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

Verificação de Conformidade entre Diagramas de Sequência UML e Código Java

Sebastião Estefânio Pinto Rabelo Júnior

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 & Dalton Dario Serey Guerrero
(Orientadores)

Campina Grande, Paraíba, Brasil

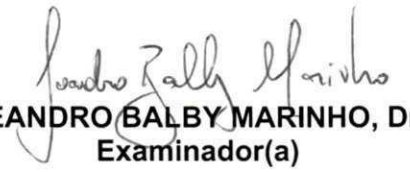
"VERIFICAÇÃO DE CONFORMIDADE ENTRE DIAGRAMAS DE SEQUÊNCIA UML E CÓDIGO JAVA"

SEBASTIÃO ESTEFÂNIO PINTO RABELO JUNIOR

DISSERTAÇÃO APROVADA EM 30.11.2011

DALTON DARIO SEREY GUERRERO, D.Sc
Orientador(a)

FRANKLIN DE SOUZA RAMALHO, Dr.
Orientador(a)



LEANDRO BALBY MARINHO, Dr.
Examinador(a)

AYLA DÉBORA DANTAS DE SOUZA REBOUÇAS, D.Sc
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Atualmente, quando se fala em UML, temos os diagramas de sequência como o mais popular entre os diagramas usados para descrever aspectos comportamentais de um software. Por outro lado, temos Java como uma das linguagens orientadas a objetos mais usada no mundo. Entretanto, não encontramos em nossas pesquisas um meio sistêmico para a verificação automática de conformidade entre modelos comportamentais e o código desenvolvido para atender esse modelo. Nesta dissertação, nós desenvolvemos uma abordagem capaz de verificar esse tipo de conformidade. O uso dessa abordagem permitirá ajudar desenvolvedores, analistas, e gerentes de projeto a manter a documentação do software atualizada, além de possibilitar a existência de um novo ponto de vista a respeito de defeitos na implementação de um sistema. Para dar suporte a essa verificação de conformidade nós desenvolvemos uma ferramenta baseada em Model Driven Architecture (MDA) capaz de gerar os testes de conformidade aqui apresentados. Além disso, esta dissertação traz uma avaliação da abordagem desenvolvida, a qual apresenta os principais resultados obtidos.

Palavras Chaves. Verificação de conformidade, Testes de *design*, *Unified Modeling Language*, Java, *Model Driven Architecture*.

Abstract

Currently, sequence diagrams are the most popular UML diagrams used to describe behavioral aspects of software systems. On the other hand, Java as one of the most popular object-oriented language used in the world. Despite that, there is no systematic approach to support verification between the behavioral design and the implemented source code. In this work, we propose an approach to verify this conformity. The use of this approach will help developers, architects, and engineers to maintain the software documentation updated. Its usage allows that the development team and managers to detect behavioral design implementation defects. We also present the tool support built for our approach using Model Driven Architecture (MDA) and a preliminary evaluation about this work.

Keywords. Conformance checking, *Design* tests, Unified Modeling Language, Java, Model Driven Architecture.

Agradecimentos

Agradeço a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) por possibilitar a realização deste trabalho através da sua bolsa de estudos para o mestrado.

Agradeço a meus professores, Franklin e Dalton, pela sua competência, vontade, paciência e dedicação durante todo o tempo em que estive sob suas tutelas. Aprendi lições importantes com os senhores, não apenas acadêmicas.

Com maior gratidão, agradeço ao apoio de meus amigos mais próximos e familiares que tanto me deram suporte e força pra continuar nesta luta, principalmente a meus pais (Sebastião e Heronice) e minhas irmãs (Mônica e Mariana) pela dedicação, carinho e lições que só a família pode dar.

Por fim e não menos importante, agradeço imensamente a minha namorada (Dayanna) que esteve presente na maior parte do tempo, e acompanhou com amor e carinho todos os momentos dessa batalha.

Conteúdo

Lista de Símbolos

UML - *Unified Modeling Language*

MDA - *Model Driven Architecture*

GQM - *Goal, Question, Metric*

LOC - *Lines Of Code*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Modelos são abstrações de *software* construídas em fases iniciais de diversos processos de desenvolvimento, como por exemplo, o Processo Unificado [AN05], e seus derivados RUP [Kru00] e OpenUP [Ope11]. Para criar esses modelos, os projetistas podem usar diversas linguagens visuais, sendo *Unified Modeling Language* (UML) [Gro11b] uma das mais famosas e utilizadas.

A UML foi criada pela Object Management Group (OMG) [OMG11] e é capaz de representar abstrações tanto estruturais quanto comportamentais de sistemas orientados a objetos através de modelos ou diagramas. Cada tipo de diagrama é responsável por representar uma ou mais características do projeto de um software. Por exemplo, o diagrama de classes é usado para visualizar as entidades que compõem o sistema bem como as relações entre essas entidades, *i.e.*, associações, heranças, dependências. Para representar o comportamento de um sistema podemos usar modelos chamados diagramas de interação, dos quais o mais utilizado é o diagrama de sequência [DP06]. Neste tipo de diagrama, as entidades que compõem o *software* são interligadas através de trocas de mensagens. Definindo qual deve ser a sequência destas trocas e quais pares de objetos participam de cada troca em cada momento, o projetista é capaz de modelar como o software deve se comportar.

De modo análogo, *softwares* escritos em Java, os quais também são orientados a objetos, têm seu comportamento implementado por meio de trocas de mensagens entre objetos. É durante a codificação que os desenvolvedores devem olhar para os modelos que descrevem o sistema e tomá-los como guia para o desenvolvimento. Entretanto, a solução desenvolvida nem sempre está de acordo com o que foi descrito nos modelos. Existe ainda um complicador

quando os modelos descrevem o comportamento do sistema, pois o mesmo comportamento pode ser codificado de maneiras distintas, a depender do modo de trabalho do desenvolvedor. Por exemplo um comando condicional simples pode, em alguns casos, ser substituído por um comando ternário de Java (<expressão booleana> ? <valor se verdadeiro> : <valor se falso>).

Manter a conformidade entre a documentação do *software* e o seu código fonte é um problema que vem sendo tratado em diferentes abordagens. Essas abordagens vão desde a checagem em relação à arquitetura do sistema, em um nível mais alto de abstração [AAA07], até a relação entre as classes que compõem o sistema, através de, por exemplo, suas associações e agregações [DTKG+05].

É comum a adoção de verificação de conformidade através de processos executados por humanos. Algumas técnicas, como a apresentada por Murphy *et al.* [MNS01], necessitam de um engenheiro de software, ou outro membro da equipe que possua grande conhecimento a respeito do sistema sob teste. Em outros casos, como na solução de Huynh *et al.* [HCSS08], a automação do processo de inspeção reduz consideravelmente a necessidade de um especialista na execução da verificação.

No contexto desta dissertação, Pires *et al.* [PRSL08] apresenta uma maneira de construir testes automáticos de *design* com base em diagramas de classe UML, para *softwares* escritos em Java. Testes de design são uma forma de teste capaz de verificar conformidade estrutural entre modelos e *softwares* implementados. Neste trabalho, Pires *et al.* definem o conceito de regras de *design*, *i.e.*, retrições aplicadas à estrutura do sistema, as quais são escritas para verificar se as relações estruturais, definidas em diagramas de classe, estão sendo seguidas no desenvolvimento do *software*, conforme planejado. Vale ressaltar que, para dar suporte a aplicação dessas regras, Pires *et al.* adotaram o Design Wizard [DWS11] como API para consultar às informações estruturias do sistema sob teste, a partir do código fonte existente.

O conceito de testes de *design* foi desenvolvido por Brunet et al., que aplicaram as regras de *design* para construir esse tipo especial de testes o qual é não capaz de verificar se as funcionalidades do sistema estão corretas, mas se foram desenvolvidas de acordo com especificação desse sistema. Posteriormente, Pires *et al.* aplicaram a técnica de teste de *design* a alguns *design patterns*, a exemplo do *Singleton*, do *Proxy* e do *Facade* [PRLS10].

Esta dissertação aparece como uma extensão dos trabalhos de Pires *et al.* [PRSL08] que verificará conformidade comportamental por meio da adição de testes dos aspectos compor-

tamentais dos *softwares* descritos em diagramas de sequência UML para produção de testes de *design* mais elaborados e confrontando-os com o código Java destes *softwares*. Utilizamos diagramas de sequência pois este é o tipo mais usado entre os diagramas de interação [DP06], e também foi devido à popularidade da linguagem Java que a escolhemos.

Tratando especificamente de diagramas de sequência UML, temos um complicador para a checagem de conformidade a partir da versão 2.0 da linguagem. Nesta versão foi introduzido o uso de fragmentos combinados - elementos que uma vez introduzidos numa sequência são capazes de alterá-la de acordo com a semântica do fragmento escolhido. Assim sendo, fragmentos combinados funcionam como comandos, ou desvios de fluxo, em linguagens textuais. Vale ressaltar que existem 12 tipos de fragmentos combinados distintos em UML. Mais que isso, os fragmentos combinados podem ser dispostos de diversas formas dando mais liberdade para criação de projetos. Essa liberdade criativa possibilita a formação de sistemas complexos. Assim, quanto mais complexa for a modelagem, mais complicado será a realização da verificação de conformidade através dos testes de *design*.

A implementação de testes de *design* deve atingir certa riqueza de detalhes para ter abrangência maior possível no que diz respeito à relação entre os modelos e a implementação do *software* sob teste. Esta riqueza torna a escrita dos testes uma tarefa extensa, cansativa e propensa a erros. Para contornar esse problema Pires *et al.* [PRSL08] desenvolveram uma abordagem *Model Driven Architecture* (MDA) para automatizar a construção de testes a partir dos diagramas de classe UML. Nesta dissertação usamos a mesma premissa para desenvolver uma abordagem MDA capaz de construir automaticamente os testes de *design* a partir de diagramas de sequência UML.

Assim sendo, este trabalho tenta resolver o problema da incongruência entre modelos visuais escritos em UML e programas codificados em Java por meio de templates de testes de *design*, *i.e.*, padrões que indicam uma maneira de aplicar-se os testes de *design*. Mais especificamente trabalhamos a verificação de conformidade entre diagramas de sequência UML e códigos Java através de testes de *design* comportamentais.

Escopo. O escopo do trabalho restringe-se à verificação de troca de mensagens entre objetos; ao uso dos fragmentos combinados *opt*, *alt*, e *loop* dos modelos UML, e à relação desses elementos com os comandos correspondentes em Java. Os demais fragmentos com-

binados não fazem parte deste trabalho. Portanto, a técnica apresentada aqui não oferece nenhum suporte aos fragmentos que estão fora desse escopo.

Contribuições. A realização deste trabalho tem as seguintes contribuições esperadas:

- Um conjunto de testes de *design* capaz de verificar conformidade comportamental entre UML e Java.
- Uma extensão do Design Wizard capaz de recuperar informações comportamentais do código fonte escrito em Java.
- Um padrão de teste de *design* capaz de abranger os elementos de *software* descritos no escopo do trabalho.
- Uma técnica capaz de extrair informações de diagramas de sequência, e usar estas informações para construir testes de *design*.
- Uma ferramenta para geração dos novos testes de *design* de modo automático, para facilitar o processo de escrita desses testes.
- A análise da técnica desenvolvida através da aplicação da mesma em estudos de caso, usando a ferramenta construída.

Este documento descreve como este trabalho foi realizado e está organizado da seguinte forma: No Capítulo 2 aprofudamos um pouco mais a proposta, os objetivos e a metodologia adotada no desenvolvimento dos trabalhos envolvendo esta dissertação. Dando sequência, no Capítulo 3 estão descritos os conceitos que servem de base para o desenvolvimento deste trabalho. Já no Capítulo 4 encontramos a descrição do modelo de dados adotado na representação dos elementos que compreendem as entidades UML e Java. Capítulo 5 trata da aplicação do da representação dos elementos, através do modelo de dados, na criação e execução dos templates de teste para cada entidade do diagrama de sequência UML dentro do escopo desta dissertação. Para preparar a validação deste trabalho adotamos o modelo *Goal Question Metric* (GQM) [BCR94] [vSB99], cuja descrição e utilização está descrita no Capítulo 6. No Capítulo 7 apresentamos alguns trabalhos relacionados e, por fim, o Capítulo 8 traz as conclusões desta dissertação e alguns possíveis trabalhos futuros.

Capítulo 2

Construção da Abordagem

Neste capítulo trataremos da construção da abordagem desenvolvida nesta dissertação. Apresentamos aqui a motivação do trabalho, bem como os objetivos, e a metodologia adotada. Este capítulo tem o objetivo de contextualizar o trabalho desenvolvido de forma a preparar a o leitor para o que será apresentado posteriormente nesta dissertação. Detalhes mais profundos sobre os pontos abordados aqui serão apresentados detalhadamente nos próximos capítulos.

2.1 Motivação do Trabalho

Nossa motivação para esta dissertação apoia-se em dois problemas que ocorrem no desenvolvimento de sistemas onde o maior foco está na manutenção da documentação do projeto. O primeiro problema é relativo à manutenção da documentação e do código implementado em sincronismo constante, sem que para isso se gaste recursos excessivos de pessoal. Com isso, chegamos ao nosso segundo problema, o qual acontece mesmo em sistemas onde há manutenção da documentação visando apenas aspectos estruturais, pois nesses casos o sistema segue a arquitetura planejada, mas não se comporta como esperado. Assim sendo, mesmo tendo sistemas bem documentados e estruturados, ainda existem casos em que o comportamento do sistema não acontece como planejado. Este último problema é o principal motivador do trabalho apresentado nesta dissertação.

2.2 Objetivos

Nosso objetivo ao realizar o trabalho envolvendo esta dissertação é construir uma abordagem capaz de automatizar o processo de verificação entre o comportamento definido no projeto de um sistema e o código correspondente a este comportamento, para assim poder confirmar se ambos estão de acordo.

2.3 Metodologia Adotada

A metodologia adotada para que conseguissemos atingir o objetivo desta dissertação corresponde a um conjunto de passos que foram seguidos na seguinte sequência:

1. Escolha das linguagens que seriam objeto de estudo desta dissertação, sendo um visual (documentação) e outra textual (código fonte).

2. Estudo da correspondência semântica entre os elementos componentes das duas liguagens.

3. Definição de uma representação comum para os elementos de ambas as linguagens, para que a correspondência entre ambas possa ser melhor analisada.

4. Definição de padrões, ou templates, para a construção dos teste de verificação de conformidade entre as linguagens, utilizando a representação comum dos elementos de ambas as linguagens.

5. Criação do mecanismo para automatizar a criação da verificação de conformidade do sistema sob teste.

6. Avaliação da abordagem desenvolvida e análise dos resultados obtidos.

Capítulo 3

Fundamentação Teórica

Este capítulo busca introduzir os principais assuntos abordados nesta dissertação com o intuito de facilitar o entendimento do trabalho desenvolvido.

3.1 Modelagem de Comportamento com UML 2.0

Veremos nesta seção todos os conceitos que envolvem diagramas de sequência UML. Assim, o leitor que desconhece este tipo de diagrama poderá entender melhor a proposta deste trabalho.

A UML 2.0 possui diversas formas de representar o comportamento de um sistema, cada uma dessas formas é representada através de diagramas agrupados em pacotes. A Figura 3.1 apresenta a relação entre esses pacotes de representação comportamental. Esta relação faz parte do metamodelo que descreve a UML [Gro09a][Gro09b].

O pacote *Common Behaviors* depende de características de representação de classes do pacote *Classes*, tais como a nomeação de elementos e as relações estruturais entre eles. Essas relações estruturais ajudam a modelar as relações comportamentais, pois delimitam quais elementos estão interligados. O pacote *Common Behaviors* fornece elementos, por dependência, para os pacotes que definem os quatro principais tipos de diagramas comportamentais. O pacote *Activities* define como devem ser os diagramas de atividade. O pacote *UseCases* define os diagramas de caso de uso. Já o pacote *StateMachine* define as máquinas de estado. Por fim, e mais importante para esta dissertação, existe o pacote *Interactions* que define como devem ser os diagramas de comunicação, de *overview* de interação, de tempo,

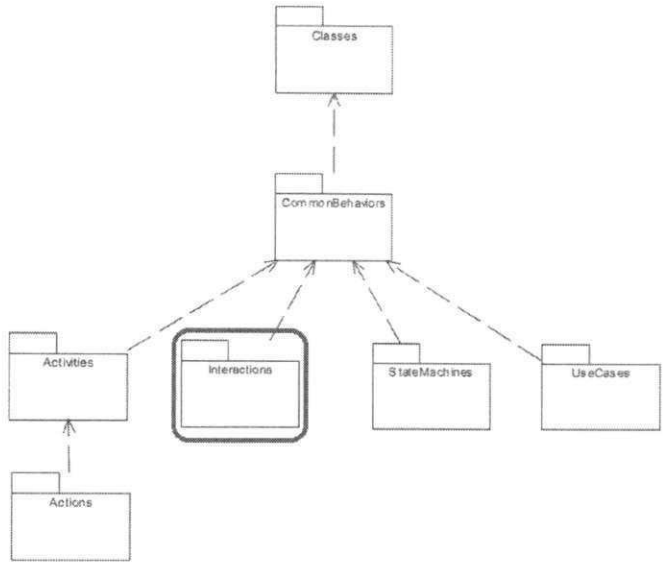


Figura 3.1: Pacotes para modelagem de comportamentos em UML

e os de sequência, sendo este último apresentado nas subseções seguintes. Para mais informações a respeito dos demais pacotes e diagramas consulte a especificação UML disponível em [Gro09b].

