como parâmetro o nome e tipo da declaração. Com esses parâmetros, ela vai criar uma entidade de um comando de uma declaração de variável, e incorpora essa nova linha aos comandos do devido método. A Figura 4.4 mostra a transformação R4 gerando a linha final do código e a acrescentando ao testes que está sendo gerado.

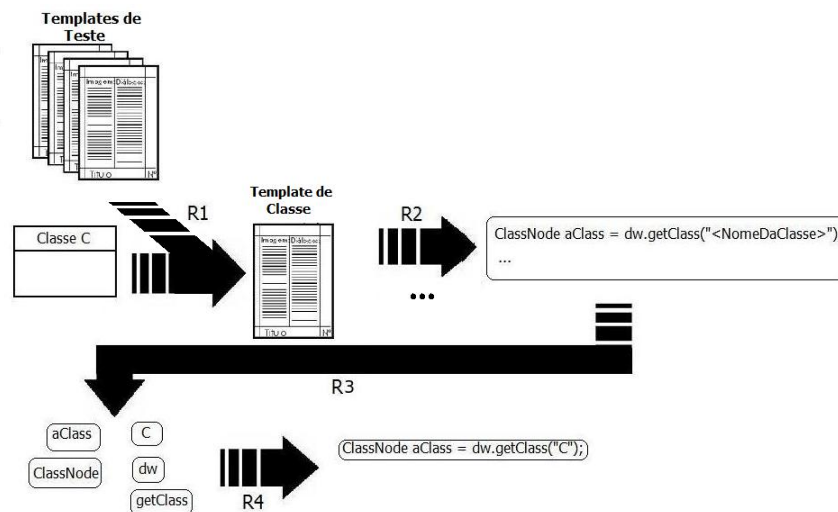

Figura 4.4: Hierarquia das regras de transformações ATL

```
1  module TestClassJava;
2  create OUT : JavaAbstractSyntax from IN : uml2;
3  ...
4  rule CreateClassTest{
5          from class: uml2!Class
6          to blockMethod: JavaAbstractSyntax!Block(
7                  statements <- Sequence{})
8          do{
9                  self.initializingClassTest(blockMethod, class);
10                 self.verifyIsAbstract(blockMethod, class);
11                 self.verifySuperType(blockMethod, class);
12                 self.verifyClassModifiers(blockMethod, class);
13                 self.verifyClassInterfaces(blockMethod, class);
14                 self.CreateGeneralTestMethod('testClass'+
15                         self.classesCounter.toString()+
16                         '_' + class.methodName, blockMethod);
17         }
18 }
19 ...
```

Código Fonte 4.1: Transformação ATL para geração dos modelos de Teste de Design para classe.

Ilustramos no código 4.1, a regra de transformação R1 para a geração dos testes para Classe. Contudo, especificamos todas as regras de todos os níveis que geram os testes de design mostrados no capítulo anterior. Essas transformações podem ser encontradas no site da ferramenta [50]. O código 4.1 está organizado da seguinte forma:

- Linha 1: identifica o módulo dessa transformação ATL, mostrando que ele trata da verificação de classes;
- Linha 2: identifica que o modelo de saída segue o meta-modelo da Sintaxe Abs-

trata de Java. E que o modelo de entrada deve seguir o meta-modelo de UML2;

- Linha 4: mostra o nome da regra responsável pela criação dos testes para a verificação das classes do template.
- Linha 5: mostra que essa *match rule* será executada para cada classe do diagrama de classe.
- Linhas 6 - 7: cria o elemento *Block* do meta-modelo de JAS. Esse elemento é o bloco do método, onde todos os comandos gerados (pelas regras R3) deverão ser incorporados;
- Linhas 8 - 17: é a parte imperativa dessa regra, é nesse trecho onde são chamadas as regras do nível R2;
- Linha 9: Essa linha chama uma *called rule* R2 responsável por criar a inicialização do método de teste (linha 3 do template de testes de design para classe, Código A.1);
- Linha 10: Essa linha também chama outra *called rule* R2, essa é responsável por verificar se a classe é abstrata (linhas 4 e 5 do template de testes para classe, Código A.1);
- Linha 11: já essa linha chama uma *called rule* R2 responsável por verificar o super tipo da classe. É nessa *called rule* onde é examinado se a classe verifica do diagrama possui algum super tipo (linhas 6, 7 e 8 do template para Classe, Código A.1). Se a classe testada não possuir super tipo, nenhuma outra *called rule* é chamada;
- Linha 12: essa linha chama uma *called rule* R2 responsável por verificar a visibilidade da classe (linhas 9, 10 e 11 do template de classe, Código A.1);

- Linha 13: essa linha chama uma *called rule* R2 responsável por verificar as interfaces realizadas pela classe (linhas 12 a 21 do template do teste de classe, Código A.1);

- Linhas 14 - 16: diferente das linhas anteriores, essa linha não chama uma *called rule* R2, ao invés disso, ela chama diretamente uma *called rule* R4. Ela é responsável por criar um método segundo o meta-modelo da Sintaxe Abstrata de Java. Essa *called rule* recebe dois parâmetros: o nome do método e o corpo do método com as suas linhas (*Block*). Para o nome do método é passada uma *String* formada por: 'testClass' (linha 14), um contador (para servir como <diff>, linha 15) e o nome completo da classe (linha 16). Para o corpo do método é passado o elemento *Block* (linha 16) criado no início da transformação (linhas 6-7), onde cada linha criada para o teste de design foi incorporada.

4.1.3 Módulo de Geração da Sintaxe Concreta

A fim de tornar os modelos de testes gerados pelo módulo da seção anterior compiláveis e executáveis, construímos um módulo para geração de sintaxe concreta. Implementamos esse módulo para modelos que sigam o meta-modelo JAS adotado.

Na Figura 4.5 é apresentada a abordagem MDA adotada para o Módulo Gerador de Sintaxe Concreta. Esse módulo é responsável pela transformação de PSM (modelos dos testes de design) para sintaxe concreta (código Java compilável e executável). Nesta figura, descrevemos os seguintes artefatos:

- **Meta-modelo**: dado que esse módulo gera a sintaxe concreta a partir de modelos de código Java, somente foi necessário adotar um meta-modelo, o JAS, como meta-modelo de origem. Um meta-modelo destino é desnecessário na geração de sintaxe concreta de uma linguagem final;

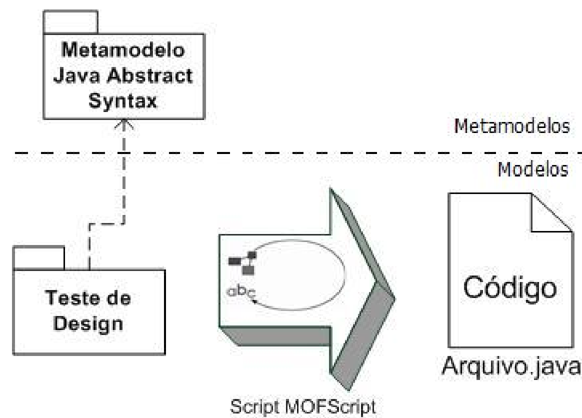

Figura 4.5: Abordagem MDA do Módulo de Geração da Sintaxe Concreta

- **Modelos**: os modelos de entrada são os modelos do código dos testes de design produzidos pelo módulo anterior. Os modelos de saída são os códigos dos testes de design na sintaxe concreta de Java;
- **Transformações**: As transformações de modelo para texto (M2T) foram definidas usando a linguagem MOFScript [35]. Essa linguagem mapeia os elementos do modelo do código em elementos da gramática de Java, formando sentenças bem formadas de acordo com essa gramática.

O processo de geração da sintaxe concreta se dá em três etapas: inicialmente, (i) são carregados os modelos de código Java; em seguida, (ii) os scripts de transformação realizam os mapeamentos dos elementos dos modelos do código para a sintaxe concreta da linguagem; e por fim, (iii) o código Java em sintaxe concreta é persistido em arquivo. Esse arquivo produzido é um arquivo de código fonte Java compilável, uma vez que ele está de acordo com o meta-modelo de Java.

Transformações em MofScript

Diferente das transformações mostradas na seção 4.2.1, responsáveis por gerar os modelos do código de teste, as transformações desse módulo possuem outro objetivo. A partir dos modelos de código, gerar código em sintaxe concreta, com expressões bem formadas de acordo com a gramática da linguagem Java.

