Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Aprimorando a Verificação de Conformidade em Programas Baseados em Contratos

Alysson Filgueira Milanez

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: Metodologia e Técnicas da Computação

Tiago Massoni e Rohit Gheyi

(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

M637a Milanez, Alysson Filgueira.

Aprimorando a verificação de conformidade em programas baseados em contratos / Alysson Filgueira Milanez. – Campina Grande, 2014.

107 f. : il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2014.

"Orientação: Prof. Dr. Tiago Massoni, Prof. Dr. Rohit Gheyi.

Referências.

1. Programação por Contratos. 2. Verificação de Conformidade. I. Massoni, Tiago. II. Gheyi, Rohit. III. Título.

CDU 004.43(043)

"APRIMORANDO A VERIFICAÇÃO DE CONFORMIDADE EM PROGRAMAS BASEADOS EM CONTRATOS"

ALYSSON FILGUEIRA MILANEZ

DISSERTAÇÃO APROVADA EM 06/06/2014

Tiago L. Massoni

TIAGO LIMA MASSONI, Dr., UFCG
Orientador(a)

Rohit Gheyi

ROHIT GHEYI, Dr., UFCG
Orientador(a)

RICARDO MASSA FERREIRA LIMA, Dr., UFPE
Examinador(a)

Wilkerson de L. Andrade

WILKERSON DE LUCENA ANDRADE, D.Sc, UFCG
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Teste é comumente usado para verificar conformidade em programas baseados em contrato; uma vez que verificação por provas formais tem baixo poder de escalabilidade e análise estática é, em alguns casos, limitada para identificar não-conformidades mais gerais. Casos de teste tradicionais, com dados de teste providos manualmente, podem ser ineficazes para detectar não-conformidades sutis que surgem apenas após diversas criações e modificações nos objetos sob teste. Essas não-conformidades podem sinalizar *bugs* mais sutis, diminuindo os benefícios de usar programas baseados em contrato. Casos de teste gerados aleatoriamente com dados de teste gerados automaticamente, por outro lado, são uma abordagem promissora quando testes mais substanciais são necessários. No presente trabalho, propomos e avaliamos uma abordagem, JMLOK 2.0, para detecção e categorização de não-conformidades, no contexto de *Java Modeling Language* (JML). Nossa abordagem objetiva auxiliar o programador no processo de correção de não-conformidades. A detecção é suportada por uma abordagem de Testes Gerados Aleatoriamente (RGT). E a categorização por uma abordagem baseada em heurísticas. Realizamos duas avaliações. Na primeira, realizamos uma avaliação de nossa abordagem para detecção de não-conformidades e de nosso processo para categorização manual: detectamos 84 não-conformidades em mais de 29 KLOC e mais de 9 K linhas de contratos JML (que iremos nos referir como KLJML); aplicando nosso modelo de categorização manual, obtivemos que a maioria das não-conformidades detectadas foram classificadas como erros de pós-condição; também observamos que uma não-conformidade é detectada após 2.54 *top-level* chamadas num caso de teste, em média, e que o número de chamadas internas ao caso de teste que revela a não-conformidade é, em média, de 2.23, nos dando evidências da necessidade de uma estrutura de testes mais elaborada para detecção de não-conformidades. Além disso, comparamos nossa abordagem com JET, uma ferramenta existente para detecção de não-conformidades em programas JML baseada em testes, utilizando um subconjunto dos programas usados no primeiro estudo (6 KLOC e 5 KLJML). JMLOK 2.0 detectou 30 não-conformidades com cobertura Java de 78.44% e JML de 67.67%, enquanto que JET detectou 9 não-conformidades cobrindo 47.97% de Java e 56.97% de JML. Na segunda, realizamos uma avaliação da nossa abordagem automática de categorização: comparamos a categorização manual e a automática e tivemos um valor de coincidências de 0.73 (considerando as não-conformidades da primeira avaliação) indicando que há similaridade entre as abordagens de categorização manual e automática; além disso, comparamos os resultados da categorização automática com a categorização realizada por experts em JML e também observamos alguma similaridade entre as abordagens.