3.1.1 Diagramas de Sequência

Diferentemente dos demais diagramas de interação, diagramas de sequência permitem que as trocas de mensagens entre objetos sejam modeladas seguindo uma ordem cronológica, *i.e.*, as mensagens devem ser trocadas uma após a outra de acordo com a sequência que aparecem no diagrama. Portanto, a compreensão dos elementos que compõem o diagrama de sequência é de importância fundamental para o entendimento do trabalho apresentado aqui.

A seguir serão apresentados alguns dos elementos que podem compor um diagrama de sequência. Para cada um deles será mostrado um exemplo indicando a sintaxe a ser usada na modelagem dos diagramas.

Linhas de vida. As linhas de vida representam os objetos do sistema que efetivamente participam da sequência de troca de mensagens sendo descrita. Sua representação é feita utilizando-se um retângulo horizontal, onde deve ser escrita a identificação do objeto junto

do nome da classe que descreve esse objeto. Logo abaixo desse retângulo deve ser usada uma linha tracejada indicando que esse objeto estará participando da sequência de troca de mensagens enquanto a linha existir. A Figura 3.2 apresenta uma linha de vida de um objeto. Este objeto é identificado por *obj* e é da classe *ObjectNode*. É possível ver ainda a linha que define a existência desse objeto no diagrama.

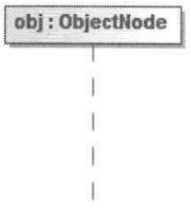


Figura 3.2: Representação da linha de vida de um objeto

Mensagens. Mensagens representam a troca de informação entre entidades de um sistema orientado a objetos. Em casos especiais, um objeto pode enviar mensagens para si mesmo, caracterizando o uso de *self-messages*. Em um diagrama de sequência, as mensagens são representadas por setas, que devem ser dispostas horizontalmente dentro do diagrama. A mensagem do topo do diagrama é considerada como a primeira mensagem, ou seja, a ordem de leitura do diagrama é da esquerda para a direita, e de cima para baixo. Vale ressaltar que a criação e a destruição de objetos também são feitas usando mensagens. Na Figura 3.3 podemos ver uma representação da troca de mensagens entre dois objetos de uma mesma classe, que neste caso chamamos de *Object*. Primeiro o objeto *obj1* manda uma mensagem *m1()* para o objeto *obj2*, depois o objeto *obj2* manda a mensagem *m2()* para o *obj1*. Por fim o *obj1* manda uma mensagem *m3()* pra si mesmo.

Foco de Controle (*Execution Specification*). Ainda na Figura 3.3 podemos perceber que existe um retângulo vertical acompanhando a linha vertical tracejada em cada objeto, e que este retângulo se inicia nas extremidades das setas. Estes retângulos são chamados foco de controle, e seu uso indica que no intervalo de tempo que ele existe, o objeto estará dedicado a executar a mensagem que iniciou o foco.

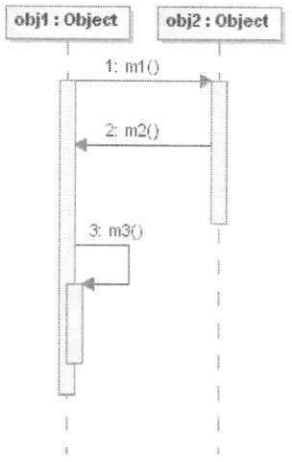


Figura 3.3: Representação de trocas de mensagem entre objetos

Eventos. Eventos são tipos especiais de mensagens que não possuem foco de controle. O uso de um evento indica atomicidade e tempo muito curto na execução de uma mensagem. A representação de um evento é levemente diferente de uma mensagem, pois a seta usada é tracejada. No contexto desta disssertação, entendemos que eventos devem ser tratados como mensagens comuns, pois não foi desenvolvido nenhum mecanismo capaz de analisar tempo de execução, nem atomicidade de mensagens.

Ocorrência de Interação (*Interaction Use*). Alguns diagramas podem ser bastante complexos, e normalmente precisam passar por reestruturação ou remodelagem. Com isso o diagrama é dividido, fazendo com que o *software* fique melhor modularizado. Além disso, as partes menores de um diagrama podem ser reusadas para compor outros diagramas posteriormente. Para interligar diagramas, ou reusar as funcionalidades já descritas em outro diagrama, podemos usar um elemento chamado ocorrência de interação, representado por um retângulo com a palavra *ref* no canto superior esquerdo, e o nome do diagrama reusado no centro. Na Figura 3.4 podemos encontrar um exemplo de modelagem do comportamento de um jogo de dados. Este diagrama foi modularizado, gerando mais três outros diagramas: *Init Game*, para começar o jogo; *roll dices* para rolar os dados; e *Get Winner*. Estes novos diagramas foram então referenciados através de Ocorrências de Interação conforme vemos na Figura 3.4.

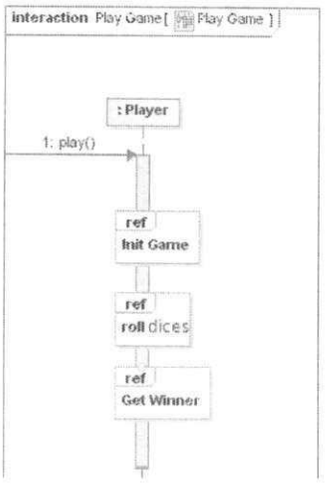


Figura 3.4: Representação do uso de ocorrências de interação

Fragmentos Combinados. A partir da versão 2.0 da UML os diagramas de sequência receberam a adição de novos elementos capazes de alterar o fluxo normal de execução modelado. Estes elementos chamam-se fragmentos combinados, e são representados por um retângulo, o qual pode em alguns casos ser dividido por linhas tracejadas, sendo cada uma dessas divisões chamada de operando. Isto implica dizer que alguns fragmentos permitem que haja mais de um desvio de fluxo em um mesmo ponto do diagrama. A Figura 3.5 apresenta uma variação da Figura 3.4, onde foi usado um fragmento combinado que modela a existência de um comando iterativo (*loop*) para alterar o fluxo normal do diagrama em questão. Neste caso, trata-se de um comando iterativo no qual está definido que o mesmo deverá ser executado 10 vezes.

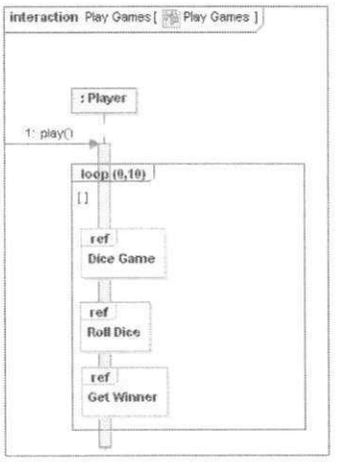


Figura 3.5: Representação do uso de fragmentos combinados

Os fragmentos combinados são divididos em 12 tipos, e cada um deles possue significado diferente na variação da sequência. Além disso, alguns tipos permitem a utilização de restrições, ou guardas, para que o desvio de fluxo aconteça. Quando houver guarda, só deve existir uma por operando. A seguir podemos encontrar a lista dos 12 tipos de fragmentos, junto de explicações sobre cada um deles.

1. *opt*: Este fragmento é utilizado obrigatoriamente com apenas um operando, pois apenas uma condição (guarda) deve ser testada antes de sua execução. Uma vez que a expressão na guarda possua valor verdadeiro, o seu operando será executado. Na Figura 3.6 podemos ver o uso de um opt onde a guarda que está entre colchetes, *i.e.*, $d1$!= $d2$, deve ser testada antes que o objeto da classe *Board* chame os métodos *roll()* dos dois objetos da classe *Dice*.

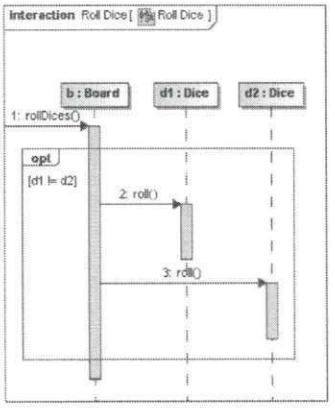


Figura 3.6: Representação do uso do fragmento combinado *opt*

2. *alt*: Este fragmento funciona de modo semelhante ao *opt*. Entretanto, é permitido o uso de mais de um operando, ou seja, mais de uma guarda pode ser testada e mais de um desvio de fluxo de execução pode ser seguido. Além disso, obrigatoriamente o último operando deve ser na guarda o valor *else*. Isto assegura que pelo menos um operando seja executado. Alterando o exemplo da Figura 3.6 encontramos na Figura 3.7 a utilização do fragmento *alt* onde testamos a guarda $d1$!= $d2$. Assim, se a guarda tiver resultado positivo chamamos o método *roll()* de $d1$ e de $d2$, senão chamamos o método *roll()* de $d1$ duas vezes.

3. *loop*: Este fragmento é usado para definir iterações no diagrama de sequência. Estas iterações podem ser de dois tipos: definidas, *i.e.*, com o numero de ciclos fixo e nesse

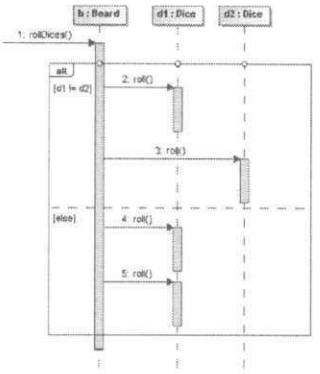


Figura 3.7: Representação do uso do fragmento combinado *alt*

caso não se usa guardas como no exemplo da Figura 3.5 onde a guarda aparece vazia; ou indefinidas, ou seja, a quantidade de ciclos inicialmente é desconhecida, assim os ciclos só páram se a guarda do operando passar a ter valor falso, neste caso a guarda é obrigatória.

4. *break*: Quando o fragmento break é executado, o fluxo da operação é desviado para o fluxo interno do operando, e após isso a operação é abortada. Como podemos ver esse fragmento é bastante útil para simbolizar o tratamento de exceções. Vale ressaltar que, quando este fragmento for combinado com outros fragmentos, e for executado, apenas a sequência que estiver dentro do fragmento que o contém será abortada.

5. *par*: Este fragmento indica que ocorre paralelismo entre os fluxos contidos em seus operandos.

6. *weak*: Este fragmento indica que a sequência contida em seu operando é fraca, ou seja, pode ser violada sem caracterizar um problema para a aplicação.

7. *strict*: Ao contrário do fragmento *weak*, este fragmento indica uma sequência que deve ser seguida à risca. Comumente estes dois fragmentos são usados em conjunto.

8. *neg*: Este fragmento representa sequências que se executadas serão consideradas inválidas. Isto é muito útil em testes, quando um trecho de código não deve ser executado.

9. *critical*: Este fragmento indica que os fluxos de seus operandos são atômicos, *i.e.*, que não deve haver interferência externa para sua execução e que os fluxos de seus

operandos devem ser executados por completo.

10. *ignore*: O uso desse fragmento indica que o fluxo contido no seu operando pode ser ignorado durante a execução da sequência. Também, comumente utilizado em testes.

11. *consider*: Diferentemente do fragmento *ignore*, este fragmento indica que o fluxo de seu operando deve ser levado em consideração na sequência. O uso desse fragmento só faz sentido quando combinado com o fragmento *ignore*.

12. *assertion*: Este fragmento indica que o fluxo em seu operando representa uma asserção, ou seja, seu fluxo só possui sequências válidas. Esse fragmento também pode ser usado em conjunto com os fragmentos *ignore* e o *consider*.

3.2 Testes de *Design* Baseados em Regras de *Design*

Testes de *Design* são testes capazes de indicar quando a implementação de um software está de acordo com as definições elaboradas durante o seu projeto por meio dos seus modelos [BGF09]. Diferente dos testes funcionais, este tipo de teste não é capaz de indicar se um software está correto, mas se as funcionalidades deste software foram implementadas da maneira como foram projetadas. Portanto, este teste não pode ser usado para medir a corretude do sistema, sendo útil para manter a sincronia entre implementação e documentação, melhorando a evolução do sistema, e evitando que a documentação se torne obsoleta.

Para que os testes de *design* pudessem ser elaborados com maior clareza Brunet *et al.* [BGF09] criaram a definição de regras de *design*: restrições aplicadas ao software durante a fase de testes que determinam quais características estruturais devem ser seguidas pela implementação, tais como herança e associação entre classes. Uma importante característica das regras de *design* é que estas regras devem ser escritas na mesma linguagem de programação em que o software foi desenvolvido, o que, segundo os autores, reduz a curva de aprendizagem de elaboração e manutenção dos testes e ainda permite melhorar a comunicação entre projetistas e desenvolvedores.

Como exemplo de uma regra de *design* simples, digamos que exista uma classe A e que esta classe possui um campo b da classe B, e, além disso, nada que esteja na classe A pode ser acessado por B [BGF09]. Com este exemplo podemos testar se:

- A possui um campo b?
- b é do tipo B?
- B referencia A?

Para as duas primeiras perguntas devemos esperar que a implementação responda afirmativamente, e a terceira pergunta deve ser respondida com uma negação. Caso alguma dessas respostas seja divergente do esperado o teste indica que houve falha na implementação da regra específica a que se refere aquela parte do teste.

Nesta dissertação, nós ampliamos o conceito de regra de *design* para que atingisse não apenas características estruturais mas também características comportamentais do software. Tais características comportamentais possuem as mesmas necessidades de gerência no processo de evolução do software que as características estruturais.

3.3 *Design Wizard*

Para concretizar a aplicação de regras de *design*, Brunet et al. [DWS11] desenvolveram a API *Design Wizard*. Com esta API escrita em Java é possível submeter um software, também escrito em Java, aos testes de *design* que indicarão se existe correspondência do modelo estrutural com o software implementado.

Esta API possui um modelo de dados capaz de representar pacotes, classes, interfaces, campos, métodos e modificadores, bem como as relações entre esses tipos de entidades de software: a contém b, a herda de b, a implementa b, a instancia b, a é acessado por b, a é declarado em b, a é implementado por b, a chama b, a é chamado b e a é super classe de b. Usando esse conjunto de representações, a API *Design Wizard* monta um grafo onde as entidades são os vértices e as relações entre as entidades são as arestas. Nesse grafo as entidades são interligadas sem uma ordem específica capaz de indicar sequência de declaração ou execução destas entidades. Desta forma, a informação contida no grafo indica corretamente a relação entre as entidades que o compõem, mas não há garantia que haja ordenação da informação guardada.

Além de gerar o grafo de entidades e relações, o *Design Wizard* fornece os métodos de acesso necessários para buscar e recuperar informações contidas nos vértices e arestas

desse grafo. Desta maneira é possível para o projetista ou testador implementar cada uma das regras de *design* através de consultas à API e posterior comparação entre os resultados obtidos e esperados.

Assim sendo, para exemplificar o uso das consultas feitas através do *Design Wizard*, apresentamos na Tabela 3.1 exemplos de como devem ser escritas as regras para responder cada uma das três perguntas feitas 'no início desta seção. Para melhor compreensão, definimos a seguinte representação das informações contidas na tabela: A representa a classe A, B representa a classe B, e b representa um campo do tipo B. Além disso, é necessário saber que o método $getDeclaredFields()$ recupera o conjunto de campos declarados em uma classe, o método $getType()$ recupera a classe de uma entidade, e o método $getCallerClasses()$ recupera a lista de classes referenciadas por uma entidade.

Tabela 3.1: Regras de *Design versus* Consultas via *Design Wizard* .

Regras de *Design*	Consulta via DW
A possui um campo b	A.getDeclaredFields().contains(b)
b é do tipo B	b.getType().equals(B)
B referencia A	B.getCallerClasses().contains(A)

Como parte do trabalho de ampliação do conceito de regras de *design* nós ampliamos as capacidades da API *Design Wizard* para que permita a consulta às informações sobre a execução dos métodos escritos em Java. Para isto adicionamos um nova entidade que passou a representar os fragmentos combinados. Assim, dado um método m e um fragmento combinado fc, podemos encontrar as relações m contém fc, fc está contido em m, fc chama m e m é chamado por fc. Ainda é possível que tenhamos dois fragmentos combinados aninhados, e neste caso usamos os tipos de relação $fc1$ contém $fc2$ ou $fc2$ está contido em $fc1$. Além disso, o grafo foi modificado para permitir repetição e ordenação de conjuntos de vértices e arestas para os casos onde, por exemplo, um método é chamado em mais de um ponto da sequência. Para ilustrarmos parte da ampliação feita trazemos na Tabela 3.2 algumas das possíveis consultas utilizando a representação de fragmentos combinados usada acima, bem como a realização dessas consultas através da API *Design Wizard*.