A OMG propôs um padrão específico para tratar as transformações de Modelos para texto, dentre as linguagens que seguem esse padrão decidimos usar a MofScript. As vantagens dessa linguagem que nos motivou a sua escolha foram: i) ela possui um ambiente de criação e execução para suas transformações; ii) ela possui características de orientação a objetos, como herança e polimorfismo; iii) o ambiente de criação possui algumas vantagens (como *highlight* de sintaxe, auto-complemento de código e indicação de erros mais clara) em relação a outras abordagens como o JET [26], e o ATL DT [8]; e iv) o MofScript está em conformidade com os padrões propostos atualmente pela OMG. Apesar de MofScript não ser o padrão adotado pela OMG, ela está em conformidade com o *Request for Proposals* da OMG para o padrão MOF2Text [23]. A resposta adotada pela OMG para esse padrão está sendo desenvolvida pela ferramenta MDT [34] do eclipse, contudo ela ainda está em desenvolvimento, e a versão 1.0 só ficará disponível em setembro de 2009. Definimos uma abordagem para a criação das regras de transformações em MofScript. Inicialmente, determinamos que deveríamos agrupar as transformações referentes a cada tipo concreto da JAS de acordo com os tipos abstratos que elas realizam. Por exemplo, agrupamos em um único arquivo as transformações referentes aos tipos concretos que realizam *AbstractTypeDeclaration*, outro para os que realizam *Annotation*, e assim por diante. No meta-modelo de JAS podem ser encontrados todos os tipos definidos (tanto concretos quanto abstratos) [24].

Seguimos essa abordagem pelos seguintes motivos: (i)*facilita a compreensão*, pois as regras que tratam os tipos que realizam o mesmo tipo abstrato, normalmente, são sema e envolvem elementos similares; e (ii) *simplifica os relacionamentos entre os*

agrupamentos, pois um agrupamento somente precisa conhecer os outros os quais seus elementos estão relacionados.

Determinamos ainda que as regras de transformação para cada agrupamento fossem divididas em dois grupos. As ***TextTransformations*** **(TT)**, responsáveis por recuperar as informações relacionadas às especificidades do elemento do meta-modelo tratado pela regra. E as ***SyntaxTransformations*** **(ST)**, responsáveis por formatar o texto gerado selecionando as palavras reservadas da linguagem e escrevendo as informações na ordem correta. A título de ilustração, considere o comando de declaração de uma variável (*VariableDeclarationStatement*). Esse elemento no meta-modelo possui 3 relacionamentos:

- Modificadores: identificam quais as características da declaração desse atributo. Por exemplo, a visibilidade (public, private, etc.), se é estático (static), etc.
- Tipo: identifica qual o nome do tipo do atributo que está sendo declarado;
- Fragmentos: identificam as informações referentes à como o atributo está sendo declarado. Por exemplo, o nome do atributo, a forma como esse atributo está sendo inicializado, etc.

O elemento comando de declaração de uma variável (*VariableDeclarationStatement*), como o próprio nome sugere, pertence ao agrupamento de Comando (*Statement*). O código 4.2 exibe um trecho da transformação TT para Comandos responsável por recuperar as informações do modelo referentes ao comando de declaração de uma variável. E o código 4.3 exibe o trecho transformação ST para Comandos responsável por formatar comando de declaração de uma variável de acordo com a sintaxe da gramática de Java.

Para escrever a expressão referente à declaração de variável em sintaxe concreta tem-se que recuperar essas informações e escrevê-las de acordo com a gramática da

linguagem Java. No código 4.2 podemos observar o código que recupera essas informações do modelo.

```
1  import "ExpressionTT.m2t"
2  import "NameTT.m2t"
3  import "TypeTT.m2t"
4  import "ASTNodeTT.m2t"
5  import "VariableDeclarationTT.m2t"
6  import "StatementTemplates.m2t"
7  texttransformation StatementTT (in jas:"JavaAbstractSyntax") {
8    jas.Statement::getStatementCode() : String {
9          if(self.oclIsTypeOf(jast.VariableDeclarationStatement))
10                 result = self.getVariableDeclarationStatementCode()
11         ...
12         }
13         ...
14         jas.VariableDeclarationStatement
15                 ::getVariableDeclarationStatementCode():String {
16                 var modifiers : String = getModifiers(self.modifiers)
17                 var type : String = self.type.getTypeCode()
18                 var fragments : String =
                       getVariableDeclarationFragmentsCode(
19                             self.fragments)
20                 result = variableDeclarationStatementTemplate(
21                             modifiers, type, fragments)
22         }
23 }
24 ...
```

Código Fonte 4.2: Transformação para recuperação das informações sobre a declaração de Variável.

O código 4.2 está organizado da seguinte forma:

- Linhas 1-5. Importam a transformação para os elementos dos outros agrupamentos os quais os elementos que realizam a classe abstrata *Statement* estão relacionados;
- Linha 6. Importa as transformação que formata os comandos;
- Linha 7. Especifica o nome da transformação para o agrupamento *Statement*, *StatementTT*. Essa linha ainda especifica com qual meta-modelo essa transformação trabalha, no caso, o meta-modelo da sintaxe abstrata de Java, *JavaAbstractSyntax*;
- Linhas 8-12. Especificam o método *getStatementCode* que realiza a geração do código para este elemento. Visto que na linha 8 ele está declarado para o elemento *Statement* do meta-modelo, ele será executado para todo elemento que realize *Statement*. Esse método verifica qual o tipo do elemento que está realizando o *Statement* e chama o método apropriado. A linha 9 verifica se o tipo do elemento é um *VariableDeclarationStatement*, e se assim o for, o resultado da transformação desse *Statement* será o retorno do método *getVariableDeclarationStatementCode* na linha 10;
- Linhas 14 - 22. Especificam o método *getVariableDeclarationStatementCode* e do elemento *VariableDeclarationStatement*. Esse método é responsável por recuperar do modelo do código as informações referentes ao comando de declaração de variável. Essas informações são modificador, tipo e fragmentos, recuperadas nas linhas 16-19. E por fim, esse método chama a transformação ST que formatará o comando adequadamente;

```
1  texttransformation StatementST(in JAS:"JavaAbstractSyntax"){
2  ...
3   module::variableDeclarationStatementTemplate(
4                           modifiers: String,
5                           type: String,
6                           fragments : String){
7       result = modifiers + " " + type + " " + fragments + ";"
8   }
9  ...
```

Código Fonte 4.3: Transformação para a formatação da declaração de Variável.

O código 4.3 trata da organização e formatação do comando de declaração de variável. Este código está organizado da seguinte forma:

- Linha 1. Nomeia a transformação, *StatementST*. Além disso, essa linha ainda especifica com qual meta-modelo essa transformação trabalha, JAS;
- Linhas 3 - 9. Especificam o método, *variableDeclarationStatementTemplate*, que organiza e formata um comando de declaração de uma variável. Ele recebe como parâmetros as expressões resultantes dos outros elementos do meta-modelo, modificadores (linha 5), tipo (linha 6) e a visibilidade (linha 7). E, por fim, na linha 8, esses resultados são organizados de forma que formem uma expressão bem formada na linguagem Java.

Foram definidas transformações seguindo o esquema tratado anteriormente para Comando de Declaração de Variável. Todas essas transformações podem ser encontradas no site do UDT [50].

4.1.4 Módulo de Pós-Processamento

Nesse módulo estão todas as ações necessárias para a finalização correta da ferramenta. A ação *default* do sistema é excluindo todos os artefatos intermediários para a geração do teste de design final.

Outras ações podem ser incorporadas ao sistema. Para tanto, essas novas ações devem estender da interface *java.lang.Runnable* da API de Java [5], e elas devem executar suas tarefas no corpo do método *run*. Além disso, deve-se adicionar as novas ações à lista de ações a serem executadas (*posActions*) contida na classe *br.edu.gmf.udt.posProcess.Posprocessor*. Definimos que as ações deve estender a interface *java.lang.Runnable* para uniformizar a execução das ações[4].

4.2 Considerações Finais

Neste capítulo, mostramos a forma como implementamos a ferramenta UDT, que além de automatizar o processo de geração automática dos testes de design (remediando as limitações para a não adoção da nossa abordagem), ainda serviu como plataforma para a validação dessa abordagem.

Nossa ferramenta foi estruturada em quatro módulos: **Módulo de Pré-processamento**, responsável por executar todas as ações necessárias para a configuração do ambiente de geração dos testes de design; **Módulo de Geração de Modelos**, responsável por gerar os modelos dos testes de design para o diagrama de classe a ser testado; **Módulo de Geração da Sintaxe Concreta**, responsável por gerar o arquivo com a classe Java final, compilável e executável; **Módulo de Pós-processamento** responsável por finalizar a ferramenta UDT.

Projetamos nossa ferramenta de forma que ela necessite o mínimo de informação possível para a geração dos testes de design. Sendo assim, o modo de interação com a

[4] As ações são executadas sequencialmente.

ferramenta se dá por linha de comando. O comando de geração está descrito a seguir:

UDT«arquivoXMI» «diretorioDosTestes» «diretorioDoJarTestado»

Sendo: (1) *«arquivoXMI»* é o caminho absoluto para o arquivo XMI que descreve o diagrama de classes do design a ser testado; (2) *«diretorioDosTestes»* é o caminho no sistema de arquivos onde a ferramenta deve salvar os testes de design gerados; (3) *«diretorioDoJarTestado»* é o caminho absoluto para o jar do software que será testado pelos testes de design gerado.