Abstract

Testing is commonly used to check conformance in contract-based programs, as verification by formal proofs is hard to scale and static analysis is, sometimes, limited for detecting general nonconformances. Traditional test cases, with manually-provided data, may be ineffective in detecting subtle nonconformances that arise only after several instantiations and modifications in objects under test. Those nonconformances may signalize more subtle bugs, hindering the benefits of using contract-based programs. Random-generated tests with automatic test data generation, on the other hand, is a promising approach when more substantial testing is demanded. In this work, we propose and evaluate an approach, JMLOK 2.0, for automatically detecting and categorizing nonconformances, in the context of Java Modeling Language (JML). Our approach aims to help the programmer in the process of nonconformances correction. The detection is backed by Randomly-Generated Tests (RGT) approach. And the categorization is backed by heuristics-based approach. We perform two evaluations. First, we perform an evaluation of our detection approach and our manual categorization process: we detected 84 nonconformances in over 29 KLOC and 9 K lines of JML contracts (that we will refer as KLJML henceforth); applying our manual classification system we got that most detected nonconformances were classified as postcondition errors; we also observed that a nonconformance is detected after 2.54 top-level test case calls, in average, and the number of internal calls within the faulty test case call is an average of 2.23, providing evidence for the need of a more complex generated test structure in nonconformance detection; furthermore, we compare our approach with JET, an existing test-based approach for detecting nonconformances in JML programs, using a subset of programs from first study (6 KLOC and 5 KLJML). JMLOK 2.0 detected 30 nonconformances with Java coverage of 78.44% and JML coverage of 67.67%, while JET detected 9 nonconformances by covering 47.97% of Java and 56.97% of JML. Second, we perform an evaluation of our automatic categorization approach: we compare automatic and manual categorization and got a matches value of 0.73 (considering the nonconformances from first evaluation) indicating similarity between automatic and manual approaches; furthermore, we compare our results with the categorization performed by voluntary JML experts and we also observed some similarity.

Agradecimentos

Agradeço primeiramente a Deus, que me concedeu o dom da vida e por todas as bençãos que me concedeu ao longo de toda a minha vida.

Aos meus pais: Severino do Ramo Milanez e Rozane F. Milanez; por apoiar e acreditar nos meus sonhos e por sempre estarem ao meu lado, dando todo o apoio que necessito para superar as barreiras e obstáculos que a vida me proporciona. Às minhas irmãs Laryssa e Maxwellia, por sempre acreditarem em mim e torcerem pelo meu sucesso.

Meus sinceros agradecimentos aos orientadores e amigos Tiago e Rohit, que me auxiliaram durante essa jornada de pesquisa, sempre com muita dedicação, disponibilidade, paciência, empenho, e vontade de ensinar e orientar. Sem eles esse trabalho não seria possível. Agradeço também pelas oportunidades concedidas.

Aos professores Adalberto Cajueiro, Patrícia Machado pelas sugestões e contribuições neste trabalho. Aos professores Wilkerson de Lucena e Ricardo Massa por terem aceitado o convite para participar da banca do presente trabalho bem como por todos os excelentes comentários que nos proporcionaram *insights* para a continuação da nossa pesquisa.

Agradeço a todos os amigos e colegas do SPLab, que contribuíram para a realização deste trabalho, com discussões e dicas que proporcionaram vários insights e bons momentos de descontração.

Aos professores e funcionários do PPGCC e do SPLab. Aos JML experts que contribuíram com o experimento para avaliar o conjunto de heurísticas que propusemos no processo de categorização de não-conformidades. À CAPES pelo apoio e suporte financeiro fornecidos para o desenvolvimento deste trabalho. E por fim, a todas as pessoas que contribuíram, direta ou indiretamente para o desenvolvimento deste trabalho.