A primeira regra de *design* da Tabela 3.2 recupera a lista de métodos e comandos contidos no corpo de um método m (usando $getSequenceOfMethodsAndCommands$) e verifica se o fragmento fc está nesta

Tabela 3.2: Regras de *Design versus* Consultas via *Design Wizard: com Ampliação da API* .

Regras de *Design*	Consulta via DW
m contém fc	m.getSequenceOfMethodsAndCommands().contains(fc)
fc chama m	fc.getCalleeMethods().contains(m)

listagem (usando $getSequenceOfMethodsAndCommands$) . Vale ressaltar que este mesmo método usado aqui ($getSequenceOfMethodsAndCommands()$) também pode ser usado pra listar os elementos contidos dentro de um fragmento combinado. Já a segunda linha da tabela mostra como podemos descobrir se um método m pertence a lista dos métodos chamados pelo fragmento fc usando a operação $getCalleeMethods()$. Com essa operação é possível conhecer a lista de métodos que são chamados dentro do fragmento combinado.

3.4 *Model Driven Architecture*

A abordagem *Model Driven Architecture* (MDA) é constituída de um conjunto de regras, ou diretrizes, de desenvolvimento de software cuja estrutura e organização tem como ponto chave a utilização de modelos de software para execução automática de transformações desses modelos em outros artefatos de software, incluindo aí o próprio código fonte [Gro11a] [MDA04] [KWB03].

A arquitetura da abordagem MDA está organizada em quatro níveis distintos de abstração, representando em cada nível um conjunto distinto de características com especificidades diferentes. Entre esses níveis, o menos abstrato é o código executável, que não será explicado aqui, os outros 3 níveis serão apresentados a seguir:

CIM. O nível *Computational Independent Model* (CIM), ou Modelo Independente de Computação, é o nível mais abstrato de modelagem, e é nele que devemos modelar o domínio da aplicação. Neste trabalho não chegaremos a esse nível de abstração.

PIM. O nível *Plataform Independent Model* (PIM), ou Modelo Independente de Plataforma, é usado para definir quais os requisitos de desenvolvimento para o software com mais detalhes que no CIM, sejam eles funcionais ou não, bem como seu *design* e sua arquitetura.

Nesse nível não é definido o que será usado pra desenvolver o software, mas já se sabe qual o escopo do software dentro do domínio da aplicação elaborado no CIM.

PSM. Por fim, o nível *Plataform Specific Model* (PSM), ou modelo específico de plataforma, é usado para representar os requisitos modelados no nível PIM para uma plataforma específica. É a partir do PSM que o desenvolvedor produzirá o código fonte do software.

Além da existência desses três níveis de abstração, em MDA existe um conjunto de mecanismos e linguagens de programação que são utilizados para realizar transformações que levam a representação de um software de um modelo a outro, ou de um modelo para texto, ou código, concreto.

No primeiro caso estamos falando de transformações de modelo a modelo. Com esse tipo de transformação podemos tanto modificar o nível de abstração empregado na representação do *software*, quanto modificar a linguagem em que um modelo está escrito sem alterar o nível de abstração adotado. Assim sendo, podemos ter transformações CIM-CIM, PIM-PIM, PSM-PSM, CIM-PIM, PIM-PSM, entre outras.

No segundo caso do emprego de linguagens de transformação de modelo a texto, em que as transformações são responsáveis por modificar representações no tocante ao nível de abstração PSM levando-o ao nível de concretização do código fonte.

Capítulo 4

Evolução do Design Wizard para Suporte a Testes de *Design* Comportamentais

Este capítulo aborda a elaboração e realização da evoluação da representação de entidades adotada pelo *Design Wizard* [DWS11] para possibilitar a analise de aspectos comportamentais contidos no código fonte do sistemas sob teste escritos em Java.

4.1 Evolução do Modelo de Dados

A representação dos dados adotada pelo *Design Wizard* [DWS11] permite consultar apenas informações estruturais a partir do sistema sob teste. Através desse modelo podemos representar não só as entidades existentes no sistema, como classes e métodos, mas também as relações entre essas entidades. Por exemplo, é possível saber quando um método pertence a uma classe ou interface. Entretanto, o modelo de dados existente não possui todas as características necessárias para suprir esta dissertação, pois precisamos de informações comportamentais, e o modelo original só nos fornece informações estruturais a respeito do sistema sob teste. Desta forma, se fez necessário expandir este modelo para que pudessemos recuperar informações comportamentais das quais precisamos.

A limitação apresentada pela API do *Design Wizard* em suas versões oficiais permite apenas a coleta de informações de natureza estrutural dos tipos: existência ou não de uma entidade, relações de posse entre entidades, relações de hereditariedade entre entidades, e descoberta de modificadores (público, privado, estático, etc.). Tendo em vista essa limita-

ção, desenvolvemos uma extensão para o *Design Wizard* permitindo-lhe extrair do sistema sob teste todas as informações relativas ao comportamento desse sistema. As informações adicionais extraídas através da extensão realizada correspondem à chamada de métodos e à utilização de comandos capazes de realizar desvios de fluxo de execução. Vale ressaltar ainda que, apenas alguns desvios de fluxo podem ser extraídos a partir da extensão realizada, e isto se deu devido à complexidade elevada dos demais desvios. Sendo assim, neste trabalho, os comandos extraídos são o comando condicional *if-else* e os comandos iterativos *for* e *while*.

Após conhecer o modelo de dados apresentado aqui, será possível entender melhor como as regras de *design* utilizadas nos testes propostos nesse trabalho são aplicadas. Isto se dá pois, uma vez que entendemos como esse modelo representa as entidades contidas nos diagramas UML e no código Java, entenderemos como construir as regras de *design* desejadas.

O modelo de dados adotado pode ser visto na Figura 4.1, onde podemos encontrar respectivamente a representação de dois pacotes os quais agrupam características similares em dois conjuntos de dados distintos. O primeiro, referente às entidades, é chamado de *design* e é composto por classes que refletem as características estruturais do software sob teste. O segundo pacote de classes - subpacote do primeiro, é chamado de *block* e representa os elementos relacionados a desvios de fluxo, os quais refletem características comportamentais do mesmo software sob teste. Cada uma das classes representadas em ambas as figuras serão apresentadas a seguir neste capítulo. Vale ressaltar que aqui apresentaremos apenas os campos de cada classe, pois cada classe é formada por campos e métodos de acesso (*getters & setters*), sendo redundante falar qual método acessa determinado campo. Este deve ser o entendimento padrão ao ler a respeito dessas classes, exceto em casos em que deva ser fornecido mais detalhes sobre métodos específicos.

4.2 Pacote *design*

O pacote *design* contém todas as classes da versão 1.4 do *Design Wizard*. As classes existentes neste pacote representam todas as entidades que, na versão oficial, correspondem às entidades da representação estrutural do *software* sob teste. No contexto da extensão construída para esta dissertação algumas das classes deste pacote passam a representar também aspec-

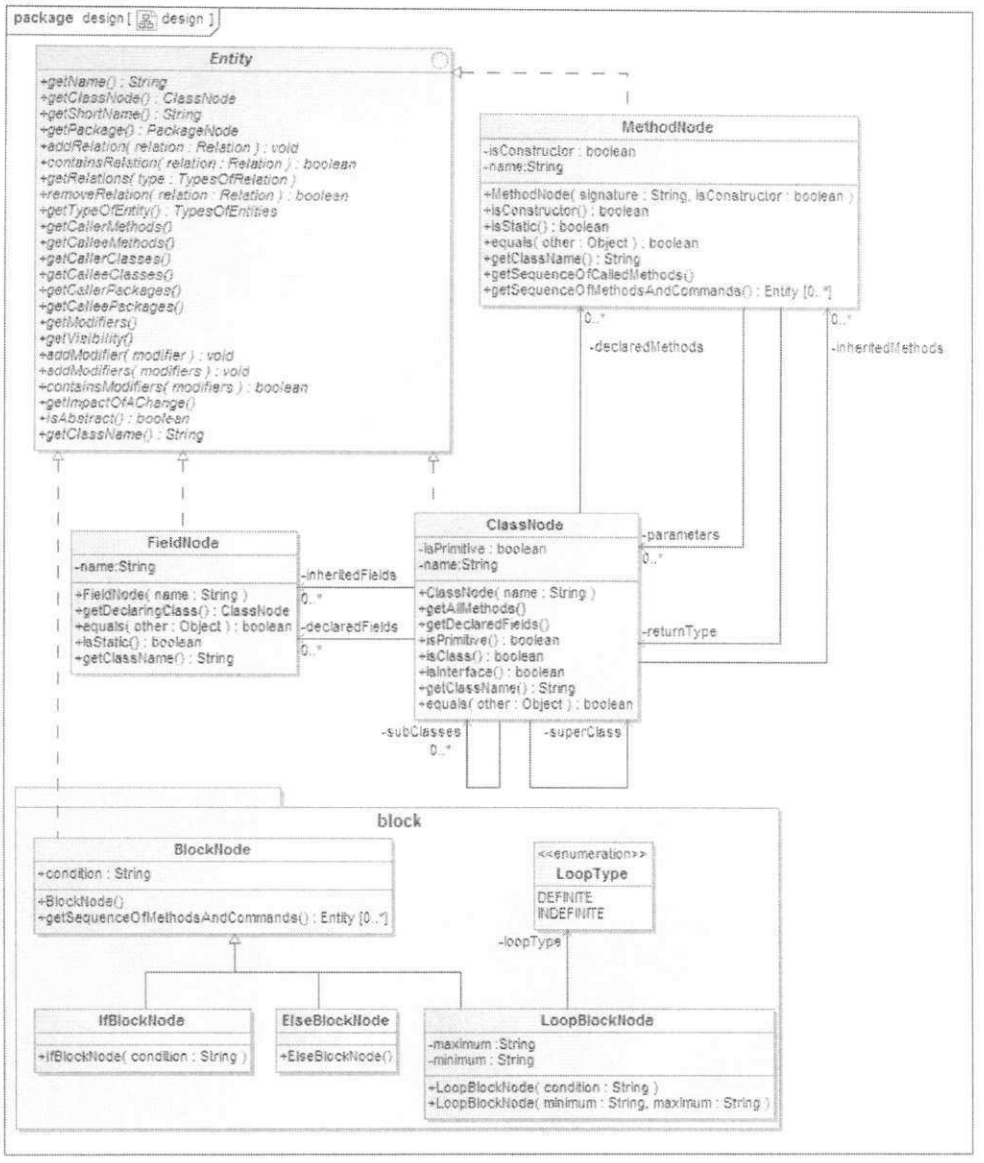


Figura 4.1: Modelo de dados do *Design Wizard* com extensão

tos comportamentais. Dentre as classes existentes neste pacote apenas *Entity*, *ClassNode* e *FieldNode* nos interessam, pois as outras tratam exclusivamente de informações estruturais. As demais classes não fazem parte do escopo deste trabalho diretamente, portanto só serão citadas quando houver necessidade.

4.2.1 Interface *Entity*

A interface *Entity* representa o conceito geral de entidade de software, ou seja, qualquer entidade que possa existir e ser indentificada em um sistema. Assim sendo, esta interface é usada para especificar os métodos usados por todos os tipos de representações pertencentes ao modelo de dados explicado aqui, capacidade esta que facilita a escrita dos testes de *design* propostos neste trabalho. Assim, em termos de representação, toda classe pertencente ao modelo é antes de tudo uma entidade, *i.e.*, implementa as funcionalidades da interface *Entity*.

Uma classe abstrata é usada pra especificar as funcionalidades da interface *Entity*, esta classe chama-se *AbstractEntity* que, anteriormente não foi exibida na Figura 4.1, mas que pode ser vista na Figura 4.2. Nesta Figura, as operações da interface *Entity* ficaram ocultas. Como podemos ver, a classe *AbstractEntity* contém os campos que são comuns a todas as representações de entidades de um *software*. Estes campos são acessados por herança pelas demais classes do pacote *design*. Isto permite que as implementações de parte das funcionalidades da interface *Entity* sejam realizadas de maneira uniforme entre as demais classes deste pacote. Por fim, elencamos a seguir os campos que formam a classe *AbstractEntity*:

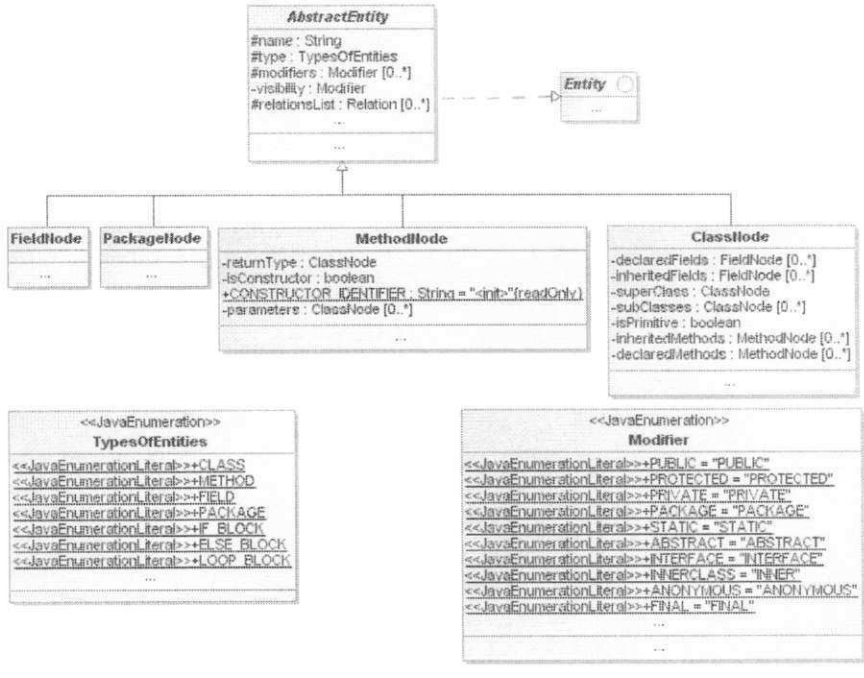


Figura 4.2: Diagrama de classe para os testes de *design*, contendo a classe *AbstractEntity*

- ***relationsList***: As relações entre uma entidade e as demais entidades do sistema. É

através dessa lista de relações contida em cada entidade que o *Design Wizard* consulta as ligações, ou relações, entre as entidades do software sob teste;

- ***name***: nome atribuído à entidade. Em alguns casos, a entidade não possui nome, como classes anônimas, por exemplo. Nesses casos esse campo fica vazio;
- ***type***: determina qual o tipo de uma entidade. Este campo é usado para facilitar algumas operações polimórficas envolvendo grande numero de entidades;
- ***modifiers***: representa os modificadores aplicados a cada entidade quanto a seu escopo (*static, final, etc.*);
- ***visibility***: representa o modificador especifico de visibilidade.

Classe *ClassNode*. Com a classe *ClassNode* representamos todas as interfaces e classes de um sistema, inclusive podendo distinguir a existência de classes abstratas. Nessa classe podemos obter informações sobre todos os membros que a compõem como campos, métodos, classes internas, enumerações, bem como as relações entre uma classe e as demais classes que compõem o sistema analisado.

Classe *FieldNode*. A representação de cada campo existente em uma classe é feita através da classe *FieldNode*. Com ela é possível saber o tipo do campo, o nome do campo, os pontos onde esse campo é acessado dentro do sistema e quais partes do sistema são acessadas por esse campo.

Classe *MethodNode*. Cada método do sistema é representado por uma instância da classe *MethodNode*. Através dessa classe é possível encontrar informações semelhantes às contidas na classe *FieldNode*, bem como informações sobre parâmetros do método representado.

4.3 Pacote *block*

O pacote *block*, apresentado na Figura 4.1, contém as classes usadas para representar os diferentes tipos de comandos possíveis em um bloco de execução, tanto em Java quanto em UML. Vale ressaltar aqui, que a forma como os comandos são construídos em Java possuem

características comuns com UML, o que permite representar comandos das duas linguagens com a mesma classe, ou com composições dessas classes. Por exemplo, para representar um *opt* em UML é necessário apenas um comando *if* em Java, portanto podemos usar duas instâncias da classe *IfBlockNode* uma para cada comando de cada linguagem. Outras composições podem ser realizadas usando as classes apresentadas nesta seção, de acordo com a necessidade de representação de comandos.

Classe *BlockNode*. Mesmo um comando não sendo uma entidade em si, as representações de comandos feitas através da classe *BlockNode* possuem características em comum com a representação feita pela interface *Entity*. Assim sendo, para fins de modelagem, todo bloco de comandos também é percebido como uma entidade, exceto pelo fato de não possuir nome ou mesmo visibilidade. Entretanto, os comandos precisam ser identificados pra que a sequência de execução seja interpretada corretamente. Para que essa identificação ocorra usamos as seguintes características dos comandos: a condição de execução ou parada, quando houver; o tipo do comando e a posição do comando na sequência de execução. Onde as duas primeiras características são herdadas pelas demais classes do pacote *block*. Por outro lado, a posição do comando só é conhecida quando o mesmo estiver presente em uma sequência, *i.e.*, a posição do comando corresponde a sua localização na sequência de comandos da qual faz parte.