Capítulo 5

Avaliação

O objetivo deste capítulo é apresentar a maneira como foram dirigidos os experimentos para a avaliação da abordagem proposta neste trabalho. Construímos a ferramenta UDT para que ela servisse não somente como forma de aplicação automática da nossa solução, mas também para que pudesse ser usada como plataforma de avaliação da nossa abordagem. Inicialmente, como prova de conceito da nossa abordagem, testamos nosso sistema com estudos de caso simples, biblioteca [42] e outros sistemas [13] (cenários esses com menos de 20 classes). Os resultados alcançados com esses sistemas foram satisfatórios, verificamos que 97% dos artefatos desses sistemas estavam implementados corretamente. Contudo, não poderíamos considerar que esses cenários serviriam como uma avaliação completa da abordagem. Sendo assim, definimos como avaliação da nossa técnica o uso de um estudo de caso que verifique a conformidade entre um sistema real e o seu diagrama de classe gerado através de engenharia reversa. Avaliamos os resultados desse experimento sobre as características: precisão e desempenho.

5.1 Metodologia dos Experimentos

O primeiro passo de qualquer experimentação é identificar os seus objetivos. Dessa forma, utilizamos a metodologia GQM (Goal/Question/Metric) [51] para definir as características que serão mensuradas neste trabalho:

Cenário 1

- *Goal*: Avaliar se a nossa abordagem é efetiva.
- *Question*: Nossa solução é capaz de verificar satisfatoriamente todas as características dos diagramas de classe UML abordadas nos capítulos anteriores?
- *Metric*: **Precisão**. No campo da ciência, engenharia, indústria e estatística, precisão é o grau semelhança entre o valor experimentado e o real [29]. No contexto da avaliação da nossa abordagem, calculamos a precisão através da fórmula mostrada na Figura 5.1. Nessa figura, o módulo do conjunto *Esperadas* é o número de artefatos que a engenharia reversa gerou corretamente. E o módulo da interseção entre os conjuntos *Achadas* e *Esperadas* é o número de artefatos do diagrama de classe gerado que os testes de design consideram como implementados corretamente no código.

$$\frac{|Achados \cap Esperados|}{|Esperados|}$$

Figura 5.1: Fórmula para Precisão

Cenário 2

- *Goal*: Avaliar o impacto em termos de tempo da implementação na nossa ferramenta.
- *Question*: Quanto tempo é gasto para a geração e execução dos testes de design?

- *Metric*: Desempenho. Consideramos como desempenho, a medição do tempo gasto pelo nosso sistema para gerar os testes de design e executá-los.

A precisão é a principal característica a ser mensurada e avaliada nessa avaliação. Ela mostra quanto nossa abordagem consegue verificar da conformidade entre o design e um código implementado corretamente. Com relação ao desempenho , ele também são características importantes, mas elas estão relacionadas com a avaliação da implementação da ferramenta.

5.1.1 Cenários

Todos os experimentos foram executados na mesma unidade de trabalho: Laptop HP Compaq 6910p, com um processador Intel Centrino de 2,1 Gz, 2 Gb de memória ram e o sistema operacional Windows Vista Business. Com relação à precisão, as configurações da estação de trabalho utilizada não impactam nos resultados.

Definimos que um cenário para servir como estudo de caso dessa avaliação deve ser um projeto de software que possua o diagrama de classe da aplicação e o código fonte implementado em Java. Apesar do uso de diagrama de classes para definir o design de sistema ser uma prática bastante usual, não conseguimos identificar nenhum projeto com mais de 100 classes que disponibilizasse tanto o diagrama de classes da aplicação, quanto o código fonte da mesma. As empresas que contatamos que adotam esse tipo de abordagem para desenvolvimento de sistemas mantêm o código e o design fechados. Esses projetos são normalmente sistemas de informação de grandes empresas ou sistemas desenvolvidos por fábricas de software.

Em contrapartida, boas fontes de códigos em Java de sistemas complexos (com mais de 1000 classes) são os repositórios de sistemas *open source*. Contudo, esse tipo de sistema não utiliza os diagramas UML como forma de expressar o seu design. Sendo assim, a solução encontrada para avaliarmos nossa proposta foi adotar um sistema *open*

source com um design reconhecidamente bom, e aplicar sobre ele engenharia reversa para gerar o seu diagrama de classes. Depois disso, usar a UDT para verificar se o diagrama gerado está em conformidade com o código original. E por fim, analisar os resultados obtidos com o uso da UDT.

Dessa forma, o primeiro passo foi a adoção de uma ferramenta para realizar a engenharia reversa. Para tanto, traçamos os seguintes requisitos para a ferramenta de engenharia de software a ser escolhida:

- **Requisito 1**: *geração o diagrama de classes a partir do código fonte em Java*;
- **Requisito 2**: *exportação pra o XMI compatível com o proposto pela OMG*;
- **Requisito 3**: *ferramenta sem necessidade de compra da licença*;
- **Requisito 4**: *capacidade de tratar sistemas complexos*.

Na Tabela 5.1 podemos ver todas as ferramentas candidatas em ordem alfabética, e se cumprem ou não os requisitos necessários.

Tabela 5.1: Ferramentas para Engenharia Reversa

	Requisito 1	Requisito 2	Requisito 3	Requisito 4
ArgoUML [6]	sim	não	sim	não
Altova UModel [2]	sim	não	não	sim
Borland Together [10]	sim	sim	não	sim
Gentleware Poseidon [46]	sim	sim	não	sim
Magic Draw [32]	sim	sim	sim	sim
OMONDO [43]	sim	sim	sim	sim
Visual Paradigm [52]	sim	sim	não	sim

Um dado que não está na tabela 5.1 é que o UModel, o Together e o Visual Paradigm também possuem uma versão sem licença (*trial version*). Contudo, nessas versões o

Together e o Visual Paradigm não realizam a engenharia reversa a partir de código Java. E a versão trial do UModel não exporta para uma versão do XMI compatível.

De acordo com a tabela 5.1 as opções para ferramentas para engenharia reversa são o Magic Draw e o OMONDO, pois contemplam todos os requisitos levantados. Todavia, o OMONDO, apesar de gerar o diagrama de classes através de engenharia reversa para sistemas grandes, precisa gerar vários XMI, um para cada pacote raiz, dificultando assim a sua adoção. Logo, a ferramenta escolhida e adotada foi o Magic Draw.

O passo seguinte foi a adoção dos projetos para servirem como estudo de caso. Inicialmente, escolhemos um projeto que servisse como estudo de caso para mensurarmos e avaliarmos detalhadamente a precisão da nossa abordagem. Definimos os seguintes requisitos para a escolha de um projeto: (1) possuir mais de 1000 classes; (2) ser *open source*; e (3) possuir um design de código reconhecido. O projeto escolhido como estudo de caso foi o FindBugs [19], uma ferramenta que usa análise estática para descobrir *bugs* em códigos Java. Ela analisa os bytecodes de uma aplicação Java e gera um relatório descrevendo os prováveis *bugs* encontrados na análise. Essa ferramenta possui todas as características apontadas anteriormente: é *open source*, possui 1426 classes, e tem um design reconhecido [38].

Para avaliarmos o desempenho, definimos um catálogo de projetos com o mesmos requisitos mostrados no parágrafo anterior, com a diferença que esses projetos devem possuir diferentes tamanhos (quantidades de classes). Os projetos escolhidos para o catálogo foram: Findbugs, JUnit [30], Apache Ant [25] e o DesignWizard. O JUnit é um framework para execução automática de testes sobre programas em Java e o Apache Ant é uma ferramenta para execução automática de tarefas independente de plataformas.

Definimos um projeto para a avaliação da precisão em detrimento aos quatro para a avaliação do desempenho, pois o tratamento dos dados produzidos para a avaliação da precisão é bem mais trabalhosa, e o volume de dados produzido também é maior.

Um material importante para a avaliação da precisão da nossa abordagem foi o uso

de um SGBD (Sistema de Gerenciamento de Banco de Dados), pois, para verificar a precisão da nossa abordagem, foi necessário monitorar todos os testes gerados para saber os resultados de cada um. O volume de dados gerados com essa monitoração foi muito alto, exigindo assim uma forma sofisticada para manipulá-los, e assim extrairmos algumas conclusões sobre eles através da mineração desses dados coletados. Usamos o banco de dados *MySql* [36] como forma de armazenamento dos dados, e a mineração foi realizada através de consultas *SQL* [36].

5.1.2 Métodos

Adotamos dois métodos para a execução dos experimentos: um referente à avaliação da precisão da nossa abordagem, outro referente à avaliação do desempenho. Estes métodos são tratados a seguir.