Conteúdo

1 Introdução **1**

1.1 Problem 2

1.1.1 Motivating Example 2

1.1.2 Relevance 3

1.2 Solution 5

1.3 Evaluation 7

1.4 Summary of Contributions 9

1.5 Outline of the Dissertation 10

2 Fundamentação Teórica **12**

2.1 Formal Methods 12

2.1.1 Formal Methods – Practice 13

2.2 Design by Contract 14

2.2.1 DBC – Concept 14

2.2.2 Eiffel 15

2.2.3 Spec# 16

2.2.4 JML 17

2.2.5 Conformance 18

2.3 Software Testing 20

2.3.1 Test Cases 21

2.3.2 Oracles in DBC Context 21

2.3.3 Tests Generation 22

2.3.4 Conformance Testing 24

3 Uma abordagem para Detecção e Categorização de Não-conformidades em Programas Baseados em Contratos 27

3.1 Randomly-Generated Tests Approach for Conformance Checking 28

3.1.1 Step 1 - Tests Generation 28

3.1.2 Step 2 - Oracle Generation 30

3.1.3 Step 3 - Tests Execution 31

3.1.4 Step 4 - Grouping distinct Nonconformances 31

3.1.5 Step 5 - Results from Detection Phase 31

3.2 Heuristics-based Approach for Categorize Nonconformances 32

3.2.1 Categorization Model 32

3.2.2 Categorization Overview 35

3.2.3 Heuristics 36

3.3 JMLOK 2.0 38

3.3.1 Step 1 - User interaction 40

3.3.2 Step 2 - The Controller module 40

3.3.3 Step 3 - The Detector module 40

3.3.4 Step 4 - The Categorizer module 43

3.3.5 JMLOK 2.0 Architecture 45

3.3.6 Limitations 46

4 Avaliação da Geração Aleatória de Testes para Detecção de Não-Conformidades em Programas Baseados em Contratos 48

4.1 Research Questions 48

4.2 First study - RGT in Isolation 50

4.2.1 Setup 50

4.2.2 Results 51

4.2.3 Discussion 53

4.3 Second Study - JMLOK 2.0 versus JET 57

4.3.1 Setup 57

4.3.2 Results 57

4.3.3 Discussion 58

4.4 Threats to validity . . . 61
4.5 Answers to the research questions . . . 61

5 Avaliação da Categorização de Não-Conformidades 63

5.1 Research Questions . . . 63
5.2 First Study - Comparison between Manual and Automatic Categorization . 65
5.2.1 Setup . . . 65
5.2.2 Results . . . 65
5.2.3 Discussion . . . 67
5.3 Second Study - Comparison between Automatic and JML Experts Categorization . . . 69
5.3.1 Setup . . . 69
5.3.2 Results . . . 70
5.3.3 Discussion . . . 72
5.4 Threats to Validity . . . 73
5.5 Answers to the research questions . . . 73

6 Considerações Finais 75

6.1 Conclusions . . . 75
6.2 Related Work . . . 77
6.2.1 Conformance Checking . . . 77
6.2.2 Categorization of Contract Violations . . . 79
6.2.3 Automatic Test Generation . . . 80
6.3 Future Work . . . 81

A Manual Categorization Results on Contract-based Programs 91

B Automatic Categorization Results on Contract-based Programs 93

C Numbered Nonconformances 95

D Form to Evaluation of the Categorization Model 97

Lista de Figuras

1.1 The steps performed in our approach: 1- The code of the program is compiled. 2- Tests are generated with the compiled code in step 1. 3- Oracles are produced from contracts. 4- Tests generated in step 2 are run against oracles produced in step 3. 5- A filter distinguishes from all failures which distinct nonconformances were detected. 6- A subset of heuristics to each nonconformance is selected based on its type. 7- Each subset is used together the source code to choose a likely cause for the nonconformance. And finally, all detect and categorized nonconformances are returned. 11