Classe *IfBlockNode*. Comandos condicionais são representados pela classe *IfBlockNode*. Através dessa classe podemos representar tanto comandos condicionais em Java, *i.e.*, comandos *if*, quanto os fragmentos combinados *opt* e *alt* de UML. No primeiro caso o fragmento *opt*, é representado por apenas um *IfBlockNode*, da mesma forma que um comando *if* de Java. Já para o caso do fragmento *alt* o uso da classe *IfBlockNode* é mais elaborado. Neste caso temos uma sequência de caminhos de execução possíveis, onde cada caminho necessita que uma condição diferente das condições usadas em outros caminhos seja considerada verdadeira para que o caminho seja executado. Sendo assim, combinamos um sequência de objetos *IfBlockNode*, um para cada condição a ser testada, finalizando com um objeto *ElseBlockNode*, usado no último caminho de execução possível quando nenhuma outra condição é considerada verdadeira. A mesma representação é usada para comandos Java quando temos

uma sequência de comandos *if*, finalizada por um comando *else*. No Capítulo 5 poderemos entender melhor a aplicação dessa sequência de comandos representando os comandos condicionais em Java e sua correlação com os fragmentos *opt* e *alt* da UML

Classe *ElseBlockNode*. A classe *ElseBlockNode* é usada para representar o último trecho de execução de fragmentos *alt* de UML, e também o comando *else* da Java. Essa classe distingue-se da classe *IfBlockNode*, por representar comandos que não possuem condições que permitam sua execução, e que só são executados quando todas as demais condições não são aceitas. Assim sendo, esta classe herda todas as características da classe *BlockNode*. Mesmo não utilizando o campo *condition* e não sendo esta uma prática de modelagem orientada a objetos correta, mantivemos o modelo de dados como se apresenta, e assim facilitamos o entendimento e a execução dos testes de *design* propostos.

Classe *LoopBlockNode*. A representação de comandos iterativos é feita através da classe *LoopBlockNode*. Com ela podemos representar iterações controladas por um contador e iterações cujo controle é feito com base em condições de execução. Em UML, o fragmento que define essas iterações é o fragmento *loop*, já em Java podemos usar dois comandos distintos, o *for* e o *while*, os quais por simplificação funcionam da mesma maneira, pois a distinção entre eles é apenas sintática, *i.e.*, apenas a forma como o comando é escrito.

Enumeração *LoopType*. Na UML, quando um *loop* tem sua execução determinada por um limite inferior e superior para delimitar a execução de iterações, este é chamado de *loop* definido, pois a quantidade de vezes que é executado é conhecida. Já quando o *loop* possui uma condição de execução, este deve ser chamado de *loop* indefinido, pois não se sabe quantas vezes será executado. Assim sendo, seguindo a mesma abordagem da UML, usamos a enumeração *LoopType* para simbolizar qual tipo de *loop* (*DEFINITE* e *INDEFINITE*) estamos representando em determinado ponto da sequência.

Capítulo 5

Templates para Testes de *Design* Comportamentais

Testes de *design* são testes capazes de verificar se um *software* teve seu projeto seguido durante seu desenvolvimento. Para que sejam montados os testes de design fazemos uso do conceito de regra de *design*, *i.e.*, restrições impostas ao sistema para definir o que deve ou não ser desenvolvido.

A princípio, tanto as regras quanto os testes de *design* foram elaborados com o intuito de checar apenas informações estruturais dos sistemas sob teste. Entretanto, como parte das contribuições desta dissertação, desenvolvemos mecanismos e padrões que passaram a permitir a escrita e a execução dos testes de *design* para verificar aspectos comportamentais do sistema sob teste. A escrita dos testes de *design* comportamentais adota as mesmas ferramentas de suporte utilizadas pelos testes estruturais. Essas ferramentas são:

- JUnit, framework de testes usado comumnte para testes funcionais, que nos permite escrever os testes de *desin* diretamente em Java.
- Design Wizard, também escrito em Java, cuja extensão desenvolvida nesta dissertação permite representar aspectos comportamentais do sistema sob teste.

Os testes responsáveis pela checagem de conformidade entre diagramas de sequência e código Java são escritos através de códigos Java. Esta linguagem foi escolhida devido à familiaridade dos desenvolvedores que já a utilizam no cotidiano, não havendo, assim, a

necessidade de aprender uma nova linguagem para escrever os testes. Além disto, utilizamos a API JUnit [JUn11], a qual é capaz de executar testes automaticamente e indicar quando o teste falha ou obtém sucesso. Outra API muito importante para a escrita dos testes é a *Design Wizard*, que permite consultar informações a respeito de softwares analisando diretamente o código fonte.

Os testes de *design* comportamentais desenvolvidos nesta dissertação foram elaborados sob forma de *templates*, *i.e.*, padrões de aplicação de regras de *design*. Os *templates* de testes de *design* foram construídos para comparar informações obtidas a partir de diagramas de sequência e informações do cóigo fonte do sistema. Estes *emplates* estão organizados em três classes Java, construídas com base no framework de testes JUnit. A primeira delas é uma interface (*ISupportVerification*) que define contratualmente os testes para mensagens e fragmentos combinados. Assim sendo, o método *testCombinedFragment(IfBlockNode, IfBlockNode)*, usado para checar fragmentos combinados do tipo *opt*. Desta maneira, cada método desta interface corresponde a um teste que deve ser executado em conjunto com os demais para realizar a checagem do sistema.

A segunda classe (*SupportToVerificationImpl*) é composta pela implementação de todos os métodos definidos pela interface *ISupportVerification*, bem como por métodos auxiliares, como por exemplo o método *checkType(Entity, Entity)* o qual compara o tipo de duas entidades. Vale ressaltar que uma classe abstrata Java pode implementar comportamentos comuns às classes que irão herdá-la posteriormente, e que, caso seja necessário, essas classes podem sobrescrever ou complementar esse comportamento.

Por último, para cada diagrama de sequência deve ser gerado uma classe de teste, e esta deve herdar o comportamento da classe *SupportToVerificationImpl*, visto que deve executar os testes implementados nesta classe, bem com implementar os métodos que dão inicio à execução dos testes de *design*.

A seguir, serão descritos em detalhes cada uma das classes e seus respectivos métodos. A Figura 5.1 apresenta o diagrama de classe que permite visualizar o relacionamento entre essas classes.

A interface *ISupportToVerification* e a classe abstrata *SupportToVerificationImpl* são a base dos testes de *design* propostos nesta dissertação. A execução dos testes contidos na classe abstrata é padrão para todos os testes e segue contratualmente o que é definido pela

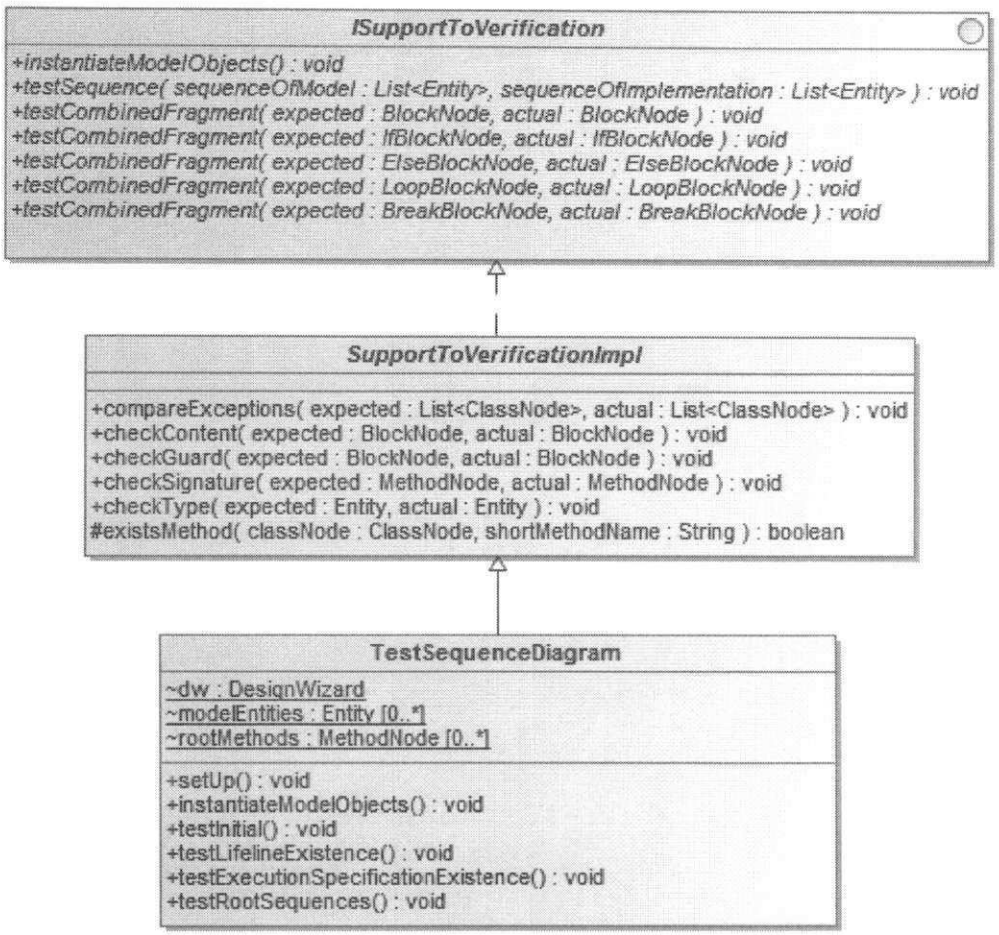


Figura 5.1: Diagrama de classe para os testes de *design*

interface. Assim sendo, permitimos que os métodos colocados nessas duas classes sejam reusados por todos os testes aplicados a qualquer diagrama de sequência. Dessa forma, evitamos a reescrita desses métodos, mas ainda assim permitimos que outras implementações sejam construídas, desde que sigam a interface proposta.

Como veremos a seguir, para todo diagrama de sequência, devemos criar uma classe de teste, assim sempre teremos um teste para cada diagrama de sequência. Além disso, a classe criada deve conter outros métodos usados em toda a checagem, mas que não poderiam ser colocados na classe *SupportToVerificationImpl* devido à uma limitação relativa à execução dos testes através da API Junit, a qual exige que a classe de teste possua pelo menos um método declarado diretamente em seu corpo. Assim sendo, para usarmos esta API, pelo menos um dos métodos de teste deve ser declarado diretamente na classe de teste. Uma vez que esses métodos são obrigatórios, de acordo com o framework JUnit, definimos arquitetural-

mente que apenas os métodos que iniciam os testes estarão nas classes de teste. Os outros métodos (comuns a todos os testes) ficam na interface *ISupportToVerification* e na classe abstrata *SupportToVerificationImpl*. Um exemplo de classe de teste é apresentado na Figura 5.1, sob o nome *TestSequenceDiagram*.

É importante ressaltar que os testes apresentados aqui utilizam a API do *Design Wizard* para consultar as informações existentes no sistema sob teste. Portanto, sempre que nos referirmos à consulta ao sistema, estaremos nos referindo ao uso *Design Wizard* com essa finalidade de consulta, e a sua extensão realizada nesta dissertação.

A seguir, neste capítulo, aprofundaremos as explicações a respeito das classes envolvidas na execução dos testes de *design* com detalhamentos dos testes de cada uma das entidades do diagrama de sequência abordadas nesta dissertação. Além disso, apresentamos um estudo de caso demonstrando a preparação e execução de um teste simples utilisando nossa abordagem. Por fim, apresentamos a arquitetura contruída para a ferramenta desenvolvida durante este trabalho de mestrado.

5.1 Classe de Teste para Diagramas de Sequência

Nessa seção apresentaremos os métodos que compõem a classe de teste de *design* para os diagramas de sequência. Cada diagrama de sequência deve ter uma classe equivalente à essa para execução dos testes de *design*. A princípio essa classe é nomeada como *Test<SequenceDiagram>*, onde *<SequenceDiagram>* é substituído pelo nome do diagrama de sequência sob teste. Na Figura 5.1, essa classe é apresentada com o nome *TestSequenceDiagram* para fins de simplificação. Cada método dessa classe é responsável pela checagem de um dos seguintes elementos de um diagrama de sequência: *Lifeline*, representando os objetos envolvidos na sequência do diagrama (*testLifelineExistence()*); e *ExecutionSpecification*, representando os métodos contidos no diagrama de sequência (*testExecutionSpecificationExistence()*). Além disso, esta classe também deve possuir o método que inicia a checagem da sequência contida no diagrama, chamado *testRootSequence()*. Vale ressaltar que as demais entidades (mensagens e fragmentos combinados) são testadas por meio dos métodos herdados da classe *SupportToVerificationImpl*.

Conforme podemos ver na Figura 5.1, existem três campos nessa classe (*dw*, *rootMethods*

e *modelEntities*) e três métodos adicionais: (*setUp(), instantiateModelObjects() e testInitial()*). Os campos representam:

- ***dw*** Objeto da API *Design Wizard* usado para realizar consultas ao sistema sob teste;
- ***rootMethods***: representa os métodos que iniciam sequências dentro de um diagrama. Isso ocorre, pois, apesar de não ser uma boa prática de modelagem, um mesmo diagrama de sequência pode conter mais de uma sequência;
- ***modelEntities***: representa todos os objetos e classes envolvidas na descrição do diagrama de sequência.

Já os métodos auxiliares possuem as seguintes funcionalidades:

- ***setUp()***: método usado para instanciar todos os campos da classe de teste (*TestSequenceDiagram*).
- ***instantiateModelObjects()***: método usado para popular o teste com as informações colhidas no diagrama de sequência. Esse método cria os objetos conforme o modelo apresentado na Seção 4.1 - e complementado na Seção 5.1.1, apresentada posteriormente - com base nos dados colhidos do diagrama de sequência. Por exemplo, podemos criar um objeto do tipo *ClassNode* para representar a classe de um objeto. Uma vez criado o modelo, os dados que representam o diagrama podem ser comparados com as informações contidas no código do sistema sob teste em Java.
- ***testInitial()***: método usado apenas para apoio à coleta resultados, sua função não é de teste, mas de separador entre as marcações de tempo usado para intanciar os dados dos testes e do tempo de execução do teste em si. Isto se dá pois, ao executarmos quaisquer testes através da API JUnit encontramos um problema quanto a coleta de dados referentes ao tempo de execução do primeiro método da classe de teste. O problema ocorre devido ao tempo gasto pelo método *setUp()* para instanciar o grafo de entidades e relações do *software* sob teste. Como este tempo é bastante alto se comparado à execução de todo o conjunto de teste, usá-lo nas medições influi na análise dos dados coletados, por isto decidimos usar esse artifício para separar tempo de instanciação do tempo de teste em si.

Os três métodos apresentados acima são executados na preparação dos testes, já os métodos de testes propriamente ditos são executados logo em seguida, obedecendo à sequência: (i) *testLifelineExistence*, (ii) *testExecutionSpecificationExistence* e (iii) *testRootSequence*. Uma vez que a execução acontece dessa forma, checamos primeiramente a existência dos elementos que fazem parte do diagrama, para só então checarmos a sequência que relaciona esses elementos. Estes três métodos serão apresentados com mais detalhes a seguir.

5.1.1 Preparação dos Informações Utilizadas pelos Templates

Como forma de preparar o teste de *design*, os templates desenvolvidos necessitam de informações que são confrontadas durante a execução dos testes. O método *instantiateModelObjects()* é responsável por instanciar os objetos que representam as entidades do sistema sob teste bem como as relações que ligam estas entidades pelo diagramas de sequência UML. Desta forma, a representação extraída a partir dos modelos UML será comparada com as informações coletadas via *Design Wizard*, a partir do código Java, sempre que os testes forem executados. Uma vez que os dois conjuntos de objetos sejam equivalentes podemos dizer que o *software* passou nos testes propostos.

A Figura 5.2 apresenta um diagrama de classes diferente do apresentado na Seção 4.1, pois aqui temos a visão mais concreta de como os objetos de cada classe do *Design Wizard* se relacionam na alimentação dos testes. Neste diagrama temos a classe de teste *Test<SequenceDiagram>*, com seus respectivos campos *dw*, *rootMethods*, *modelEntities*. Podemos perceber que toda entidade relaciona-se com outra através de um objeto da classe *Relation*, assim cada entidade é ou uma *caller* (entidade fonte da relação) ou uma *called* (entidade destino da relação).