Método para a Avaliação da Precisão

O método adotado para a avaliação da precisão da nossa abordagem segue uma sequência de passos bem definidos:

1. XMI do FindBugs. A partir do código fonte disponibilizado no repositório do *Findbugs*, geramos o seu diagrama de classe através da ferramenta de engenharia reversa do MagicDraw. Depois disso, usando ainda o MagicDraw, exportamos esse diagrama para o formato XMI;

2. Testes de Design. Usamos o XMI do diagrama de classe do FindBugs como entrada para o UDT e geramos os testes de design para verificar se o código estava em conformidade com o diagrama;

3. Instrumentação do Código. Nessa fase usamos a ferramenta de aspectos em Java, *AspectJ*, para instrumentarmos o código, a fim de coletarmos automaticamente os dados referentes aos resultados dos testes e salvá-lo no banco de dados;

4. Execução dos testes. Depois que os testes de design foram gerados e instrumentados, eles foram executados, e as informações sobre a execução dos testes foram inseridas no SGBD. Ao fim dessa fase, o banco de dados conterá todas as informações relevantes sobre a execução de todos os testes;

5. Depuração dos Dados. Nessa fase, foram executadas algumas consultas SQL sobre os dados crus obtidos. Nestas consultas são contabilizados os testes que passaram ou não. E sobre os que não passam, são identificadas as causas da falha dos testes.

Os dados desses resultados serão mostrados em tabelas que resumem os dados relevantes sobre a execução dos teste[1]. As tabelas que analisaremos aqui possuem os seguintes dados:

- **Entidades no diagrama**: o número de entidades criadas no diagrama de classes através da engenharia reversa. Esse dado foi fornecido pela própria ferramenta de engenharia reversa adotada;
- **Entidades no código**: o número de entidades presentes no código fonte do *FindBugs*. Esses fatos foram extraídos do código através do ASM extractor [7];
- **Testes gerados corretamente**: o número de testes que verificam entidades que foram geradas corretamente pela engenharia reversa. A engenharia reversa gera erroneamente muitas entidades que não estão presentes no código. Os aspectos instrumentam o código de forma que seja verificado a existência da entidade antes que os dados referentes à sua execução sejam mandados para o banco de dados. A existência da entidade é verificada usando o DesignWizard;

[1] As tabelas completas com todos os dados da execução dos testes podem ser acessados no site do UDT: *http://www.designwizard.org/index.php/UDT/*.

- **Testes que passam**: o número de testes de design que são executados e não apontam diferença entre o design e a implementação;

- **Precisão**: mensuramos a precisão como sendo a relação entre o número de os testes que passam e o número de testes gerados corretamente;

- **Testes que falham**: o número de testes que não passam, seja pelo fato de quem alguma verificação falhou ou por lançar alguma exceção durante a sua execução.

É importante ressaltar que, idealmente, o número de entidades no diagrama deve ser igual ao número de testes gerados corretamente, que por sua vez devem ser iguais ao número de entidades no código. A diferença entre esses dados pode apontar a imprecisão da engenharia reversa e/ou da geração dos testes.

Método para a Avaliação do Desempenho

O método adotado para a avaliação do desempenho da nossa abordagem segue uma sequência de passos bem definidos. Os dois primeiros passos são iguais aos mostrados no método anterior, o restante dos passos estão tratados a seguir:

1. Log do Código. Alteramos o código do UDT para ele ser capaz de mensurar o tempo gasto na geração do código de teste e salvar esse dado no log do sistema. Nessa fase foram coletados os dados referentes ao tempo de geração dos testes de design;

2. Execução dos Testes. Os testes foram executados e o tempo de execução foi mensurado usando o próprio *framework* de execução dos testes, o JUnit. Nessa fase foram coletados os dados referentes ao tempo de execução dos testes de design.

5.2 Resultados e Avaliação

Nesta seção iremos mostrar os resultados obtidos com a execução dos experimentos da forma como foi tratado na seção anterior. Os resultados serão mostrados e avaliados inicialmente para a precisão, e em seguida para o desempenho.

5.2.1 Precisão

Resultados para os Testes de Design para Classe

A tabela 5.2 mostra os dados relativos à execução do experimento de avaliação referente ao artefato Classe do diagrama de classe.

Tabela 5.2: Dados sobre os Testes de Design para Classe

Dado	Resultado
Entidades no Diagrama	2935
Entidades no código	1575
Testes gerados corretamente	1575
Testes que passam	1268
Precisão	80,5%
Testes que falham	305

Analisando a primeira e segunda linhas da tabela 5.2 podemos perceber que a engenharia reversa produziu 1360 classes a mais que as existentes no código, **classes-lixo**. Isso é decorrente de dois problemas na engenharia reversa. O primeiro é referente ao uso do *generics*[2] de Java no código, pois sempre que a ferramenta encontra uma sentença do tipo *Collection<E>*, a ferramenta cria uma classe chamada *E* na raiz do diagrama.

[2] É uma funcionalidade provida pela linguagem Java que permite especificar tipos genéricos, dentro de uma classe. Esses tipos depois podem ser especificados durante o uso da classe.

O segundo é referente aos tipos de Java e dos pacotes que são usados. Sempre que a ferramenta encontra um tipo diferente dos tipos primitivos de Java, ela cria esse tipo no pacote onde ele está sendo usado. Por exemplo, se a classe do sistema *p1.p2.C1* usa uma coleção *List*, a ferramenta vai criar a classe *p1.p2.List*. Sendo assim, para avaliar mais corretamente nossa abordagem, excluímos da nossa análise os testes gerados para as 1360 classes-lixo geradas pela engenharia reversa.

Ainda, analisando as linhas 4 e 5 da tabelas 5.2, podemos concluir que mesmo gerando testes para todas as classes do diagrama de classes (tanto as geradas corretamente quanto as erroneamente), o número de testes gerados corretamente é igual ao número de classes no código, isso indica que muito provavelmente todas as classes presentes no código foram verificadas.

Relacionando o número de testes que passam com o número de testes referentes às classes no código, temos uma precisão de 80,5%. Logo, constatamos que 1268 geradas pela engenharia reversa estão condizentes com a implementação. Contudo, 307 testes não passam, isso se deve especialmente a dois motivos:

- **Erro da engenharia reversa**: analisando os dados do banco de dados percebemos que esses erros foram decorrentes de falha da engenharia reversa. Desses testes, 11 são classes que realmente existem no código, porém herdam de classes provenientes de *jars* importados no código. E, 90 deles são provenientes de classes que herdam de tipos de Java. Durante a engenharia reversa os nomes dessas entidades que são herdadas é gerado errado, ocasionado em erro quando o Design Wizard tenta recuperá-la.

- **Provenientes de *bugs* do Design Wizard**: dentre os testes que falham, 72 deles são provenientes de um bug na representação de classes abstratas, pois, algumas classes abstratas são recuperadas do código, e não são representadas como abstratas. E o restante dos 132 erros são proveniente de um bug na representação da visibilidade das classes recuperadas. Algumas classes quando são recuperadas do

código são representadas como classes de visibilidade protegida (*protected*), ao invés de pública (*public*) como foi implementada no código. Todos esses *bugs* encontrados foram reportados para os desenvolvedores da ferramenta.

Resultados para os Testes de Design para Atributo

A tabela 5.3 mostra alguns dados referentes à execução do experimento de avaliação referente a Atributo.

Tabela 5.3: Dados sobre os Testes de Design para Atributo

Dado	Resultado
Entidades no Diagrama	1662
Entidades no código	7607
Testes gerados corretamente	1560
Testes que passam	1518
Precisão	97,3%
Testes que falham	42

Analisando a primeira e segunda linhas da tabela 5.3, podemos perceber que a engenharia reversa gerou muito menos atributos que os presentes no código. Isso se deve ao fato de que muitos atributos de algumas classes, que tem como tipo do atributo outra classe do mesmo sistema, foram mapeados (na engenharia reversa) como associações diretas (*Directed Associations*) entre essas classes.

Outro dado relevante é a diferença entre as entidades no diagrama e os testes gerados corretamente. A explicação para esse fato é que mesmo gerando um número inferior de atributos, a engenharia reversa ainda gerou alguns atributos inexistentes no código fonte, atributo-lixo. Contudo, se fizermos uma comparação entre os testes que realmente refletem entidades no código (testes gerados corretamente) e o número de testes que

passam, podemos perceber que obtivemos uma alta precisão, 97,3%.

Do testes gerados corretamente, 42 resultaram em falhas, como podemos perceber na linha 6 da tabela 5.3. Dessas, onze são provenientes de um bug na representação da visibilidade dos atributos, da mesma forma como foi apresentado para classe. E o restante das falhas são provenientes de uma limitação da nossa verificação. Essa limitação se deve ao fato que nossa implementação considera que atributos com multiplicidade maior que 1 deve ser mapeada como uma coleção de Java. Contudo, atributos com essa multiplicidade podem ser implementados também como um *array*.

Resultados para os Testes de Design para Método

A tabela 5.4 mostra os dados referentes à execução do experimento de avaliação referente a Método.

Tabela 5.4: Dados sobre os Testes de Design para Método.

Dado	Resultado
Entidades no Diagrama	8204
Entidades no código	4423
Testes gerados corretamente	4423
Testes que passam	3990
Precisão	90,21%
Testes que falham	433

Analisando a primeira e segunda linhas da tabela 5.4 podemos perceber que a engenharia reversa gerou mais métodos que os presentes no código, gerou 3781 métodos a mais, ou seja, método-lixo. Esse comportamento se deve a dois fatos: (i) 644 desses métodos realmente não existem no código e são provenientes de *bugs* na engenharia reversa, e (ii) 3137 são correspondentes a métodos duplicados gerados na engenharia

reversa. Relacionando o número de testes que passam com os testes que realmente refletem entidades no código (testes gerados corretamente), podemos perceber que obtivermos uma alta precisão, 90,21%.