2.1 The Randoop test generation process. The process starts when are given the following inputs: a list of classes under test, a time limit, and optionally a set of properties to check. Then sequences of method calls are generated, executed and examined; the feedback from the execution feeds back the process until the time limit be reached. 24

3.1 The internal structure of the proposed RGT-based approach for nonconformances detection. As inputs are given the contract-based program, a directory of external libs (optional) and the time to tests generation (also optional, default value = 10 seconds). 1- Tests are generated into the RGT engine. 2- Oracles are produced from contracts. 3- Tests generated in step 1 are run against oracles produced in step 2. 4- A filter distinguishes from all failures which distinct nonconformances were detected. 5- The list of detected nonconformances is returned. 29

3.2 Grouping nonconformances – this process receives as input a list of several failures (some possibly equal), then the failures are compared and the filter returns a set of distinct nonconformances (distinct faults). 32

3.3 Internal structure of the approach for nonconformances categorization. As input are given the list of nonconformances and the source code. 1- a subset of heuristics to each nonconformance is selected based on its type. 2- Each subset is used together the source code to choose a likely cause for the nonconformance. 3- The list of categorized nonconformances is returned. . 35

3.4 Heuristic for nonconformances from precondition type. 36

3.5 Heuristics for nonconformances from postcondition and evaluation types. . 37

3.6 Heuristics for nonconformances from invariant type. 38

3.7 Heuristics for nonconformances from constraint type. 39

3.8 The architecture of JMLOK 2.0 tool. There are 4 modules in this tool: UI, Controller, Detector, and Categorizer. 39

3.9 The initial screen of the tool. In this screen the user can give the inputs needed to use the tool. If no input is given, a message dialog warns the user about the needed of at least the contract-based program to be given as input. 40

3.10 The screen resultant of Detection phase. In order to get the results of the categorization, the user have to press the button Nonconformances. 44

3.11 Categorization screen presenting the results of JMLOK 2.0 tool. For each nonconformance are presented: the type, the complete location (including information of package, class, and method), the suggested likely cause, and the test case with highlight in the line that reveals the nonconformance. . . 45

4.1 Nonconformances distributed between their types, for all experimental units. 53

4.2 Nonconformances distributed between their likely causes, for all experimental units. 55

Lista de Tabelas

3.1 Categorization model for nonconformances in contract-based programs. . . 33

4.1 Programs characterization in terms of Lines of Code (LOC) and Lines of JML specification (LJML). 51

4.2 For each experimental unit we present the number of generated test cases, test coverage, and all nonconformances detected, grouped by their types. . . 52

4.3 Likely Causes from detected nonconformances in the experimental units. . 54

4.4 Results of mean for *breadth* and *depth* metrics to each experimental unit and for all nonconformances detected. 54

4.5 Comparison between JMLOK 2.0 and JET. For each experimental unit is presented the number of tests generated by each tool, test coverage (Java and JML), number and types of nonconformances. 58

5.1 *matches* results for each experimental unit and the mean of this metric. The metric was obtained using the Equation 5.1. 66

5.2 Metrics *matches* and *depth* for each experimental unit. For Samples we calculate the metrics for each package used – BoundedStack, stacks, dbc, misc, and list. The *Spearman's* coefficient (last line of table) indicates a strong negative relation between these metrics. 67

5.3 Randomly selected nonconformances released to JML experts categorize. The nonconformance number corresponds to the position of the nonconformance considering our 84 nonconformances – the counting starts in sample programs (`BoundedStack`) and continues until the last nonconformance discovered in `TransactedMemory` unit. Experimental Unit gives the name of the experimental unit. Class and Method columns give information about location of the nonconformance into the experimental unit. Finally, column Type gives the type of the nonconformance. 70

5.4 Column # NC displays the nonconformance number (the number is the same presented on Table 5.3). Column Type shows the nonconformances type. For each Subject we present the Category and Likely Cause for each categorized nonconformance. Finally, Automatic results shows the results of our automatic categorization for each categorized nonconformance. The line *matches* presents the *matches* metric for each Subject in comparison with tool results. We use the following acronyms for type: pre for precondition, post for postcondition, and inv for invariant; and for category: Spec for Specification error, Code for Code error, and Undef for Undefined. 71