Como exemplo de relação entre entidades temos que, se um método *m()* pertence a uma classe *C*, então dizemos que temos relação do tipo *contains* onde C é *caller* e *m()* é *called*. Com este exemplo, podemos perceber que ao aplicarmos o mesmo processo de definição de relações entre entidades obtemos um grafo representando todo o sistema. Para os testes de *design* propostos, temos que o grafo de entidades construído a partir do diagrama de sequência UML deve corresponder a uma parte do grafo montado a partir do sistema sob teste. Uma ilustração desse processo pode ser visto na Figura 5.3, onde o grafo com nós quadrados é comparado com o grafo com nós circulares. Os elementos vindos do diagrama

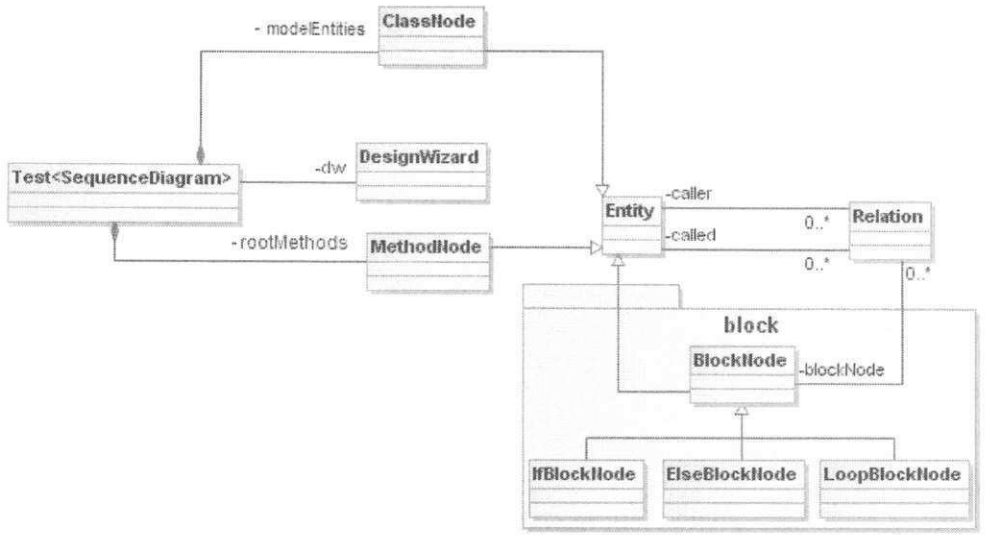


Figura 5.2: Modelo de dados dos testes de *design*

de sequência (quadrados), devem combinar com os elementos contidos no sistema (círculos), e é através dos testes de *design* propostos que essa comparação acontece. Por fim, o resultado da comparação corresponde aos nós do grafo de círculos demarcados pela área tracejada.

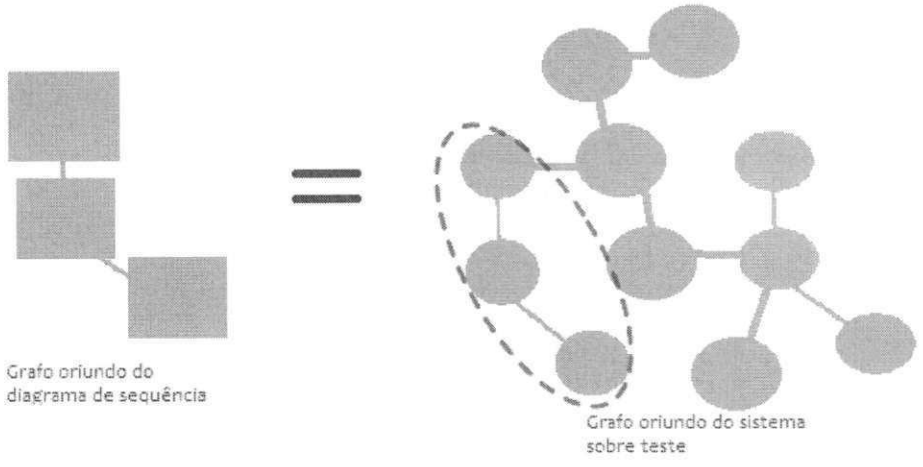


Figura 5.3: Comparação entre dois grafos de entidades e relações

A introdução do pacote *block* fica evidenciada na Figura 5.2, pois o mesmo adiciona novas entidades ao modelo de dados. Entretanto, suprimimos a existência das demais classes do pacote *design* pois apenas a classe *Entity* é importante na construção das relações, e as

demais classes conseguem a mesma característica por herança. Outra informação suprimida na Figura 5.2 é relativa à classe *DesignWizard* a qual possui todas as referências para todos as entidades do sistema sob teste, mas que aqui não está ligada à classe *Entity*. Entretanto essa ligação é importante uma vez que temos acesso a todo o sistema sob teste através de uma instância dessa classe. Assim, usamos o campo *dw* da classe *Test<SequenceDiagram>* para coletar informações e assim compará-la com o modelo contido nos campos *rootMethods* e *modelEntities* de acordo com as regras definidas nos métodos de teste especificados a seguir neste capítulo.

5.1.2 *Testando a Existência de Objetos*

O primeiro elemento do diagrama de sequência a ser analisado é o *lifeline*. Cada *lifeline* representa uma classe ou objeto que faz parte da interação. Portanto devemos verificar se tanto a classe quanto o objeto (dependendo da situação) existem no sistema. A Figura 5.4 apresenta o método testLifelineExistence() criado para a checagem deste elemento. O funcionamento desse teste é composto pelos seguintes passos:

1. Iterar por todos os lifelines existentes (*linha 3*);
2. Identificar o tipo que cada lifeline representa, se é um objeto (*TypesOfEntities.FIELD*) (*linha 5*) ou uma classe (*linha 9*);
3. Caso seja um objeto, verificamos se o nome desse objeto está presente no sistema sob teste (*linhas 5-8*);
4. Caso seja uma classe verificamos se a classe está presente no sistema consultando seu nome (*linhas 10-11*).

5.1.3 *Testando as Execution Specification*

Seguindo a sequência definida para a execução dos testes, o próximo elemento a ser checado é o *ExecutionSpecification*. Este elemento indica o escopo de um método dentro da sequência, *i.e.*, o conjunto de chamadas a outros métodos, de blocos de comandos, e de objetos que compõem o trecho de execução coberto pelo método.

```
1  @org.junit.Test
2  public void testLifelineExistence() {
3      for (org.designwizard.design.Entity entityInModel : modelEntities) {
4          try {
5              if (entityInModel.getTypeOfEntity() == org.designwizard.design.Entity.TypesOfEntities.FIELD) {
6                  org.junit.Assert.assertTrue("Inexistent object " + entityInModel.getName(),
7                          dw.getField(((FieldNode) entityInModel).getName()) != null);
8
9              } else {
10                 org.junit.Assert.assertTrue("Inexistent class " + entityInModel.getClassName(),
11                         dw.getClass(entityInModel.getClassName()) != null);
12             }
13         } catch (org.designwizard.exception.InexistentEntityException e) {
14             org.junit.Assert.fail("Inexistent class/object "
15                     + entityInModel.getName());
16         }
17     }
18 }
```

Figura 5.4: Teste de *design* para *Lifelines*

O teste que checa a existência de *ExecutionSpecification* deve executar os seguintes passos, os quais podem ser observados na Figura 5.5:

1. Analisar todos os lifelines (objetos e classes; *linha 4*);
2. Recuperar a classe para o caso de objetos (*linhas 7-8*);
3. Recuperar todos os métodos chamados no lifeline analisado (*linhas 12-15*);
4. Analisar cada método para saber se ele existe no sistema sob teste (*linhas 17-22*).

Já na linha 20 da Figura 5.5, encontramos a chamada para o método *existsMethod()*. Este método faz parte da classe *SupportToVerificationImpl*, mas será apresentado aqui por ser chamado apenas nesse ponto do teste. Como podemos ver na Figura 5.6 (*linha 4*), esse método é usado para saber se um método descrito no modelo (representado pelo parâmetro *shortMethodName*) existe na classe implementada (representada pelo parâmetro *classNode*).

5.1.4 Testando as Sequências de Mensagens

Uma vez que sabemos que objetos, classes e métodos que devem fazer parte da sequência descrita no diagrama existem no sistema sob teste, podemos dar início à análise da sequência propriamente dita. Para tal, checamos a primeira mensagem trocada entre os lifelines. O campo *rootMethods* da classe *Test<SequenceDiagram>* contem a mensagem que inicia a sequência sob teste. Este campo pode possuir mais de um método, visto que um diagrama pode ter mais de uma sequência independente. Esta não é uma prática recomendada para a modelagem de um diagrama desse tipo, entretanto é permitida, e dada a facilidade de

```
1  @Test
2  public void testExecutionSpecificationExistence()
3          throws InexistentEntityException {
4      for (Entity entityInModel : modelEntities) {
5          ClassNode classNode = null;
6          if (entityInModel.getTypeOfEntity() == TypesOfEntities.FIELD) {
7              classNode = dw.getClass(((FieldNode) entityInModel).getType()
8                      .getClassName());
9          } else
10             classNode = dw.getClass(entityInModel.getClassName());
11
12         Set<MethodNode> methodsInModel = entityInModel.getClassNode()
13                 .getAllMethods();
14         methodsInModel.addAll(entityInModel.getClassNode()
15                 .getInheritedMethods());
16
17         for (MethodNode methodInModel : methodsInModel) {
18             Assert.assertTrue("Inexistent method \""
19                     + methodInModel.getShortName() + "\" in class \""
20                     + classNode.getClassName() + "\"", existsMethod(
21                     classNode, methodInModel.getShortName()));
22         }
23     }
24 }
```

Figura 5.5: Teste de *design* relativo à existência de *Execution Specification*

```
1  protected final boolean existsMethod(ClassNode classNode,
2          String shortMethodName) {
3      try {
4          classNode.getMethod(shortMethodName);
5          return true;
6      } catch (InexistentEntityException e) {
7          System.err.println(e.getMessage());
8          return false;
9      }
10 }
```

Figura 5.6: Teste de *design* relativo à existência de métodos em classes

interpretar esse tipo de situação, nós a consideramos como uma possibilidade válida. Sendo assim, uma vez que sabemos qual método será analisado, podemos executar os seguintes passos, codificados na Figura 5.7:

1. Consultamos o sistema sob teste para recuperar o método (representante da mensagem) da sequência testada (*linhas 5-7*). Na linha 5 encontramos a variável $rootMethod$ que representa o método contido no diagrama de sequência e, na linha 6, encontramos a variável $methodNode$ representando o método contido no código em Java; e

2. Passamos a analisar a sequência de mensagens usando o método *testSequence()*. O método *testSequence* será detalhado na próxima seção, pois faz parte da interface *ISup-*

potToVerification, e por consequência, da classe *SupportToVerificationImpl*.

```
1  @Test
2  public void testRootSequences() throws InexistentEntityException {
3      int numberOfRootMethods = rootMethods.size();
4      for (int i = 0; i < numberOfRootMethods; i++) {
5          MethodNode rootMethod = rootMethods.get(i);
6          MethodNode methodNode = dw.getClass(rootMethod.getClassName())
7                  .getMethod(rootMethod.getShortName());
8
9          testSequence(rootMethod.getSequenceOfMethodsAndCommands(),
10                 methodNode.getSequenceOfMethodsAndCommands());
11     }
12 }
```

Figura 5.7: Teste de *design* do método que inicia a sequência sob teste

5.2 Classe SupportToVerificationImpl

A classe *SupportToVerificationImpl* contém todas as funcionalidades necessárias para auxiliar a classe de teste de *design* na checagem de conformidade do diagrama de sequência. Tais funcionalidades são capazes de analisar a maioria dos elementos de uma sequência, sejam eles mensagens ou fragmentos combinados. Entretanto, vale ressaltar que o elemento *lifeline* já é checado na classe de teste (Subseção 5.1), bem como a existência de todos os métodos envolvidos na sequência. Além disso, alguns fragmentos combinados não possuem uma funcionalidade responsável por sua checagem, como por exemplo, os fragmentos *critical* e *par*. Esses elementos não possuem correspondência direta entre UML e Java, pois a linguagem Java não possui instruções nativas para paralelismo e tratamento de regiões críticas. Assim, apesar de ser possível executar essas funcionalidades em Java por meio de API específicas, entendemos que é suficiente analisarmos a sequência contida nesses fragmentos. Dessa forma, mesmo não tendo a garantia do paralelismo, analisaremos a sequência contida no fragmento combinado. Portanto, esses fragmentos são interpretados como sequências comuns de execução.

Uma vez que a classe *SupportToVerificationImpl* implementa as funcionalidades definidas na interface *ISupportToVerification*, apenas explicaremos os métodos propriamente ditos, ficando subentendido que os mesmos fazem parte tanto da interface quanto da implementação. Vale ressaltar ainda que todos os fragmentos combinados são analisados por

métodos com mesmo nome (*testCombindedFragment()*). Portanto, existe sobrecarga entre eles, variando-se apenas o tipo dos parâmetros de cada um de acordo com o fragmento testado. Com isso é possível, em trabalhos futuros, padronizar a criação de novos métodos para fragmentos não checados até o momento, a exemplo do *break*.

5.2.1 Checagem de Sequência de Mensagens

A Figura 5.8 apresenta o método *testSequence()*, o qual é responsável pela checagem de uma sequência onde existam exclusivamente métodos sendo chamados. O seguinte procedimento deve ser executado para a checagem:

```
1  public final void testSequence(List<Entity> sequenceOfModel,
2          List<Entity> sequenceOfImplementation) {
3
4      int size = sequenceOfModel.size();
5      for (int i = 0; i < size; i++) {
6          Entity expected = sequenceOfModel.get(i);
7          Entity actual = sequenceOfImplementation.get(i);
8          checkType(expected, actual);
9
10         if (expected.getTypeOfEntity() == TypesOfEntities.METHOD) {
11             checkSignature((MethodNode) expected, (MethodNode) actual);
12
13             testSequence(((MethodNode) expected)
14                     .getSequenceOfMethodsAndCommands(),
15                     ((MethodNode) actual).getSequenceOfMethodsAndCommands());
16         } else if (expected instanceof BlockNode) {
17             testCombinedFragment((BlockNode) expected, (BlockNode) actual);
18         }
19     }
20 }
```

Figura 5.8: Teste de *design* da sequência de mensagens

1. Analisamos cada um dos elementos do diagrama de sequência e do código Java (*linhas 4-7*);

2. Comparamos o tipo do elemento para saber se ambos são métodos ou desvios de fluxos (fragmentos combinados *versus* comandos de desvio de fluxo) (*linha 8*);

3. Caso se trate de um método, checamos se as assinaturas dos métodos correspondem entre si (*linha 11*);

4. Ainda no caso de ser um método, verificamos recursivamente possíveis sequências internas definidas para o método em questão (*linhas 13-15*);

5. Caso seja um fragmento combinado, delegamos a checagem desse fragmento para outro método de teste chamado testCombinedFragment() (*linha 17*).

Complementando o que vimos na Figura 5.8, as Figuras 5.9 e 5.10 apresentam os métodos auxiliares para o *testSequence()* os quais são respectivamente, o método para comparação de tipos de elementos (*checkType()*) e o método para comparação de assinaturas de mensagens (*checkSignature()*).

```
1 private final void checkType(Entity expected, Entity actual) {
2     Assert.assertEquals("Type of enntities are incompatibles ", expected
3             .getTypeOfEntity(), actual.getTypeOfEntity());
4 }
```

Figura 5.9: Teste de *design* para comparação de tipos de entidades

```
1 private final void checkSignature(MethodNode expected, MethodNode actual) {
2     Assert.assertEquals("Method names are imcompatibles ", expected
3             .getShortName(), actual.getShortName());
4     Assert.assertEquals("Classes of methods are imcompatibles ", expected
5             .getClassName(), actual.getClassName());
6 }
```

Figura 5.10: Método para comparação de assinatura de métodos

5.2.2 Checagem de Fragmentos Combinados Condicionais

Existem dois tipos de fragmentos usados para desvios puramente condicionados: o *opt* e o *alt*. A diferença entre eles é que o *opt* só possui um desvio possível, enquanto o *alt* permite vários. O método *testCombinedFragment(IfBlockNode, IfBlockNode)* deve ser chamado para cada condição a ser checada em um dos dois tipos de fragmento. A Figura 5.11 apresenta este método, que nada mais é que uma checagem do conteúdo do fragmento, *i.e.*, condição e sequência interna através do método *checkContent()*. A Figura 5.12 apresenta o método usado para verificar a ultima condição existente em um fragmento *alt*, a condição *else*. Essa última, apesar de não ser uma condição propriamente dita, indica a ultima sequência que deve ser executada caso as demais condições do fragmento *alt* não sejam interpretadas como verdadeiras. Neste caso, precisamos apenas verificar a sequência interna do fragmento. Os atributos *expected* e *actual* correspondem, respectivamente, à informação vinda do diagrama

```
1 public final void testCombinedFragment(IfBlockNode expected,
2         IfBlockNode actual) {
3     checkContent(expected, actual);
4 }
```

Figura 5.11: Teste de *design* para desvios condicionais

```
1 public final void testCombinedFragment(ElseBlockNode expected,
2         ElseBlockNode actual) {
3     testSequence(expected.getSequenceOfMethodsAndCommands(), actual
4             .getSequenceOfMethodsAndCommands());
5 }
```

Figura 5.12: Teste de *design* do operando *else*

de sequência e à informação vinda do sistema sob teste. A mesma nomenclatura é adotada com o mesmo sentido em todos os métodos da interface *ISupportToVerification*.