Contudo, ainda foram identificadas 433 falhas na execução dos testes gerados corretamente. Dessas, 81 são *bugs* na engenharia reversa que criam métodos abstratos que no código são concretos. O restante dos erros é decorrente de erros na representação do retorno do método na engenharia reversa. Os tipos de retorno dos métodos quando são tipos de Java (como coleções, *threads*, etc.) são criados com nomes errados. Por exemplo, se um método retorna um *Set*, o nome do retorno atribuído pela engenharia reversa vai ser *Default.Set*, ao invés de *java.util.Set*.

Resultados para os Testes de Design para Pacote

A tabela 5.5 mostra os dados referentes à execução do experimento de avaliação referente a pacote.

Tabela 5.5: Dados sobre os Testes de Design para Pacote.

Dado	Resultado
Entidades no Diagrama	103
Entidades no código	49
Testes gerados corretamente	49
Testes que passam	49
Precisão	100,00%
Testes que falham	0

Analisando a primeira e segunda linhas da tabela 5.5 podemos perceber que a engenharia reversa gerou mais pacotes que os presentes no código. Contudo, 49 desses pacotes realmente refletiam pacotes presentes no código, o restante foi considerado como

pacote-lixo. Esses pacotes a mais são provenientes de uma falha na engenharia reversa que gerou, além dos pacotes do sistema, pacotes que estão presentes nas bibliotecas usadas pelo *Findbugs*.

Podemos perceber que todos os pacotes existentes no código são verificados corretamente. Contudo, a engenharia reversa não consegue reconhecer os relacionamentos entre os pacotes. Sendo assim, essas características não foram consideradas neste experimento.

Resultados para os Testes de Design para Interface

A tabela 5.6 mostra os dados referentes à execução do experimento de avaliação referente a Interface.

Tabela 5.6: Dados sobre os Testes de Design para Interface.

Dado	Resultado
Entidades no Diagrama	163
Entidades no código	645
Testes gerados corretamente	171
Testes que passam	163
Precisão	100%
Testes que falham	0

Analisando a primeira e segunda linhas da tabela 5.6 podemos perceber que a engenharia reversa gerou mais interfaces que os presentes no código (482 interfaces a mais), ou seja, interface-lixo. Contudo, na linha 3 podemos perceber que 171 dos testes gerados realmente testam interfaces presentes no código, o que é uma disparidade comparando com os 163 interfaces presentes no código. Esse comportamento se deve ao fato de que dos 171 testes gerados corretamente, 8 são correspondentes à interfaces

duplicadas geradas na engenharia reversa. Relacionando o número de testes que passam com os testes que realmente refletem entidades no código (testes gerados corretamente), podemos perceber que obtivermos a precisão de 100,00%.

5.2.2 Desempenho

Alguns requisitos importantes que nossa abordagem deveria cobrir para obter uma melhor aceitação da solução proposta são: Desempenho e Escalabilidade. Sendo assim, fizemos alguns experimentos sobre nossa ferramenta usando projetos de diferentes tamanhos como estudos de caso: FindBugs, JUnit, Ant e o DesignWizard. Os dados coletados com esse experimento podem ser vistos na tabela 5.7.

Tabela 5.7: Tempo de geração e execução dos testes de design para os projetos selecionados.

Projeto	Total de Entidades[1]	Tempo de Geração	Tempo de Execução
Findbugs	12079	18min	2min 24,094seg
Apache Ant	8307	8min 32seg	1min 2,723seg
DesignWizard	2231	1min 53seg	45,007seg
JUnit	1002	1min 34seg	31,011seg

Considerando os dados da tabela 5.7, podemos concluir que o sistema obteve um desempenho satisfatório com relação à avaliação dos sistemas tratados neste trabalho. Analisando os resultados quanto ao tempo de geração, podemos perceber que esse parâmetro tende a crescer polinomialmente com relação ao tamanho do sistema. Ressaltamos que mesmo o maior tempo de geração dos testes de design, 18 min para o *Findbug*, não representa um impacto no tempo total do processo de desenvolvimento, pois os testes podem ser gerados logo após a criação do diagrama de classe, enquanto o sistema

[1]Considere Entidades como sendo todos os artefatos tratados neste trabalho.

Tabela 5.8: Tempo de geração e execução para cada tipo de artefato no projeto Findbugs.

Artefato	Número de Testes Gerados	Tempo de Geração	Tempo de Execução
Classe	1575	4min 36seg	13.163seg
Atributo	1560	2min 22seg	12,112seg
Método	7560	7min 12seg	1min 34,043seg
Interface	161	2min 01seg	12,560seg
Pacote	49	1min 47seg	12,216seg

ainda está sendo desenvolvido.

A tabela 5.8 mostra o tempo gasto na geração e execução dos testes de design referentes a cada tipo de artefato tratado por nossa abordagem para o projeto *Findbugs*. Analisando essa tabela, podemos perceber que em média os testes de métodos foram os que mais gastaram tempo sendo gerados (7min 12 seg) e executados (1min 34,04 seg). Isso se deve ao fato de existirem mais dessas entidades com relação as outras, e sendo assim, foram gerados um número maior de testes. Outra observação que podemos fazer é o fato que apesar de que as entidades Classe e Método possuírem um número de total entidades muito próximo, Classe gastou mais 2min 14seg a mais para gerar seus testes de design, isso se deve ao fato de que a engenharia reversa gerou um número de entidades-lixo muito maior para Classe que para Método.

No tocante à escalabilidade, a abordagem mostrou-se capaz de tratar sistemas com diferentes tamanhos desde sistemas pequenos com menos de 100 entidades (as provas de conceito mostradas na introdução desse capítulo) e até sistemas com mais de 12.000 entidades. E analisando a solução proposta neste trabalho, a sua limitação sobre escalabilidade está relacionada com a infra-estrutura usada na implementação do UDT. Mais especificamente, o tamanho dos sistemas que nossa ferramenta pode tratar é limitado pela: capacidade do ATL-DT em gerir todo o XMI do sistema a ser testado, e

capacidade do DesignWizard em abstrair e tratar todo o código do sistema a ser testado.

5.3 Considerações Finais

Neste capítulo apresentamos a forma de avaliação que foi utilizada para avaliar nossa abordagem. Sucintamente, ela consiste da metodologia: uso de engenharia reversa para gerar o diagrama de classes de um sistema; verificação do diagrama resultando contra o código do próprio sistema.

Observamos com nosso experimento que a engenharia reversa da ferramenta MagicDraw é capaz de gerar o digrama de classes de um sistema com mais de 1000 classes. Contudo, gera muitas entidades que não estão presentes no código.

Analisando o comportamento da nossa abordagem sobre as entidades que foram mapeadas corretamente, pudemos perceber que ela alcançou um desempenho satisfatório. A maioria das falhas encontradas diz respeito as falhas na engenharia reversa da ferramenta e a *bugs* na infra-estrutura dos testes de design, o Design Wizard. E dentre as limitações na nossa abordagem, a que mais provocou falso-positivos foi o não tratamento de arrays como possível implementação de relacionamentos de multiplicidade maior que 1.

Sobre desempenho e escalabilidade, a UDT mostrou-se capaz de gerir sistemas grandes com eficiência. E apontamos que o gargalo nesse sentido com relação à abordagem está relacionado com a infra-estrutura usada na implementação, o ATL-DT e o Design Wizard.

E sobre a avaliação de toda a abordagem, as grandes limitações foram com relação a associações bidirecionais que não foram verificadas no experimento. Isso se deve ao fato de que a engenharia reversa não gerou nenhuma associação desse tipo. Outra limitação é o fato que os relacionamentos entre pacotes também não foram cobertos nessa abordagem. Neste caso também, a engenharia reversa não foi capaz de reconhecer

esse tipo de relacionamento entre os pacotes. Sendo assim, essas duas características só foram verificadas nos exemplos dos sistemas de informação da avaliação preliminar.

Capítulo 6

Trabalhos Relacionados

O objetivo deste capítulo é discorrer sobre trabalhos semelhantes ao nosso que propõem técnicas de verificação de design. Além disso, compararemos esses trabalhos com nosso trabalho mostrado nessa dissertação, mostrando as limitações de cada abordagem. Trataremos ainda de alternativas para a infra-estrutura adotada para o UDT, mostrando alternativas para a verificação de design.

6.1 Verificação de Design

Uma das técnicas mais comuns de verificação de design é o Design Peer Review [1]. Essa técnica sugere um processo de revisão de design amplamente usado em empresas. Ele consiste de um mecanismo que assegura a manutenção do design separando a equipe de desenvolvimento do sistema em grupos e as fazendo analisar umas o código desenvolvido pelas outras, dessa forma, além de uniformizar a codificação, mantêm o design uniforme e entendido por todos. A limitação dessa abordagem é que o processo sugerido é manual e, assim, pode levar a erros durante a análise. Além do fato que é um processo que pode demorar demais pra ser executado, visto que para projetos muitos grandes com dezenas ou centenas de classes esse processo por durar um tempo muito

longo.