6.1 Related Work about conformance checking. 79

6.2 Related Work about categorization approaches for contract violations. . . . 80

6.3 Related Work about automatic test generation. 81

A.1 Results of the manual categorization process on Sample Programs. Column Experimental Unit shows the name of the experimental unit. Columns Class and Method display the names of the class and the method, respectively, where a nonconformance was detected. Column Category presents the category manually assigned to the discovered nonconformance. Column Type exhibits the nonconformance's type. Finally, column Likely Cause reveals the likely cause manually assigned. 91

A.2 Results of the manual categorization process on Open-source Programs. Column Experimental Unit shows the name of the experimental unit. Columns Class and Method display the names of the class and the method, respectively, where a nonconformance was detected. Column Category presents the category manually assigned to the discovered nonconformance. Column Type exhibits the nonconformance's type. Finally, column Likely Cause reveals the likely cause manually assigned. 92

B.1 Results of the automatic categorization process on Sample Programs. Column Experimental Unit shows the name of the experimental unit. Columns Class and Method display the names of the class and the method, respectively, where a nonconformance was detected. Column Category presents the category manually assigned to the discovered nonconformance. Column Type exhibits the nonconformance's type. Finally, column Likely Cause reveals the likely cause automatically assigned. 93

B.2 Results of the automatic categorization process on Open-source Programs. Column Experimental Unit shows the name of the experimental unit. Columns Class and Method display the names of the class and the method, respectively, where a nonconformance was detected. Column Category presents the category manually assigned to the discovered nonconformance. Column Type exhibits the nonconformance's type. Finally, column Likely Cause reveals the likely cause automatically assigned. 94

C.1 Numbered nonconformances for Sample Programs. 95

C.2 Numbered nonconformances for Open-source Programs. 96

Lista de Códigos Fonte

1.2 A test case that reveals the nonconformance present into GenCounter class . 3

1.1 GenCounter and MapMemory classes 4

2.1 Example of DBC in Eiffel 15

2.2 Example of DBC in Spec# 17

2.3 Example of DBC in JML 18

2.4 Test case that shows a nonconformance in updateCount method. 19

2.5 Code that presents a nonconformance in the client side – Case 2. 20

2.6 Oracle generated to updateCount method 23

2.7 Test case generated by Randoop 25

3.1 Test generated by our approach from motivating example 29

3.2 Test case that reveals a nonconformance that we categorize as specification error 34

3.3 Test case that reveals a nonconformance that we categorize as code error . . 34

3.4 Test case that reveals a nonconformance that we categorize as undefined error 35

Capítulo 1

Introdução

Engenharia de Software [68] é um tema da Ciência da Computação que lida com todos os aspectos relacionados com a produção de software – da fase de especifição do sistema até a evolução e manutenção após o software estar em uso. Sistemas de software são abstratos e intangíveis. Eles não são restringidos pelas propriedades dos materiais, governados por leis físicas ou por processos de manufatura [68]. Isto pode simplificar a engenharia de software, uma vez que não há limites naturais para o potencial do software. Contudo, por conta da falta de restrições físicas, os sistemas de software podem rapidamente se tornar complexos, difíceis de entender e caros para sofrerem modificações. Hoje em dia os sistemas de software estão presentes em nossas vidas, desde programas simples que usamos para edição de documentos, a programas complexos para controle de tráfego aéreo, controle de processos industriais, ou até mesmo sistemas que controlam o nível de medicamentos a serem injetados em nossos corpos. Deste modo, a busca por confiabilidade em sistemas de software tem aumentado e ganhado a atenção de diversas pessoas e organizações.