A Figura 5.13 apresenta o método *checkContent()* o qual é responsável por comparar duas informações: *(1)* a condição de execução de um fragmento usando o método *checkGuard()* (*linha 2*); e *(2)* a sequência interna do fragmento combinado (*linhas 4-5*). Este último passo caracteriza um processo de recursão na análise dos elementos do diagrama de sequência. Tal processo é necessário, dado que é possível compor sequências tanto colocando os elementos um após o outro, quanto os aninhando. Na Figura 5.14 apresentamos o método *checkGuard()* usado para comparar as condições de fragmentos combinados e de comandos Java. Vale ressaltar que a checagem é feita comparando diretamente a *String* que representa a guarda e a *String* da condição contida no código fonte, portanto a comparação é feita lexicamente.

```
1 private final void checkContent(BlockNode expected, BlockNode actual) {
2     checkGuard(expected, actual);
3
4     testSequence(expected.getSequenceOfMethodsAndCommands(), actual
5             .getSequenceOfMethodsAndCommands());
6 }
```

Figura 5.13: Teste de *design* do conteúdo de um fragmento combinado

5.2.3 Checagem de Fragmentos Combinados Iterativos

Os desvios iterativos modelados através de fragmentos combinados têm duas possíveis formas modeladas no fragmento *loop*: A primeira indicando a quantidade de vezes que a

```
1 private final void checkGuard(BlockNode expected, BlockNode actual) {
2     Assert.assertEquals("Conditions are incompatibles ", expected
3             .getCondition(), actual.getCondition());
4 }
```

Figura 5.14: Teste de *design* de uma condição de um operando

sequência deve ser executada (*loop* definido), e a segunda usando uma condição de parada para a iteração (*loop* indefinido). No primeiro caso temos um valor de máximo e um valor de mínimo, enquanto no segundo caso uma expressão booleana que deve ser checada. Na Figura 5.15, podemos visualizar o procedimento de checagem para desvios iterativos que segue os seguintes passos:

1. Verificamos se os comandos (*loop*) são do mesmo tipo (definido ou indefinido) (*linhas 3-4*);

2. Sendo indefinido, checamos a guarda do fragmento usando o método *checkGuard()* (*linha 7*);

3. Sendo um *loop* definido, checamos se os valores de máximo e mínimo estão de acordo com o que foi modelado no diagrama de sequência (*linhas 9-12*);

4. Por fim, checamos a sequência contida no fragmento combinado usando recursão novamente (*linha 15-16*).

```
1  public final void testCombinedFragment(LoopBlockNode expected,
2          LoopBlockNode actual) {
3      Assert.assertEquals("Type of loop are incompatible", expected
4              .getLoopType(), actual.getLoopType());// loop type
5
6      if (expected.getLoopType() == LoopBlockNode.LoopType.INDEFINITE) {
7          checkGuard(expected, actual);
8      } else if (expected.getLoopType() == LoopBlockNode.LoopType.DEFINITE) {
9          Assert.assertEquals("Mininum limits are incompatible", expected
10                 .getMinimum(), actual.getMinimum());
11         Assert.assertEquals("Maximum limits are incompatible", expected
12                 .getMaximum(), actual.getMaximum());
13     }
14
15     testSequence(expected.getSequenceOfMethodsAndCommands(), actual
16             .getSequenceOfMethodsAndCommands());
17 }
```

Figura 5.15: Teste de *design* de desvios iterativos

5.3 Exemplo Ilustrativo

Para demonstrar a criação dos testes de *design* usando a técnica proposta nesse trabalho, foi desenvolvido um exemplo ilustrativo simples, mas que contempla todos os elementos envolvidos nos testes de *design* propostos neste capítulo. Este exemplo apresenta o passo a passo para a criação do código de teste a partir de um diagrama de sequência, bem como o resultado da execução do teste gerado.

Passo 1. Para desenvolver este exemplo de caso utilizamos o diagrama de sequência apresentado na Figura 5.16, no qual podemos encontrar três objetos (b do tipo $Board$ e $d1$ e $d2$ do tipo $Dice$), dois métodos ($rollDices()$ para objetos do tipo $Board$ e $roll()$ para objetos do tipo $Dice$), e um fragmento combinado do tipo alt cuja única guarda verifica se $d1$ é diferente de $d2$.

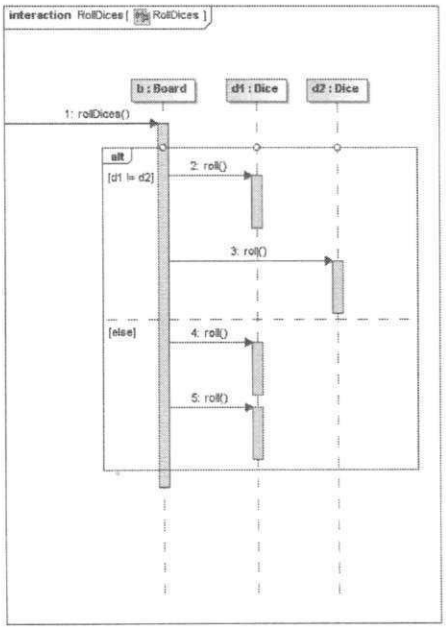


Figura 5.16: Diagrama de Sequência para o método $rollDices()$

Passo 2. Uma vez que temos o diagrama que serve de fonte para a técnica, podemos executar o próximo passo o qual consiste na escrita do corpo do método $instantiateModelObjects()$ detalhado na Subseção 5.1.1. Para escrever o corpo do método devemos ser cautelosos para que as regras de *design* executadas posteriormente durante

os testes tenham dados suficientes para sua execução, e para que esses dados sigam o modelo de dados definido na Seção 4.1. Assim sendo, no primeiro momento instanciamos os objetos que representam os elementos do diagrama de sequência conforme ilustrado na Figura 5.17. Como podemos ver cada um dos objetos existentes no diagrama é representado por um $FieldNode$ e seu nome é atribuido de acordo com a classe a que ele pertence. Por exemplo o objeto b da classe $Board$ é representado pelo objeto $FieldNode$ nomeado como $Board\_b$ $(linha4 - 5)$.

```
1  modelEntities = new java.util.HashSet<org.designwizard.design.Entity>();
2  rootMethods = new java.util.ArrayList<org.designwizard.design.MethodNode>();
3
4  org.designwizard.design.FieldNode Board_b_ObjectNode =
5      new org.designwizard.design.FieldNode("Board.b");
6  modelEntities.add(Board_b_ObjectNode);
7
8  org.designwizard.design.FieldNode Dice_d1_ObjectNode =
9      new org.designwizard.design.FieldNode("Dice.d1");
10 modelEntities.add(Dice_d1_ObjectNode);
11
12 org.designwizard.design.FieldNode Dice_d2_ObjectNode =
13     new org.designwizard.design.FieldNode("Dice.d2");
14 modelEntities.add(Dice_d2_ObjectNode);
```

Figura 5.17: Instanciação dos Objetos Referentes aos Elementos UML

Passo 3. A seguir, devemos indicar qual método pertence a qual objeto, *i.e.*, devemos relacionar qual objeto deve receber determinada mensagem. Para exemplificar a criação dessa relação apresentamos na Figura 5.18 a instanciação da representação do método $rollDices()$ da classe $Board$, bem como a adição da relação de posse $(CONTAINS)$ entre o objeto contenedor $(Board\_b\_ObjectNode)$ e o método contido$(rollDices())$.

```
1 org.designwizard.design.MethodNode Board_rollDices__ =
2     new org.designwizard.design.MethodNode("Board.rollDices()", false);
3 Board_b_ObjectNode.addRelation(
4         new org.designwizard.design.relation.Relation(
5                 Board_b_ObjectNode,
6                 Board_rollDices__,
7                 org.designwizard.design.relation.Relation.TypesOfRelation.CONTAINS));
```

Figura 5.18: Instanciação da relação entre objeto e mensagem a ser recebida

Passo 4. Como explicado anteriormente na descrição do modelo de dados, o fragmento combinado alt, deve ser reprentado por uma sequência de comandos if, tantos quantos forem necessário, e um comando $else$ cuja função é indicar qual trecho de código deve ser exe-

cutado caso nenhum bloco if o seja. A Figura 5.19 ilustra a criação de um $IfBlockNode$ cuja guarda é $d1! = d2$, representando o primeiro operando do fragmento alt. Além disso a Figura 5.19 apresenta a instanciação da representação da mensagem $Dice\_roll\_\_$, bem como a montagem da relação entre a representação do comando if e a representação da mensagem $Dice\_roll\_\_$.

```
1  org.designwizard.design.block.IfBlockNode ifBlockNode_1 =
2      new org.designwizard.design.block.IfBlockNode("d1 != d2");
3  org.designwizard.design.MethodNode Dice_roll__ = new org.designwizard.design.MethodNode(
4          "Dice.roll()", false);
5  Dice_d1_ObjectNode.addRelation(
6          new org.designwizard.design.relation.Relation(
7                  Dice_d1_ObjectNode,
8                  Dice_roll__,
9                  org.designwizard.design.relation.Relation.TypesOfRelation.CONTAINS));
10 ifBlockNode_1.addRelation(
11         new org.designwizard.design.relation.Relation(
12                 ifBlockNode_1,
13                 Dice_roll__,
14                 org.designwizard.design.relation.Relation.TypesOfRelation.INVOKEVIRTUAL));
```

Figura 5.19: Instanciação da relação entre bloco if e método $roll()$

Vale salientar que o conteúdo da Figura das figuras apresentadas neste estudo de caso foi gerado automaticamente a partir dos diagrams de sequência utilizados neste para o sistema sob teste. O resultado da execução dos testes pode ser observado na Figura 5.20 onde podemos ver a sequência na qual os métodos de teste foram executados (à esquerda) e o recorte da classe de teste executada. De acordo com o especificado, temos que:

Figura 5.20: Execução da Classe de Teste $RollDicesSequenceTest$

- A classe de testes chama-se $RollDicesSequenceTest$, e possui os métodos indicados na Seção 5.1.
- Testamos a existência dos objetos ($testLifelineExistence$).

- Testamos a existência dos métodos envolvidos na sequência ($testExecutionSpecificationExistence$).
- Testamos a sequência em si ($testRootSequences$).

Como podemos observar nessa figura, os tempos de execução dos métodos em questão são baixos. Isso se deve à baixa complexidade e ao tamanho reduzido deste estudo de caso, uma vez que serve apenas de exemplo para a utilização da ferramenta, a qual foi produzida para suportar os testes de *design* propostos nessa dissertação.

5.4 Suporte Ferramental

Como pudemos ver nas seções anteriores, a aplicação da técnica para construção de testes de *design* através de regras de *design* desenvolvida neste trabalho exige que muitos detalhes sejam considerados. Para que esses detalhes não sejam esquecidos, ou seja, para reduzir a possibilidade de falha durante a criação dos testes de *design* desenvolvemos um conjunto de ferramentas capazes de transformar as informações contidas no diagrama de sequência em código de teste propriamente dito.

Neste trabalho nós desenvolvemos uma abordagem MDA capaz de gerar automaticamente o código fonte contendo testes de *design* a partir dos diagramas de sequência que modelam o seu comportamento. Para tal, usamos a linguagem ATL [ATL05][ATL11] - linguagem adotada para as transformações de modelo a modelo - que interpreta o modelo contido no diagrama de sequência e o transforma no modelo Java correspondente. Num segundo momento aplicamos o modelo Java a um conjunto de transformações escritas em MOFScript [Mof11] - linguagem de transformação de modelo a texto - para transformar o modelo Java, obtido como resultado das transformações ATL, em código fonte Java escrito de acordo com o próprio metamodelo da linguagem de destino. Vale ressaltar aqui que tanto o modelo quanto o código fonte gerado contêm os testes com todas as regras de *design* elaboradas nesta dissertação. Assim sendo, o modelo UML fornece as informações usadas para construção dos testes, e não para o código do *software*, como se poderia pensar. Por outro lado, os testes confrontam as informações contidas no diagrama de sequência UML com o código do *software* sob teste.

O conjunto de transformações está dividido em dois módulos. O primeiro deles é formado pelas transformações que constroem um modelo Java contendo o detalhamento a respeito do que deve conter o código de teste. Para realizar a transformação entre modelos, *i.e.*, de UML para Java, adotamos a linguagem a ATL [ATL11] [ATL05]. Esta linguagem, apesar de não ser o modelo padrão adotado pela OMG [OMG11], possui suporte ferramental robusto e se integra muito bem a IDE Eclipse [Ecl11], sendo por isso escolhida para as transformações de UML para Java. A Figura 5.21 apresenta as configurações necessárias para a IDE Eclipse controlar a execução das transformações ATL. Nesta tela, devemos informar qual o arquivo ATL contém as regras a serem executadas ($JavaStatements.atl$), quais os metamodelos devem ser usados como guias para as transformações ($uml2$:metamodelo UML, $parameter$: uso simplificado de parâmetros do tipo String, $JavaAbstractSyntax$: Sintaxe abstrata de Java), qual modelo servirá de fonte ($RollDice.uml$) para a transformação, e, por fim, qual será a saída do processamento da transformação ($SaidaRollDice.xmi$).

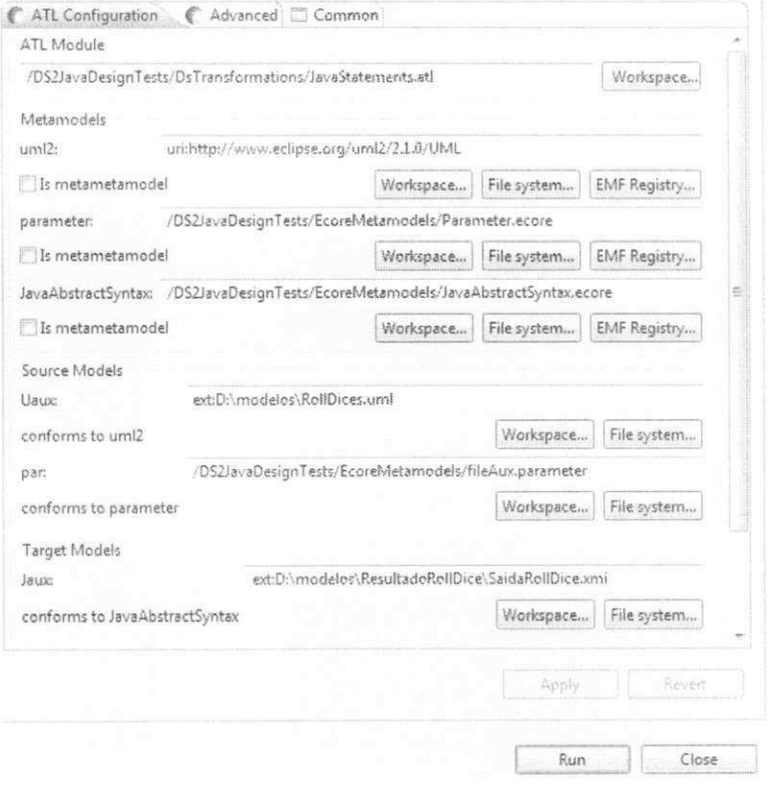


Figura 5.21: Preparação para execução de transformações usando ATL

O segundo módulo de transformação consiste em converter o modelo Java

($SaidaRollDice.xmi$) em código fonte propriamente dito. Para realizar esta tarefa usamos as transformações desenvolvidas por Pires *et al.* [PRLS10]. Ao final do processo temos a classe de teste escrita em Java, contendo todas as regras de *design* necessárias para verificar a conformidade entre o modelo UML de entrada do processo e o código fonte desenvolvido para o sistema sob teste.

5.4.1 Arquitetura da Ferramenta

A ferramenta construída para dar suporte ao processo de montagem dos testes de *design* foi desenvolvida segundo a arquitetura ilustrada na Figura 5.22. Esta é a arquitetura do único módulo da ferramenta, chamado *SequenceToDesignTest*. Este módulo, alimentado pelo diagrama de sequência e tomando por base os metamodelos Java e UML, executa as transformações ATL (*de modelo para modelo*) e MOFScript (*de modelo para código*). Essa primeira transformação gera o arquivo XML que contém a representação do modelo Java para os testes, ainda em sintaxe abstrata. Uma vez que a sintaxe abstrata é montada ela passa pela segunda transformação, que finalmente constrói a código fonte Java, em sintaxe concreta. Como podemos ver na Figura 5.22, cada um dos componentes desta arquitetura disponibiliza interfaces para o processamento das transformações que lhes compete. Vale ressaltar que o componente responsável pelas transformações ATL requer apenas uma interface do componente MOFScript, e esta interface é responsável por transformar todo o modelo gerado pela transformação ATL. Outro ponto importante é que as interfaces de cada um dos componentes usam funcionalidades das outras entre si. Desta forma, por exemplo, o gerenciamento de mensagens (interface $IMessageTransformable$) usa funcionalidades do gerenciamento de objetos (interface $IObjectTransformable$), ou do gerenciamento de fragmentos (interface $IFragmentTransformable$), para recuperar informações do diagrama de sequência sobre o objeto que a chamou, ou o fragmento que a compõe, respectivamente.