Outra abordagem para verificação de design é proposta por Dinh-Trong et al. [49]. Neste trabalho o autor recupera automaticamente do diagrama de classe e sequência do UML 1.4, sequências de chamadas de métodos. Depois disso, ele monitora a execução do sistema comparando com o comportamento esperado dos diagramas. Dessa forma, se na execução de um método ele chamar algum outro método não modelado nos diagramas UML essa falha será reportada, mostrando um comportamento diferente do esperado. Apesar desse trabalho também tratar com diagramas UML, o foco dele é voltado para a verificação de comportamentos do sistema. Em detrimento a sua estrutura, que é o foco do trabalho apresentado nesta dissertação.

Um trabalho sobre verificação de design em código é o projeto PTIDEJ (*Pattern Trace Identication, Detection, and Enhancement in Java*) [37], que visa desenvolver uma ferramenta capaz de avaliar e melhorar a qualidade de programas orientados a objetos. Essa ferramenta cria um modelo do código a partir de análises estáticas e dinâmicas do programa. E, analisando o modelo gerado a ferramenta é capaz de detectar padrões de design e defeitos de design. Uma limitação dessa ferramenta é o fato de que os padrões de design que podem ser identificados são fixos da ferramenta. Em contrapartida, uma alternativa para a adoção dessa ferramenta seria gerar automaticamente algumas regras para identificar defeitos de design provenientes de diagramas de classe. Contudo, a ferramenta possui uma gramática BNF própria e limitada para descrever os defeitos de design. Já o UDT usa templates de testes que são facilmente customizáveis, e adota padrões que estão completamente alinhados com os da OMG. Em um relatório técnico [58] a equipe de desenvolvimento do PTIDEJ propõe uma série de regras usando a gramática da ferramenta para identificar associações entre as classes presentes no código. Neste trabalho, eles executam as regras sobre um código de teste, em seguida, esse código é passado para especialistas identificarem as associações, e por fim verificam se a ferramenta identificou todas as associações encontradas pelos especialistas.

No tocante a associações, nossa ferramenta diferencia-se pelo fato de que o UDT não tem como objetivo identificar associações, tendo em vista que todas elas já estão todas identificadas no diagrama. O UDT verifica se elas estão presentes de forma adequada no código fonte.

Outro trabalho sobre verificação de design em código é o FindBugs [19]. Esta ferramenta usa análise estática para descobrir bugs em códigos Java. Ela analisa os bytecodes de uma aplicação Java e gera um relatório descrevendo os prováveis bugs encontrados na análise. Um ferramenta similar é o LClint [12] que tem sido usada para verificar códigos escritos na linguagem C. Esta ferramenta foca-se na detecção de problemas baixo-nível em códigos fonte tais como referências à apontadores nulos ou código não usado. Ambas as ferramentas se concentram em detectar design em um nível de abstração muito baixo, e desconsideram completamente informações de design de mais alto nível, como em UML.

O PDL (*Program Description Logic*) [11] é um ferramenta que verifica de design estático sobre código Java. Nesse trabalho os autores descrevem uma abordagem capaz de verificar regras de design através do uso de aspectos. Para tanto, eles propõem uma linguagem, baseada na linguagem usada para descrever os pointcuts de aspectos, responsável por descrever as regras de design a serem verificadas. Essa abordagem é capaz de verificar todas as características que verificamos nos testes de design usando o DesignWizard. Contudo, esses testes devem ser descritos na linguagem própria da implementação do PDL, ao invés de um teste comum em Java, como é feito usando o Design Wizard. Consideramos que na forma de código Java os testes seriam mais fáceis de serem entendidos e estendidos. E ainda, ele não trata de forma alguma design expresso em diagramas de classe UML.

	Testa Design Estrutural	Trata UML	Código Java	Completamente Automática
Nossa abordagem	sim	sim	sim	sim
Design Peer Review	sim	não[1]	sim	não
Dinh-Trong	não	sim	sim	não
PTIDEJ	sim	não	sim	sim
PDL	sim	não	sim	sim

Tabela 6.1: Comparação entre trabalhos relacionados.

6.2 Considerações

Neste capítulo, apresentamos alguns trabalhos relacionados com a nossa abordagem. Esses trabalhos estão relacionados com o nosso em várias características: verificação de diagramas UML em código, geração automática de código a partir da documentação, verificação do design estrutural no código, verificação de regras de design em código, etc. *Dentre esses trabalhos, o nosso se destaca por ser o único que garante, através de testes gerados automaticamente, a correta implementação do código segundo um design expresso através de diagrama de classes UML.* Resumimos na tabela 6.1 a comparação entre os trabalhos relacionados nesta seção e a nossa abordagem.

[1]Em relatos de algumas empresas, eles aplicam esse trabalho para verificar design UML manualmente.

Capítulo 7

Conclusões

Neste trabalho de mestrado investigamos uma abordagem para a verificação do design descrito através de diagramas de classes UML sobre a sua implementação em linguagem Java. Com o objetivo de realizar essa verificação propusemos uma abordagem baseada em dois conceitos: testes de design e templates de teste.

Nós propusemos um catálogo de templates de testes de design que verificam uma série de artefatos dos diagramas de classe. Neste catálogo, não cobrimos todos os artefatos da UML para diagramas de classes, pois esta é bastante vasta e completa. Contudo, para alcançarmos uma maior aceitação possível da nossa abordagem, tentamos cobrir os artefatos mais usados: Classe, Atributo, Método, Interface, Associação e Pacote.

A aplicação de nossa abordagem pode ser bastante eficaz para garantir a manutenção do design no código. Contudo, na prática, a aplicação dessa abordagem possui algumas limitações: necessidade do aprendizado de uma nova API, grande quantidade de testes para grandes designs, criação manual pode gerar muitos erros, etc. Sendo assim, desenvolvemos a ferramenta UDT capaz de verificar automaticamente a conformidade entre a implementação em Java de um sistema e o seu design UML usando testes de design gerados automaticamente pela ferramenta. O UDT se baseia no *framework* de MDA para aplicar os templates sobre os diagramas de classes, e gerar o código dos testes de

design para cada artefato do diagrama.

Construímos a ferramenta UDT para que ela servisse não somente como forma de aplicação automática da nossa solução, mas também que pudesse ser usada como plataforma de validação da nossa abordagem. Para validarmos nossa abordagem definimos que ela deveria ser realizada através da aplicação de um estudo de caso sobre um software usado na prática. Dado que esse software não possui disponível o seu design expresso através de um diagrama de classe UML, nós propusemos a seguinte metodologia de avaliação: inicialmente, aplicamos engenharia reversa sobre o código fonte do *Findbugs* para a criação do diagrama de classe desse sistema; depois disso, usando o UDT, testamos se o diagrama gerado estava em conformidade com o código original; e por fim, analisamos os resultados obtidos. Na análise dos resultados obtidos comprovamos a eficácia da nossa abordagem, pois para o cenário utilizado na avaliação dos sistema, o menor resultado alcançado para a precisão foi o 80,5% para classe, e ainda justificamos o fato de não termos alcançado 100,00%. Além disso, verificamos que nossa abordagem consegue tratar com uma grande massa de dados com um bom desempenho, devido o fato de que mesmo para o cenário mais complexo abordado (o *Findbugs*), o sistema gastou 18 min para gerar a suite de testes de design e 24 seg para executá-la.

7.1 Contribuições

Nossa abordagem contribui em vários setores da engenharia de software, conforme ressaltaremos a seguir.

Processo de desenvolvimento. Com a adoção da nossa abordagem em alguns processos de desenvolvimento, podemos ter melhoras significativas na confiança de que o código final mantêm o design estrutural projetado. Ressaltando que nossa abordagem não sobrecarregaria o processo como um todo, pois: (1) os testes de design são todos ge-

rados automanticamente, onde na prática esse processo de verificação é completamente manual, e com a nossa abordagem esse processo pode ser realizado completamente automático ou com o mínimo de revisão manual (para os artefatos do diagrama de classe que não foram abordados nesse trabalho); e (2) os resultados mostrados sobre o desempenho da ferramenta são aceitáveis para um processo de desenvolvimento, seção 5.2.2 do capítulo de avaliação.

Manutenção da documentação. O UDT pode servir como elo entre a documentação do design e a implementação. Como já foi ressaltado na introdução, a paridade entre a documentação do design e o código é fundamental para facilitar a manutenção e compreensão posterior do código. Sendo assim, com a adoção do UDT pode-se aumentar a confiabilidade em que, durante a evolução do sistema como um todo, o código não evolua sem que a documentação evolua junto, e vice-versa.