Programas baseados em contratos [34], uma solução baseada em linguagem na qual código e contratos são integrados em um único artefato, desempenham um importante papel no contexto do desenvolvimento de sistemas de software confiáveis e de qualidade. Neste cenário, a metodologia *Design by Contract* (DBC) [48] – Programação por Contratos – impõe contratos (invariantes, pré-condições e pós-condições) que expressam direitos e obrigações para módulos clientes e fornecedores. Os contratos regulando o comportamento do código correspondente provê dados adicionais para verificação de conformidade. No contexto do desenvolvimento de programas Java, a linguagem *Java Modeling Language* (JML) [43] é uma notação para habilitar o uso de DBC (e correspondente conjunto de ferramentas), com contratos como comentários junto ao código Java.

O restante deste capítulo tem a seguinte estrutura: primeiro apresentamos o problema que motivou o desenvolvimento do presente trabalho (Seção 1.1); na Seção 1.2 discutimos acerca da solução proposta para o problema; posteriormente na Seção 1.3 mostramos a avaliação realizada; então na Seção 1.4 resumimos as principais contribuições deste trabalho; e por fim na Seção 1.5 mostramos a estrutura desta Dissertação.

1.1 Problem

With contracts, early detection of nonconformances is highly desirable, as developers are able to provide a more reliable account of correctness and robustness of the software written [48]. Developers tend to apply automated approaches, although incomplete, as verification by formal proofs is hard to scale.

For JML, there are basically two ways to automatically check conformance: statically, with tools such as ESC/Java [29], ESC/Java2 [23], LOOP [7], and JACK [4]; and dynamically, with tools such as JMLUnit [18], JMLUnitNG [74], JET [16], Korat [11] and FAJITA [1].

While static analysis tools can be useful for diagnosing a number of common errors (such as null dereferences and invalid accesses to arrays), they may be limited for detecting general nonconformances (those only arise in the runtime environment), furthermore they can produce false positives and false negatives. Test-based approaches present, on the other hand, lower costs and higher precision in detecting conformance problems. Nevertheless, those approaches present a number of limitations, mostly by falling short in providing (1) effective and automatic test data generation; (2) comprehensive unit tests that fully exercise sequences of calls to unveil subtle nonconformances (as seen in the example from Section 1.1); and (3) a classification for the detected nonconformances.

1.1.1 Motivating Example

In this section, we present an example to illustrate the problem of conformance checking in a contract-based program. In JML, contracts are written as qualified comments (Source

Code 1.1). The example is adapted from the experimental unit `TransactedMemory` (Section 4.2.1) – visibility is omitted, for simplicity.

`GenCounter` represents a piece of information about some named tag, while `MapMemory` represents a Java implementation of memory for smart cards. In our adaptation, these classes have a constructor and two methods: one for updating and another for resetting values. JML method contracts are declared with keywords `requires` and `ensures`, specifying pre- and postconditions, respectively. The `invariant` clause must hold after constructor execution, and before and after every method call. The invariant in `GenCounter` enforces that field `cntGen` must be in range [0, `MapMemory.MAX`]. The `\old` clause is used to refer to pre-state of some value, the used in the postcondition refers to pre-state value of `cntGen`.

The program in Source Code 1.1 is not in conformance with its contracts. `GenCounter` presents one nonconformance that can only be detected with a sequence of three calls to `MapMemory.updateMap` with parameter *m* = *true*. In Source Code 1.2, a test case reveals this problem. This problem may be solved by adding a precondition to `GenCounter.updateCount`, testing whether the value of `cntGen` is less than `MapMemory.MAX`. Regardless of where the bug is located (contract or code, or both), the failure may only arise within a sequence of calls to two or more methods, called in a particular order. Therefore, nonconformances between contract and implementation may be subtle to detect. And manually-provided test cases or data have a considerable low probability of detecting this kind of nonconformance.