O processo interno de execução das transformações MDA em cada um dos módulos da Figura 5.22 pode ser visualizado na Figura 5.23. Nesta figura, podemos ver como é realizado o processo de extração de informações a partir dos diagramas de sequência de maneira simples. Esse processamento, apoiado pelas informações contidas nos metamodelos, permite que cada processo de extração ocorra de maneira específica de acordo com os tipos de elementos contidos na sequência, bem como permite reusar funcionalidades quando as re-

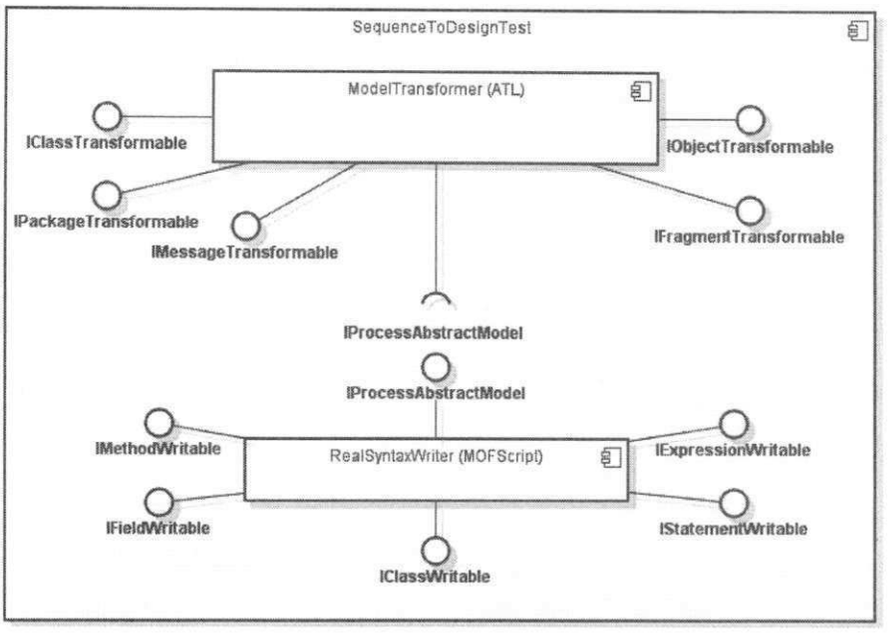


Figura 5.22: Arquitetura Adotada para a Ferramenta

presentações são equivalentes. Assim sendo, mensagens e fragmentos são tratados cada um a seu modo, mas as suas sequências internas tem a mesma forma de ser reprentada, portanto são interpretadas de uma mesma forma. As informações, coletadas através transformações ATL, são então usadas para gerar o modelo o modelo abstrato de Java, que contem os testes de *design*. Por fim, o modelo abstrato é usado pelas transformações MOFScript para gerar o código fonte dos testes de *design* seguindo o mesmo processo indicado na Figura 5.23, mas produzindo código concreto, ao invés do modelo Java abstrato, gerado via ATL.

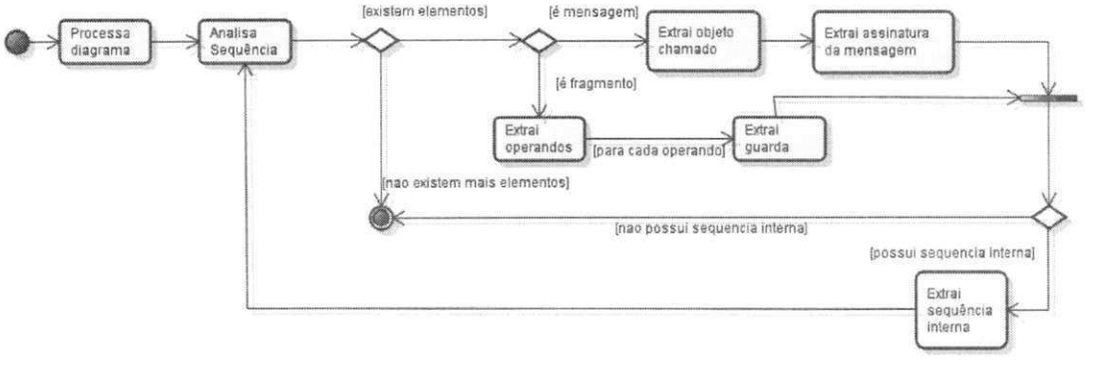


Figura 5.23: Atividades envolvidas nas Transformações MDA

Capítulo 6

Aplicação da Metodologia GQM na Avaliação dos Resultados

Para executar a avaliação da abordagem proposta para testes de *design* comportamentais adotamos a metodologia GQM (*Goal, Question, Metric*) [BCR94] [vSB99]. A abordagem GQM segue exatamente o que o nome diz, *i.e.*, é preciso ter um objetivo bem descrito (*Goal*), levantar questões de pesquisa e hipóteses possíveis (*Questions*), e definir quais as métricas devem ser usadas para responder as questões propostas (*Metrics*). A seguir apresentamos o detalhamento da aplicação de GQM neste trabalho.

6.1 Fase de Planejamento

6.1.1 Perfil da Equipe

A equipe foi formada por um único membro, e este foi responsável por executar os testes e analisar os resultados obtidos. Isto se deu devido à característica automática dos testes, pois os mesmos necessitam apenas de controle para inicio e fim da bateria de execuções. Além disso, um supervisor foi responsável por confirmar a validade dos experimentos e dos dados coletados.

6.1.2 Descrição do Experimento

O experimento foi realizado na forma de estudo de caso para avaliação da técnica aplicada a um sistema real, orientado a objetos e escrito em Java. Para isso selecionamos softwares disponíveis na internet com base em quatro critérios, que foram:

1. Ser um software largamente utilizado;
2. Possuir testes funcionais que garantam seu funcionamento;
3. Possuir documentação descrevendo suas funcionalidades;
4. Possuir projeto em que existam modelos compostos por diagramas de sequência.

Em um cenário ideal os quatro critérios deveriam ser cobertos, entretanto nenhum dos softwares selecionados para o experimento cobriu o critério número 4. Uma vez que os demais critérios foram atendidos, optamos por escolher o sistema em que houvesse maior familiaridade por parte da equipe, com isso garantimos o conhecimento do que está sob teste por parte da equipe. A Tabela 6.1 apresenta os softwares escolhidos como candidatos ao estudo de caso, bem como quais critérios cada software cobre.

Tabela 6.1: Softwares candidatos ao uso durante a avaliação.

Software	Linhas de Código	Utilizado largamente	Testes Funcionais	Documentação	Diagramas de Sequência
EhCache [EhC11]	21.695	SIM	SIM	SIM	NÃO
FindBugs [Fin11]	68.830	SIM	SIM	SIM	NÃO
Hibernate [Hib11]	934.299	SIM	SIM	SIM	NÃO
JEdit [JEd11]	70.021	SIM	SIM	SIM	NÃO
SoapUI [Soa11]	250.113	SIM	SIM	SIM	NÃO

Dentre os softwares listados na Tabela 6.1, escolhemos o FindBugs como objeto do estudo de caso. O FindBugs é um software capaz de realizar analise estática em código fonte escrito em Java em busca de erros de programação, os chamados *bugs*. Uma vez que, tanto no FindBugs quanto no outros softwares selecionados, existe falta de documentação por meio de diagramas de sequência, e este critério é essencial para a avaliação do experimento, precisamos adotar um mecanismo que pudesse recuperar esta documentação. Para atingir esse objetivo e fazer com que o FindBugs cubra os quatro critérios exigidos, escolhemos 20

métodos para realizar engenharia reversa de seus códigos fonte, e assim obter os diagramas que necessitamos.

O número reduzido de métodos justifica-se por dois motivos. Primeiro, os métodos foram escolhidos de maneira que pudessem abranger diferentes tamanhos e quantidades de componentes que formam diagramas de sequência, a saber, métodos e fragmentos combinados. Segundo, a engenharia reversa para diagramas de sequência ainda não é um processo perfeito. Portanto, mesmo facilitando a tarefa de construção dos diagramas, fez-se necessário aplicarmos correções aos diagramas para que atendessem perfeitamente a linguagem UML, sendo esta uma tarefa demorada, o que inviabiliza a engenharia reversa de todo o software. Vale ressaltar que estas correções não inviabilizam a justificativa de utilização de engenharia reversa, pois, mesmo sob risco de inserir erros nos diagramas, o processo de correção foi realizado metódicamente e o resultado foi analisado e validado logo em seguida.

A Tabela 6.2 apresenta os métodos escolhidos, bem como as quantidades de fragmentos combinados, de métodos invocados e o tamanho dos métodos medido em linhas de código (LOC). Como podemos observar a quantidade de linhas de código está entre 10 e 30 LOC, entretanto foram escolhidos alguns casos com valores inferiores e superiores a esses limites com o intuito de analisar situações além do que seria considerado normal para o sistema.

6.2 Fase de Definição

Nesta fase é definido quais objetivos devem ser atingidos ao final do processo de avaliação. Aqui também é definido quais as questões que mais se adéquam ao contexto do projeto para definir se os objetivos foram atingidos, bem como as métricas para responder a essas perguntas.

6.2.1 Objetivos

Essa avaliação teve por objetivos avaliar os recursos necessários para execução dos testes de *design* comportamentais e avaliar a precisão destes testes. A análise de utilização de recursos levou em consideração apenas o tempo utilizado na execução, pois tempos elevados podem representar problemas de aceitação da abordagem. Já a análise de precisão levou em consideração a existência de falsos-positivos e falsos-negativos nos resultados obtidos.

Tabela 6.2: Métodos do FindBugs submetidos à avaliação

Métodos	Fragmentos Combinados	Quantidade de Métodos Invocados	Tamanho (Linhas de Código)
FindBugs2.execute()	3 break, 2 loop, 2 alt, 3 opt	62	59
FindBugs2.buildClassPath()	3 loop, 3 opt	20	23
FuzzyBugComparator.compare()	11 opt, 3 alt, 1 loop	69	45
AddAnnotation.execute()	1 loop, 1 break, 1 opt	13	13
Findbugs.configureTrainingDatabases()	3 opt, 2 loop, 3 alt, 1 break	35	68
AddMessages.execute()	2 loop	40	25
JavaVersion.<init>()	1 opt, 1 break, 1 alt	17	13
AddMessages.addBugCategories()	2 loop, 3 opt	16	20
HTMLBugReporter.getStylesheetStream()	2 break, 2 opt	15	16
TypeAnnotation.format()	1 alt, 1 opt	9	8
TextUIBugReporter.reportAnalysisError()	4 opt, 1 loop	8	15
TextUIBugReporter.checkBugInstance()	2 opt, 1 break, 1 loop	9	10
PluginLoader.getResource()	5 opt	8	15
SwitchHandler.getNextSwitchOffset()	2 opt, 1 loop	7	11
QueryBugAnnotation.scan()	2 loop, 1 break	5	11
PluginLoader.addCollection()	1 opt, 1 break	12	7
OpcodeStack.resetForMethodEntry()	3 opt, 1 break	10	19
ParameterWarningSuppressor.match()	4 opt	8	11
OpcodeStack.pushByLocalObjectLoad()	2 opt	8	11
DiscoverSourceDirectories .findSourceDirectoriesForAllSourceFiles()	1 break, 1 opt	17	18

Tempo

Em uma análise de aspecto não funcional, decidimos por verificar informações referentes ao tempo necessário para realizar as verificações de conformidade através da execução automática dos testes. Assim, temos uma idéia de quanto recurso é gasto pela técnica para que a mesma funcione corretamente.

Quantidade de inconformidades encontradas

Para avaliarmos o trabalho realizado corretamente devemos verificar a taxa de acerto da nossa técnica em relação às inconformidades encontradas na execução dos testes. Portanto, contabilizar as ocorrências de falsos-positivos e falsos-negativos no que se refere a existência de inconformidades, bem como os casos onde a técnica obtenha resultados corretos, para só então analisar esses casos e determinar o que causou esses problemas. Tomamos como falsos-positivos os casos em que inconformidades são encontradas em pontos onde elas não

existem, e por falsos-negativos os casos onde os testes indicam sucesso, mas onde na verdade existe alguma inconformidade.

6.2.2 Questões e Métricas

Tempo

Q1. Quanto tempo foi gasto para realizar a execução dos testes?

Métrica: T.

Onde T é dado em segundos gastos na tarefa.

Quantidade de falsos-negativos encontrados

Q1. Qual a quantidade de falsos-negativos encontrados nas busca pelas inconformidades do sistema sob teste?

Métrica: QFN

Onde QFN é a quantidade de inconformidades onde existe falsos-negativos.

Quantidade de falsos-positivos encontrados

Q1. Qual a quantidade de falsos-positivos encontrados nas busca pelas inconformidades do sistema sob teste?

Métrica: QFP

Onde QFP é a quantidade de inconformidades onde existe falso-positivo.

6.3 Coleta de Dados e Análise

A coleta de dados e a análise correspondem às duas últimas etapas do processo de GQM. Tratamos essas duas fases em conjunto devido as suas características bastante interligadas. Estas duas fases são responsáveis por responder às perguntas da fase de definição. Vale ressaltar, que foram realizados 10 execuções de cada teste e que todos os testes realizados apresentaram basicamente o mesmo tempo de execução e a mesma quantidade de inconformidades, não havendo variação digna de nota entre uma execução e outra de um mesmo teste.

6.3.1 Tempo

Os tempos gastos na execução de cada teste, excetuando-se o tempo usado pra carga da API *Design Wizard*, são encontrados da Tabela 6.3. Vale ressaltar que, estes valores foram obtidos utilizando um PC com processador Pentium Dual Core T2310 de 1,43GHz, e 3GB de memória RAM. Nesta tabela, os tempos são apresentados em segundos e os valores correspondem às médias de tempo encontradas nas execuções dos testes de *design* para cada método. Essas informações da Tabela 6.3 permitem perceber que o tempo gasto na realização dos testes não é impedimento para a realização nem desse nem de qualquer outro teste mesmo em casos de funcionalidades complexas ou que possuam um conjunto extenso de elementos. Outra percepção feita após a análise dos tempos gastos é que, como esperado, o tamanho dos métodos sob teste (LOC) e sua complexidade (número de fragmentos combinados) influem no tempo de execução dos testes, mas não foi possível indicar com precisão qual desses dois fatores mais influência nos tempos de execução dos testes.

Tabela 6.3: Dados sobre Tempo

Métodos	Tempo Médio (segundos)
FindBugs2.execute()	44,45
FindBugs2.buildClassPath()	10,02
FuzzyBugComparator.compare()	45,30
AddAnnotation.execute()	0,75
Findbugs.configureTrainingDatabases()	19,80
AddMessages.execute()	21,25
JavaVersion.<init>()	8,04
AddMessages.addBugCategories()	7,60
HTMLBugReporter.getStylesheetStream()	5,35
TypeAnnotation.format()	2,85
TextUIBugReporter.reportAnalysisError()	1,96
TextUIBugReporter.checkBugInstance()	2,72
PluginLoader.getResource()	2,58
SwitchHandler.getNextSwitchOffset()	1,52
QueryBugAnnotation.scan()	1,19
PluginLoader.addCollection()	1,80
OpcodeStack.resetForMethodEntry()	1,83
ParameterWarningSuppressor.match()	0,49
OpcodeStack.pushByLocalObjectLoad()	0,52
DiscoverSourceDirectories .findSourceDirectoriesForAllSourceFiles()	3.15

6.3.2 Quantidade de Inconformidades Encontradas e Precisão

A análise das quantidades de inconformidade em busca de falsos-positivos e falsos-negativos é parte essencial deste trabalho de dissertação, e a parte principal no que se refere à coleta de dados. Nesse ponto, analisamos a técnica proposta sob diferentes pontos de vista. No primeiro caso, contabilizamos quantas inconformidades apareceram, na execução dos testes sem modificações em parte alguma. Num segundo caso inserimos inconformidades propositais no modelo para que pudéssemos descobrir se ele encontra essas discrepâncias. De modo análogo, em um terceiro momento mantivemos o modelo intacto e alteramos apenas o código do sistema sob teste, com o mesmo intuito de adicionar inconformidades propositalmente.

Assim sendo, uma vez preparados e executados os testes, encontramos constância na quantidade de falsos-positivos e falsos-negativos, sempre com valores pequenos. Mesmo não sendo em grandes quantidades que viessem a inviabilizar a técnica proposta, esses falsos-positivos e falsos-negativos apontam pontos onde a solução ainda precisa melhorar. Entretanto, aparecem mais frequentemente em diagramas mais complexos, que exigem cuidados mais específicos para evitar confusões na indicação de inconsistências nos testes. Isso se dá pela representação imprecisa de determinadas sequências, que por falta de correspondência semântica, não puderam ser implementadas a contento, visto que a definição de uma correspondência semântica mais completa, está fora do escopo desta dissertação. Um exemplo é o fragmento *break*, que pode ser interpretado de 3 maneiras distintas dependendo dos fragmentos que são combinados em sua utilização, ora com semântica de captura de execeção, ora com semântica de abortagem de fragmentos em que os operandos possuem o fragmento *break*. A Tabela 6.4 apresenta as porcentagens de acertos, falsos-positivos e falsos-negativos encontrados nos três casos de testes a saber: sem alterações, alteração apenas no modelo, alteração apenas no código.