Verificação de engenharia Reversa. O UDT pode servir como plataforma de validação de ferramentas de engenharia reversa de diagramas de classes UML a partir de códigos fonte em Java. Um esboço dessa verificação foi realizado na seção 6.3, onde constatamos que a engenharia reversa da ferramenta MagicDraw gera muitos artefatos no diagrama de classe produzido que não estão presentes no código fonte.

Verificação de geração automática de código. O UDT pode servir ainda como plataforma de validação de ferramentas que geram código automaticamente a partir de diagramas de classe UML. O papel do UDT nessa verificação seria o de constatar se a estrutura do código gerado reflete a estrutura do diagrama de classe fonte.

MDA comportamental. Apesar de MDA ser amplamente usada atualmente, poucos trabalhos geram o código completo das aplicações. Ou seja, muitas aplicações de MDA geram códigos parciais. E nossa solução gera os códigos completos dos testes. Isso se deve ao módulo de geração de sintaxe concreta, mostrado na seção 4.1.3. E dado ao fato de reusarmos o *framework* de MDA, esse módulo pode ser facilmente reusado por outras aplicações, desde que estas gerem modelos de acordo com o meta-modelo da

linguagem Java.

7.2 Trabalhos Futuros

Com a finalização do trabalho, tivemos a possibilidade de fazer uma análise do mesmo, a qual resultou no seguinte conjunto de propostas para a sua continuidade:

- **Alcançar mais artefatos do diagrama de classe UML**. Criar mais templates e a geração de testes de design para verificar um maior número de características de UML (por exemplo, *dependency*, *port*, etc.), até o ponto de alcançarmos todos os artefatos da mesma;

- **Avaliação mais abrangente**. Nossa avaliação foi paltada na engenharia reversa, mais especificamente a engenharia reversa da ferramenta MagicDraw, isso limitou a avaliação da nossa abordagem, um exemplo dessa limitação é o fato de não termos verificado Associações com o mesmo rigor que avaliamos os demais artefatos do diagrama de classe UML. Dessa forma, um trabalho futuro será realizar uma avaliação mais abrangente da abordagem proposta. Uma proposta da forma como essa avaliação poderia ser realizada é através do uso de clones de diagramas de classe. A idéia dessa avaliação é, inicialmente, adotar um projeto que possua a implementação em conformidade com seu diagrama de classe (ou seja, que passe nos testes de design gerados pelo UDT), em seguida, criar automaticamente cópias (clones) desses diagramas de classe alterando alguma característica do original, e por fim, verificar se o código final ainda está em conformidade com diagrama;

- **Verificação de design comportamental**. Criar uma abordagem capaz de verificar designs comportamentais sobre informações adquiridas a partir dos diagramas comportamentais de UML, tais como diagramas de sequência e atividade. Tal

abordagem pode ser realizada através da monitoração da execução do sistema através de artefatos gerados automaticamente, ao invés de uma análise estática do código do sistema, como foi adotado no UDT para a verificação do design estrutural;

- **Verificação de design patterns**. Estender nossa abordagem de forma que possamos explicitar nos diagramas de classe onde os design patterns então sendo usados, e assim, possamos gerar testes de design específicos para a aplicação de cada pattern;

- **Verificação de Diagrama de classes para outras linguagens**. Estender os templates e a implementação do UDT para ela ser capaz de verificar a correta implementação dos diagramas de classe, não somente em códigos na linguagem Java, mas para qualquer linguagem orientada a objeto;

- **Criação de um plugin de verificação**. Criar um plugin para integrar a ferramenta UDT ao ambiente de desenvolvimento. Dessa forma, facilitando a adoção e utilização da mesma.

Bibliografia

[1] The peer-review process. *Learned Publishing*, 15:247–258, Outubro 2002.

[2] Altova UModel. http://www.altova.com/products/umodel/.

[3] AMMA Project. Atlas Transformation Language, 2005. http://www.sciences.univ-nantes.fr/lina/atl/.

[4] Anneke Kleppe, Jos Warmer e Wim Bast. *MDA Explained: The Model Driven Architecture—Practice and Promise*. Addison-Wesley, 2003.

[5] Api java 2 platform se v1.4.2, interface runnable. http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Runnable.html.

[6] ArgoUML. http://argouml.tigris.org/.

[7] Asm bytecode manipulation. http://asm.objectweb.org/.

[8] ATL-DT. http://www.eclipse.org/m2m/atl/.

[9] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.

[10] Borland Together. http://www.borland.com/br/products/together/.

[11] Clint Morgan, Kris De Volder e Eric Wohlstadter. A static aspect language for checking design rules. Em Brian M. Barry e Oege de Moor, editors, *AOSD*, volume

208 of *ACM International Conference Proceeding Series*, páginas 63–72. ACM, 2007.

[12] James J. Horning e Yang Meng Tan David Evans, John V. Guttag. LCLint: A tool for using specifications to check code. Em *Symposium on the Foundations of Software Engineering*, Dezembro 1994.

[13] David H. Akehurst, Gareth Howells e Klaus D. McDonald-Maier. Implementing associations: UML 2.0 to java 5. *Software and System Modeling*, 6(1):3–35, 2007.

[14] David Lorge Parnas. Software aging. Em Bruno Fadini, editor, *Proceedings of the 16th International Conference on Software Engineering*, páginas 279–290, Sorrento, Italy, Maio 1994. IEEE Computer Society Press.

[15] Design Wizard. http://www.designwizard.org.

[16] Ecore. http://www.eclipse.org/modeling/emf/?project=emf.

[17] Erich Gamma, Richard Helm, Ralph Johnson e John M. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.

[18] Len Erlikh. Leveraging legacy system dollars for E-business. *IT Professional*, 2(3):17–23, 2000.

[19] Find Bugs. http://findbugs.sourceforge.net/.

[20] Frédéric Jouault e Jean Bézivin. KM3: a DSL for metamodel specification. Em *IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037*, páginas 171–185. Springer, 2006.

[21] Gail C. Murphy, David Notkin e Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. Em G. Kaiser, editor, *Proc.*

3rd ACM SIGSOFT Symp. on the Foundations of Software Engineering, volume 20:4 of *ACM SIGSOFT Software Engineering Notes*, páginas 18–28, Washington, DC, Outubro 1995.

[22] Object Management Group. Meta object facility (MOF) specification. Technical Report ad/97-08-14, Object Management Group, 1997.

[23] Object Management Group. Mof model to text transformation language. Technical Report ad/2004-04-07, Object Management Group, 2004.

[24] JAS-metamodel. http://www.eclipse.org/gmt/am3/zoos/atlanticZoo.

[25] Jesse E. Tilly e Eric M. Burke. *Ant: The Definitive Guide*. O'Reilly & Associates, Inc., pub-ORA:adr, 2002.

[26] Java Emitter Templates. http://www.eclipse.org/articles/Article-JET/.

[27] Jilles van Gurp e Jan Bosch. Design erosion: problems and causes. *The Journal of Systems and Software*, 61(2):105–119, Março 2002.

[28] Dalton Guerrero e Jorge Figueiredo João Brunet. Design tests: An approach to programmatically check your code against design rules. Em *Proc. 31th International Conference on Software Engineering (ICSE 2009), New Ideas and Emerging Results*, Maio 2009.

[29] John Robert Taylor. An introduction to error analysis: The study of uncertainties in physical measurements. páginas 128–129. University Science Books, 199.

[30] JUnit. http://www.junit.org.

[31] Len Bass, Paul Clements e Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[32] MagicDraw UML. http://www.magicdraw.com/.

[33] Mark Utting e Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2006.

[34] MDT OCL. http://www.eclipse.org/modeling/mdt/?project=ocl.

[35] MOFScript. http://www.eclipse.org/gmt/mofscript/.

[36] MySQL. http://www.mysql.com/.

[37] Yann-Gael Gueheneuc e Pierre Leduc Naouel Moha. Automatic generation of detection algorithms for design defects. Em *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, páginas 297–300, Washington, DC, USA, 2006. IEEE Computer Society.

[38] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix e YuQian Zhou. Using findbugs on production software. Em Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, e Guy L. Steele Jr, editors, *OOPSLA Companion*, páginas 805–806. ACM, 2007.

[39] Object Management Group. http://www.omg.org/.

[40] Object Management Group, Framingham, Massachusetts. *UML 2.0 Superstructure Specification*, Outubro 2004.

[41] Object Management Group. MOF, 2006. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[42] Object Management Group. The OCL 2.0 Specification, 2006. http://www.omg.org/technology/documents/formal/ocl.htm.

[43] OMondo - Eclipse UML. http://www.eclipsedownload.com/.

[44] P. A. Stocks e D. A. Carrington. Test templates: a specification-based testing framework. Em *Proceedings of the 15th International Conference on Software Engineering*, páginas 405–414. IEEE Computer Society Press, April 1993.

[45] Philippe Kruchten, Patricia Lago e Hans van Vliet. Building up and reasoning about architectural knowledge. Em Christine Hofmeister, Ivica Crnkovic, e Ralf Reussner, editors, *QoSA*, volume 4214 of *Lecture Notes in Computer Science*, páginas 43–58. Springer, 2006.