Código Fonte 1.2: A test case that reveals the nonconformance present into GenCounter class

```java
MapMemory mm = new MapMemory();
mm.updateMap(true);
mm.updateMap(true);
mm.updateMap(true);
```

1.1.2 Relevance

Since we use contract-based programs with the aim of obtaining software quality, any contract violation must be detected and corrected to software quality be maintained and ensu-

Código Fonte 1.1: GenCounter and MapMemory classes

```
class GenCounter {
  //@ invariant 0 <= cntGen && cntGen <= MapMemory.MAX;
  int cntGen;
  GenCounter() {
    cntGen= 1; }
  //@ ensures (b == true)==>(cntGen == \old(cntGen+1));
  void updateCount(boolean b){
    if(b){ cntGen++; }
  }
  //@ ensures cntGen == 0;
  void resetCount(){
    cntGen= 0;
  }
}

class MapMemory {
  final static int MAX = 3, MSIZE = 10;
  GenCounter g;
  boolean[] map;
  int pos;
  MapMemory(){
    g = new GenCounter();
    map = new boolean[MSIZE];
    pos= 0; }
  //@ requires pos < MSIZE-1;
  void updateMap(boolean m){
    map[pos++] = m;
    g.updateCount(m); }
  //@ ensures pos == 0;
  void resetMap(){
    map = new boolean[MSIZE];
    g.resetCount();
    pos= 0;
  }
}
```

red [48]. Furthermore, considering the software life-cycle, an early detection and correction of a fault is very important because reduces the cost of correction and maintains the software quality [68]. Regardless of where the bug is located (contract or code, or both), some contract violations may only arise within a sequence of two or more method calls, called in a particular order. Therefore, nonconformances between contract and implementation may be subtle to detect, and manually-provided test cases or data, like in JMLUnit [18] or in JMLUnitNG [74] approaches or test cases with a single method call, like in JET [16] approach, have a considerable low probability of detecting this kind of nonconformance. Thus, an approach that automatically detect and suggests a likely cause for contract violations can be useful to software quality maintenance.

1.2 Solution

In this work, we propose and implement a RGT-based (Randomly-Generated Tests) approach to detection and a model to categorize nonconformances in contract-based programs. Our approach aims to help the programmer in the process of nonconformances correction.

In order to investigate the conformance of the contract-based programs, our approach automatically generates and executes tests, comparing the test results with oracles (generated from the contracts). The generated tests are composed basically of sequences of calls to methods and constructors under test. The test oracles are assertions from the contracts. Some tools like *jmlc*, *OpenJML* – for JML, *AutoTest* – for Eiffel; perform this kind of transformation. So, the assertions present in the contracts are used as oracles to the test results. After tests execution, the approach applies two filters: the first to distinguish test results between *meaningless* [19] – meaningless are tests that violate preconditions in method entries (preconditions violated directly by the test case) because the test generator approach does not consider specifications in the process of test generation – and *failures* (contract violations, inconsistencies between test results and oracles); the second to distinguish from all failures those that are distinct nonconformances (faults). The second filter returns the distinct nonconformances to be used in the process of automatic categorization of nonconformances.

Regarding to nonconformances categorization, we propose a three-level model composed by a category, a type and a likely cause. This model is implemented in a heuristics-based approach to automatically suggests a categorization for nonconformances. The category corresponds to the artifact in which probably occurs the nonconformance – source code or contract. The type is given automatically by the assertion checker, and corresponds to the part of JML that was violated - considering only visible behavior from the systems. The suggested likely cause is given by specific heuristics derived from our experience in investigate likely causes for nonconformances. Each heuristic is based on a set of possible errors that can induce to a nonconformance revelation and it is related to the type of detected nonconformance. In our implementation we use a set of heuristics and the contract-based program – the source code and its contract. Based on the contract-based program, the nonconformance type and the corresponding set of heuristics; a likely cause is suggested to the nonconformance. For example, regarding an invariant error, we suggest a likely cause following the heuristics aforementioned: first check whether there are some field from the class that is not initialized into the constructor, the likely cause suggested is *Code error*; otherwise, check whether there is the default precondition, or nothing, or whether there is at least one field modified on method body; in either case, the likely cause suggested is *Weak precondition*; otherwise *Strong invariant* is the suggestion. From the example in Section 1.1.1, in which there is a nonconformance of invariant type, once the method `GenCounter.updateCount` does not have an explicit precondition (it receives the default *true*) and the likely cause suggested is *Weak precondition*. After the categorization, a set of categorized nonconformances is returned to the user.