Tabela 6.4: Porcentagem de Acerto na Descoberta de Inconsistências

Caso	Inconsistências Corretas	Falsos-Positivos	Falsos-Negativos
Caso 1: Sem alterações	85%	10%	5%
Caso 2: Alterações nos modelos	80%	10%	10%
Case 3: Alterações no código	78%	8%	14%

Uma análise mais profunda permitiu identificar alguns fatores que influenciam o aparecimento dos erros indicados na Tabela 6.4. Levando em conta que a detecção de inconsistência

de maneira correta é a resposta esperada, ou ideal, por isso não vamos entrar em maiores detalhes para esse caso. Entretanto, os erros ocorridos e relatados na Tabela 6.4 onde detectamos tanto falsos-positivos quanto falsos-negativos devem-se aos seguintes fatores:

- A condições existentes nos fragmentos combinados (*guard*) nem sempre são codificadas de acordo com o padrão definido no diagrama de sequência. Aindam assim, mesmo sendo codificado de outra forma, possuem a mesma semântica. Desta forma, aponta-se a incidência de falsos positivos, ou seja, indica-se a existência de erros onde eles não existem.

- A repetição de uma mesma entidade (mais comumente métodos) em uma sequência, pode fazer com que uma inconsistência não seja detectada corretamente, caracterizando, assim, a existência dos falsos negativos.

- A otimização relizada durante a compilação do código Java, altera a ordem de execução de alguns trechos de código para aumentar o desempenho das aplicações.

Este último ponto, referente a compilação dos sistemas em Java, será trabalhado mais profudamente em trabahos futuros, podendo aplicar um procedimento equivalente nas transformações MDA realizadas. Assim, dado que compilação Java e transformações ATL passariam a comportar-se de maneiras equivalentes, esperamos diminuir a incidência de falsos-positivos e falsos-negativos.

Capítulo 7

Trabalhos Relacionados

Neste capítulo listamos os principais trabalhos relacionados à abordagem desenvolvida nesta dissertação. Os trabalhos listados aqui não fazem a verificação de conformidade entre modelos e códigos implementados da mesma forma que a técnica desenvolvida, entretanto fornecem apoio para a sua formulação.

Não encontramos na literatura qualquer outro trabalho que apresente uma abordagem semelhante à apresentada neste trabalho para o teste de *design* comportamental. Anteriormente, apenas Pires *et al.* [PRSL08] [PRLS10] desenvolveram uma abordagem semelhante à desenvolvida aqui, entretanto apenas é possível verificar a conformidade estrutural em *softwares* usando a técnica apresentada por esses autores.

Em [PRSL08], os autores aplicam o conceito de testes de *design* para verificar se o código implementado está em conformidade com os diagramas de classe UML construídos para um determinado sistema. Usando um processo automático baseado em uma abordagem MDA, os autores preenchem um conjunto de *templates* de teste para gerar casos de teste JUnit, sendo um caso de teste para cada aspecto que pode ser testado a partir de um diagrama de classe, tais como a implementação de interfaces, ou a existência de métodos em classes. Tentando mostrar a variedade de aplicações para este processo, em [PRLS10] os autores aplicam essa técnica especificamente para gerar testes que verificam se padrões de projeto modelados em diagramas de classe UML estão de acordo com as suas respectivas implementações em Java. Esta solução funciona bem na verificação de aspectos estruturais do software, porque todas as verificações podem ser feitas em partes específicas do código para verificar um aspecto estrutural específico, como a herança, por exemplo. No entanto, a

verificação dos aspectos comportamentais com um método equivalente que instancie esses *templates* não é intuitivo. Em nossa abordagem, adotamos *templates* ou modelos apenas para criar um grafo com as entidades e relações que compõem o diagrama de sequência, mas a execução do teste segue sempre os mesmos passos. Portanto, nós não criamos inteiramente o teste para cada diagrama testado, mas sim aplicamos os templates de testes já existentes a cada novo conjunto de dados retirado de cada diagrama de sequência, para só então executarmos os testes. Com isso ganha-se tempo ao evitar que o código do teste, que é sempre o mesmo, seja reescrito a cada novo teste que se queira preparar.

A verificação de conformidade pode ser usada para recuperar informações a respeito de códigos fonte através de engenharia reversa, podendo assim extrair modelos desse conjunto de informações, como em [MNS01]. Neste trabalho os autores propõem que os modelos extraídos por meio de engenharia reversa devem ser verificados e comparados com os modelos projetados pelo analista de software. Assim sendo, o analista avalia manualmente a diferença entre esses modelos, e interpreta o resultado - a diferença entre ambos - como um erro produzido pela ferramenta, ou uma distinção real entre o modelo e a implementação. Neste último caso, a diferença pode ser interpretada como um problema, que, se bem tratado pode resultar na evolução do sistema, de acordo com a experiência do analista com aquele sistema. Entretanto, a solução apresentada pelos autores não considera uma questão importante: a necessidade de intervenção humana para produzir a análise. Assim sendo, esta característica torna a solução propensa a erro, pois exige alto conhecimento de engenheiros para avaliar as diferenças entre os modelos. Por outro lado, esta dissertação apresenta uma abordagem capaz de: (i) automatizar o processo de checagem de conformidade de aspectos comportamentais, e (ii) aproximar mais o desenvolvedor do processo de desenvolvimento dos testes, visto que o código de teste é escrito na mesma linguagem adotada pelo desenvolvedor.

Por outro lado, em [HCSS08], os autores tentam resolver o problema de verificação de conformidade entre especificação e implementação considerando as relações entre os módulos do software. Este trabalho utiliza um conjunto de dados chamado *Design Structure Model* (DSM) que é extraído de diagramas e do código-fonte, sendo ambos os resultados comparados automaticamente em busca de diferenças entre eles. A DSM é uma matriz onde são definidas regras para o *design*, as quais representam relações estruturais entre os módulos do sistema. A verificação de conformidade baseia-se na busca de homomorfismo entre os

dois DSM - modelo *versus* código. Segundo os autores, este é um problema NP-completo, e para tentar evitar essa complexidade, os autores utilizam um algoritmo genético para projetar gráfico em outro, e encontrar diferenças. No entanto, esta solução tem dois problemas: primeiro, se o código é muito diferente do módulo a solução não vai funcionar como proposto, pois o algoritmo genético não irá convergir. Em segundo lugar, a aplicação de algoritmos genéticos apresenta problemas de desempenho nesta verificação da conformidade, introduzindo erros na solução.

A abordagem baseada em Model Driven Development (MDD), criada por [DTKG+05], tem por objetivo gerar testes para modelos UML executáveis para que se possa testar as funcionalidades de um software ainda no nível de abstração do *design*. Usando diagramas de interação e de classe, a ferramenta criada gera o código para teste dos modelos, e a estrutura dos dados que deve ser usada para instanciar os testes. De modo complementar, a ferramenta utiliza uma linguagem chamada JAL para apoiar uma semântica UML específica para ser usada nos testes e uma ferramenta chamada USE, criada para verificar se as pré e pós condições são seguidas durante a execução de testes. Essas condições são baseadas em restrições OCL e nos diagramas de classe que formam a base de dados dos testes. Os autores pretendem incorporar mais diagramas para melhorar os resultados obtidos nos testes, para assim poderem detalhar com mais precisão as restrições a aplicadas ao conjunto de testes. Assim sendo, diferentemente do nosso trabalho, os autores apenas aumentam o nível de abstração de execução do teste, mas os desenvolvedores ainda têm o mesmo problema não resolvido: o *gap* entre os modelos e o código ainda existe.

Capítulo 8

Considerações Finais

Neste trabalho apresentamos uma abordagem para a realização de teste de *design* para verificação de conformidade de software. Teste de *design* é um tipo de teste capaz de indicar quando o software implementado está de acordo com a especificação feita através de modelos. Este tipo de teste não se caracteriza como teste funcional, pois este tipo de teste ajuda a equipe de desenvolvimento a verificar se o código está de acordo com a documentação do software, mas não se preocupa em indicar se as funcionalidades estão corretas. Queremos saber se o que foi feito está de acordo com as especificações, e não se funciona corretamente.

A nossa abordagem é focada em testes de *design* para aspectos comportamentais, utilizando diagramas de sequência UML como fonte para guiar o processo de criação dos testes. Trabalhamos com as entidades que compõem uma sequência UML, dando preferência, nesse primeiro momento, para aqueles que são considerados mais utilizados, como mensagens e alguns tipos de fragmentos combinados, como *alt*, *opt*, e *loop*.

O uso de UML 2.0 aumenta a semântica do diagrama de sequência, uma vez que permite o uso de novas entidades para construir diagramas mais complexos. Entretanto, essa linguagem não tem uma semântica formal e bem definida na maioria dos diagramas, incluindo diagramas de sequência. Uma vez que existe essa falta de formalismos em UML, nós realizamos nosso trabalho de modo independente e flexível e definimos nossas próprias restrições à análise e criação da abordagem de testes apresentada aqui. Estamos conscientes sobre a formulação dessas restrições e sabemos que elas podem causar problema de aceitação para os usuários, mas é necessário determinar os limites do nosso trabalho, uma vez que a falta de formalismos deixa a UML com uma liberdade indesejada para interpretação. Consideramos

que este é um ponto importante a falar, mesmo que minimamente, porque restringe o nosso trabalho, controlando o que os usuários podem fazer. Assim, o que acontece com diagramas de sequência que não sigam nossas restrições é imprevisível, ou seja, a solução pode funcionar em alguns casos fora de nosso escopo, mas não damos qualquer garantia sobre isso.

Nossa abordagem age consultando o *bytecode* Java pertencente ao software sob teste o qual é analisado por uma API, chamada *Design Wizard* capaz de construir um grafo de entidades e relações entre essas entidades. Uma vez que esse grafo é montado a nossa abordagem realiza consultas a este grafo para recuperar as informações exigidas pelos diagramas de sequência e mostra para o usuário se a implementação corresponde aos modelos.

Um importante aspecto de nossa abordagem é a utilização de ferramentas de MDA, assim como Pires *et al.* [PRSL08][PRLS10]. O conjunto de transformações MDA, que tem como ponto de partida os diagramas de sequência, é utilizado para gerar automaticamente nossos testes de *design*. Esta característica é essencial para a ampla adoção da nossa abordagem, uma vez que impede erros nos testes de *design* e diminui o tempo usado para executar esta tarefa. Desta forma, pretendemos favorecer o processo de adoção deste tipo de teste, o qual é naturalmente detalhista.

8.1 Limitações

Para avaliar a nossa abordagem para teste de *design* utilizamos um conjunto de 20 métodos do software Findbugs. Identificamos através dessa avaliação alguns pontos que precisamos melhorar em nosso trabalho. Isto acontece pois encontramos um número pequeno, mas ainda relevante, de falsos-positivos e falsos-negativos na execução de nossos testes de *design*. Esses resultados, merecem um trabalho mais detalhado para que tenham seu número reduzido ou zerado, no melhor caso.

Outra limitação diz respeito a utilização da ferramenta construída para a execução das transformações MDA desenvolvidas nesta dissertação. Isto se dá pois a mesma ainda está interligada diretamente a IDE Eclipse o que pode limitar a sua utilização.

Outro ponto a ser levado em consideração são as otimizações feitas no código compilado por parte do compilador da linguagem. Essas otimizações melhoram o desempenho das

aplicações, mas para que isso aconteça é necessário que algumas instruções sejam reposicionadas. Isso faz com que a nossa abordagem não consiga indentificar corretamente sequências que estão em conformidade no código, mas foram otimizadas, violando a sequência definida.

Por fim, uma importante limitação desta dissertação reside na quantidade de entidades que podem ser representadas no atual modelo de dados. Vale ressaltar que o modelo de dados atual já é uma extensão do modelo de dados original, adotado pelo *Design Wizard*, que foi melhorado para que pudesse representar as entidades do diagrama de sequência cobertos nesta dissertação, mas que ainda não cobre todas as entidades presentes neste tipo de diagrama.

8.2 Trabalhos Futuros

Como trabalhos futuros, pretendemos criar mais testes de *design* com foco nos demais tipos de fragmentos combinados, além de melhorar a nossa avaliação para considerar outros quantitativos (tempo e recursos computacionais utilizados) e qualitativos (facilidade de uso) da nossa abordagem.

Outro aspecto que vamos analisar em trabalhos futuros é o desempenho da nossa solução isoladamente para cada tipo de entidade para saber qual o custo real do teste de cada tipo de entidade existente nos diagramas de sequência.

Existe ainda a idéia de abranger a abordagem desenvolvida nesta dissertação para outras linguagens orientadas a objeto diferentes de Java, sendo necessário para isso um trabalho de compreensão na síntaxe e da semântica de cada nova linguagem. Uma das dificuldades da adoção da nossa abordagem em outra linguagem reside justamente nessa compreensão dos aspectos sintáticos e semânticos que a compõem. Entretanto, a maior dificuldade reside na compreensão dos metamodelos que deescrevem a linguagem. Uma vez que é preciso reconstruir as regras das transformações tanto para geração de modelos abstratos quanto de códigos concretos. Esta reconstrução não é uma tarefa trivial, pois deve ter um minimo de cobertura da linguagem para poder gerar um teste que funcione minimamente.

O melhoramento tanto da ferramenta construída, quanto do modelo de dados adotado são o principal trabalho a ser desenvolvido no futuro, para que tenhamos uma abordagem mais completa. Outro ponto importante quanto ao melhoramento dos resultados da aborda-

gem é a possibilidade de substuição do Design Wizard por outra API de consulta a código de software que permita a consulta direta a código fonte ainda em texto, e não em código compilado como nas versões existentes. Esta substituição busca, principalmente, resolver os problemas causados pela otimização feita pelos compiladores, a qual causa problemas na busca de inconformidades. Uma vez que possamos analisar o código da maneira exata como foi construído, poderemos aumentar a precisão de nossas comparações entre sequências de instruções, melhorando assim o resultado de nossa abordagem.

Bibliografia

[AAA07] Jonathan Aldrich and M. Abi-Antoun. Checking and Measuring the Architectural Structural Conformance of Object-Oriented Systems, 2007.

[AN05] Jim Arlow and Ila Neustadt. *UML 2.0 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition)*. Addison-Wesley Professional, 2005.

[ATL05] ATL: Atlas Transformation Language User Manual, 2005.

[ATL11] Atl. http://www.eclipse.org/m2m/atl/, 2011.

[BCR94] V.R. Basili, Gianluigi Caldiera, and H.D. Rombach. The Goal Question Metric Approach. *safety*, 1(5):1, 1994.

[BGF09] Joao Brunet, Dalton Guerrero, and Jorge Figueiredo. Design tests: An approach to programmatically check your code against design rules. *2009 31st International Conference on Software Engineering - Companion Volume*, pages 255–258, May 2009.

[DP06] B. Dobing and J. Parsons. How UML is used. *Communications of the ACM*, 49(5):113, 2006.

[DTKG+05] T. Dinh-Trong, N. Kawane, S. Ghosh, R. France, and a.a. Andrews. A Tool-Supported Approach to Testing UML Design Models. *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 519–528, 2005.

[DWS11] DWSite. Design wizard. http://www.designwizard.org, 2011.

[Ecl11] Eclipse. http://www.eclipse.org/, 2011.

[EhC11] Ehcache. http://www.ehcache.org/, 2011.

[Fin11] Findbugs. http://findbugs.sourceforge.net/, 2011.

[Gro09a] OMG Group. OMG Unified Modeling Language TM, Infrastructure. http://www.uml.org/, 2009.

[Gro09b] OMG Group. OMG Unified Modeling Language TM, Superstructure. http://www.uml.org/, 2009.

[Gro11a] OMG Group. Mda. http://www.omg.org/mda/, 2011.

[Gro11b] OMG Group. Uml. http://www.uml.org/, 2011.

[HCSS08] Sunny Huynh, Yuanfang Cai, Yuanyuan Song, and Kevin Sullivan. Automatic modularity conformance checking. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 411–420, New York, NY, USA, 2008. ACM.

[Hib11] Hibernate. https://www.hibernate.org/, 2011.

[JEd11] Jedit. http://www.jedit.org/, 2011.

[JUn11] Junit. http://www.junit.org, 2011.

[Kru00] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[MDA04] *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley Professional, 1 edition, March 2004.

[MNS01] G.C. Murphy, D. Notkin, and K.J. Sullivan. Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, April 2001.

[Mof11] Mofscript. http://www.eclipse.org/gmt/mofscript/, 2011.

[OMG11] Omg. http://www.omg.org, 2011.

[Ope11] Openup. http://epf.eclipse.org/wikis/openup/, 2011.

[PRLS10] Waldemar Pires, Franklin Ramalho, Anderson Ledo, and Dalton Serey. Checking UML Design Patterns in Java Implementations. *2010 Fourth Brazilian Symposium on Software Components, Architectures and Reuse*, pages 120–129, September 2010.

[PRSL08] Waldemar Pires, Franklin Ramalho, Dalton Serey, and Anderson Ledo. UDT - Uma Ferramenta para Geração de Testes de Design. *lbd.dcc.ufmg.br*, 2008.

[Soa11] Soapui. http://www.soapui.org/, 2011.

[vSB99] R. van Solingen and E. Berghout. The Goal/Question/Metric Method: a practical guide for quality improvement of software development. *McGraw Hill, ISBN*, 7(709553):7, 1999.