[46] Poseidon for UML. http://www.gentleware.com/.

[47] Ian Sommerville. *Software Engineering*. Addison-Wesley, 7th edition, 2004.

[48] Stephen Eick, Todd Graves, Alan Karr, J. Marron e Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[49] Trung T. Dinh-Trong, Nilesh Kawane, Sudipto Ghosh, Robert B. France e Anneliese Amschler Andrews. A tool-supported approach to testing UML design models. Em *ICECCS*, páginas 519–528. IEEE Computer Society, 2005.

[50] UDT - UML Design Tester. http://www.designwizard.org/index.php/UDT/.

[51] Victor R. Basili, Gianluigi Caldiera e H. Dieter Rombach. Goal Question Metric Paradigm. Em John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, páginas 528–532. John Wiley & Sons, 1994.

[52] Visual Paradigm for UML. http://www.visual-paradigm.com/product/vpuml/.

[53] Franklin Ramalho e Dalton Serey Waldemar Pires, Jo/ ao Brunet. Uml-based design test generation. Em *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, páginas 735–740, New York, NY, USA, 2008. ACM.

[54] William Wesley Peterson. *Introduction to Programming Languages*. Prentice Hall, Englewood Cliffs, 1974.

[55] Wolfgang Hesse. RUP - A process model for working with UML. Em *Unified Modeling Language: Systems Analysis, Design and Development Issues*, páginas 61–74. 2001.

[56] Xalan-Java Version 2.7.1. http://xml.apache.org/xalan-j/.

[57] XSL Transformations. http://www.w3.org/TR/xslt/.

[58] Rémi Douence e Pierre Cointe Yann-Gael Guéhéneuc, Hervé Albin-Amiot. Bridging the gap between modeling and programming languages. Technical report, Computer Science Department, École des Mines de Nantes, 2002.

Apêndice A

Templates de Teste de Design

Neste apêndice mostraremos todos os códigos dos teste de templates tratados neste trabalho.

```
1  public void testClass<dif><NomeDaClasse>() throws IOException,InexistentEntityException{
2    ClassNode aClass = dw.getClass("<NomeDaClasse>");
3    assert<TrueFalse>("The class <NomeDaClasse> must <not> be abstract",aClass.isAbstract());
4    ClassNode superClass = dw.getClass("<NomeDaSuperClasse>");
5    assertEquals("The class <NomeDaClasse> must extend the class <NomeDaSuperClasse>",superClass,
6          aClass.getSuperClass());
7    Collection <Modifier> modifs = aClass.getModifiers();
8    assertTrue("The visibility of class <NomeDaClasse> must be <visibility>",
9          modifs.contains(Modifier.<visibility>));
10   String[] superInterfaces = {"<NomeDaInterface1\>" , "<NomeDaInterface2>" ,...};
11   for (String superInterface : superInterfaces) {
12     ClassNode classnodeInterface = dw.getClass(superInterface);
13     Set<ClassNode> interfacesExtend = aClass.getImplementedInterfaces();
14     assertTrue("The class <NomeDaClasse> must realize the interface " + superInterface,
15         interfacesExtend.contains(classnodeInterface));
16   }
17 }
```

Código Fonte A.1: Template do Teste de Design para Classe.

```java
public void testAttribute<dif><NomeDoAtributo> () throws IOException,InexistentEntityException {
  ClassNode aClass = dw.getClass("<NomeDaClasse>");
  FieldNode attrClass = aClass.getField("<NomeDoAtributo>");
  assertTrue("The attribute <NomeDoAtributo> of the class <NomeDaClasse> must <not> be static",
     attrClass.isStatic());
  Collection<Modifier> modifs = attrClass.getModifiers();
  assertTrue("The attribute <NomeDoAtributo> of the class <NomeDaClasse> must be <visibilidade>"
     ,modifs.contains(Modifier.<visibilidade>));
  assertEquals("The attribute <NomeDoAtributo> of the class <NomeDaClasse> must return the type "+
     "<TipoDoAtributo>", <TipoDoAtributo>, attrClass.getDeclaredType().getName());
}
```

Código Fonte A.2: Template do Teste de Design para Atributo.

```
1  public void testMethod<dif><MethodName>() throws IOException,InexistentEntityException{
2   ClassNode aClass = dw.getClass(\"<ClassName>");
3   MethodNode methodClass = aClass.getMethod("<MethodName>");
4   assertEquals("The return of the method <MethodName> from the class <ClassName>must be<TypeName>",
5       "<TypeName>", methodClass.getReturnType().getName());
6   assert<True False>("The method <MethodName> from the class <ClassName> must <not> be abstract",
7       methodClass.isAbstract());
8   assert<TrueFalse>("The method<MethodName> from the class <ClassName> must <not> be static",
9       methodClass.isStatic());
10  Collection<Modifier> modifs = methodClass.getModifiers();
11  assertTrue("The visibility of The method<MethodName> from the class <ClassName> must be "+
12      "<visibility>",modifs.contains(Modifier.<visibility>));
13 }
```

Código Fonte A.3: Template do Teste de Design para Operações.

```java
1  public void testInterface<dif><NomeDaInterface>()
2          throws IOException, InexistentEntityException{
3    ClassNode aClass = dw.getClass("<NomeDaInterface>");
4    assertTrue("The class <NomeDaInterface> must be a interface",
5      aClass.isInterface());
6    String[] superInterfaces =
7      {"<superInterface1>","<superInterface2>",",...};
8    for (String superInterface : superInterfaces) {
9      ClassNode cnInterface = dw.getClass(superInterface);
10     Set<ClassNode> interfacesExtend =
11         aClass.getImplementedInterfaces();
12     assertTrue("The interface <NomeDaInterface> must extends from "+
13          superInterface,interfacesExtend.contains(cnInterface));
14   }
15 }
```

Código Fonte A.4: Template do Teste de Design para Interface.

```
1  public void testPackage<dif><NomeDoPacote>() throws java.io.IOException,InexistentEntityException
     {
2    PackageNode thePackage = dw.getPackage("<NomeDoPacote>");
3    String[] importedPackages = {"<PackageImp1>", "<PackageImp2>",... };
4    Set<ClassNode> internalEntyties = thePackage.getAllClasses();
5    Set<ClassNode> extentableEntiites = internalEntyties;
6    for (String aPackage : importedPackages) {
7       extentableEntiites.addAll(dw.getPackage(aPackage).getAllClasses());
8    }
9    for (ClassNode aEntity : internalEntyties) {
10     assertTrue(aEntity+" cannot extends the class " + aEntity.getSuperClass(),
11         extentables.contains(aEntity.getSuperClass()));
12     for (ClassNode aInterface:aEntity.getImplementedInterfaces()){
13        assertTrue(aEntity+" cannot implements the interface" + aInterface,
14           extentableEntiites.contains(aInterface));
15     }
16   }
17 }
```

Código Fonte A.5: Template do Teste de Design para Pacote.

```
1  public void testAssociationClass<dif><NomeTipoFonte><NometipoAlvo> ()
2          throws InexistentEntityException, IOException{
3    ClassNode c1 = dw.getClass("<NomeTipoFonte>");
4    FieldNode f1 = c1.getField("<papel>");
5    MethodNode getAssoc1 = c1.getMethod("get<papel>");
6    MethodNode setAssoc1 = c1.getMethod("set<papel>");
7    assertEquals("the method get<papel> of the class <NomeTipoFonte> must return the type "+
8      "<NomeTipoAlvo>", "<NomeTipoAlvo>", getAssoc1.getReturnType().getName());
9    assertEquals("the method set<papel> of the class <NomeTipoFonte> must return the type void",
10     "void", setAssoc1.getReturnType().getName());
11   assertEquals("the attribute <papel> of the class <NomeTipoFonte> must return the type "+
12     "<NomeTipoAlvo>", "<NomeTipoAlvo>", f1.getDeclaredType().getName());
13   Collection <Modifier> modifs = f1.getModifiers();
14   assertTrue("the attribute <papel> of the class <NomeTipoFonte> must be private"
15     ,modifs.contains(Modifier.PRIVATE));
16   modifs = getAssoc1.getModifiers();
17   assertTrue("the method get<papel> of the class <NomeTipoFonte> must be <visibilidade>"
18     ,modifs.contains(Modifier.<visibilidade>));
19   modifs = setAssoc1.getModifiers();
20   assertTrue("the method set<papel> of the class <NomeTipoFonte> must be <visibilidade>"
21     ,modifs.contains(Modifier.<visibilidade>));
22   Collection<MethodNode> methodsGetAssoc = getAssoc1.getCalledMethods();
```

```
23    boolean isReadOnly = false;
24    for( MethodNode method: methodsGetAssoc ){
25        if(method.getName().equals("java.util.Collections.unmodifiable")){
26            isReadOnly = true;
27            break;
28        }
29    }
30    assertTrue("The method set<papel> must returns a java.util.Collections.unmodifiable",
31          isReadOnly);
32  }
```

Código Fonte A.6: Template do Teste de Design para Associa ção.