JMLOK 2.0 is our implementation of this approach in the context of Java/JML programs (JMLOK 2.0 is an improvement of JMLOK [71]). While JMLOK was able to generate tests and displays the test results between *meaningless* and *relevant*; JMLOK 2.0 is able to detect and categorize nonconformances and to display for the user only the distinct nonconformances detected and categorized. Our detection does not present for the user false positives; whereas JMLOK presents. Furthermore, the user interface was improved in the new version of the tool, providing to the user more information about the nonconformance that was detected. JMLOK 2.0 was developed following an adaptation of MVC pattern: there is a view module – the User Interaction; a Controller module – that intermediates the communication between the View and the Model; and two modules that composes the Model, one module to nonconformances detection and one module to nonconformances categorization.

In detection module of the tool, the test generation is performed automatically and randomly by Randoop [55] tool. Randoop is a feedback-directed test generator tool; and the tests generated are composed by sequences of calls to methods and constructors under test. We chose Randoop as RGT engine because this tool generates several sequences of calls to the object under test in a given time limit. The oracles generation is performed by *jmlc* compiler and JUnit is the framework used to tests execution. Afterwards the execution, two filters are past: one to separate *meaningless* from relevant tests (tests that can reveals nonconformances – failures); the other to get only distinct nonconformances from all relevant tests (get distinct faults from all failures). After these filters the set of distinct nonconformances is returned to the categorization module. In the categorization module, the contract-based program and a set of heuristics are used to suggest a likely cause for each nonconformance. Subsequently the categorization, the results (the list of categorized nonconformances) are sent to Controller module; then Controller sends the results to be presented for the user in the UI. Figure 1.1 presents the steps performed in our approach for detect and categorize nonconformances in contract-based programs.

1.3 Evaluation

We evaluated our approach in two experiments [5]. First we evaluate our detection approach and our manual categorization process in open-source contract-based programs (Chapter 4). The experimental units consist of sample programs available in the JML web site[1] and programs collected from some open-source JML projects. Second we evaluate our implementation of the categorization model by means of a module to JMLOK 2.0 in the same contract-based programs. Furthermore, we asked voluntary JML experts to manually categorize 10 nonconformances (randomly selected using the R statistical tool [10]) and compare their results with our heuristics to evaluate our model (Chapter 5).

In the first experiment, we observed that most of detected nonconformances in our experimental units are related to *postcondition errors* with likely causes stay between *Weak preconditions* (mostly related to the lack of preconditions for the methods) and *Code errors*

[1] `http://www.eecs.ucf.edu/~leavens/JML/examples.shtml`

(mostly related to null fields). We also found that most nonconformances are hard to detect without sequences of modifications into the object under test, with the results of metrics *breadth* and *depth*. In this experiment we performed two experimental studies (studies one and two). In study one, six open-source contract-based programs and a set of sample programs (Section 4.2.1) amounting to 29 KLOC and 9 K lines of JML contracts (that we will refer as KLJML henceforth) were subject to JMLOK 2.0, which detected 84 nonconformances in total. In addition, we classified the detected nonconformances in terms of category, type and their likely cause, employing our categorization model (Section 3.2.1). Most nonconformances were postcondition violations, and causes often fall between weak preconditions and code errors. Even in small examples, developed for JML training, nonconformances were found – mainly problems in contracts. We also observed that the *breadth* (position of the top-level test case method call within which the violation occurs) and the *depth* (number of internal calls until the nonconformance occurs) of test execution that discovers nonconformances are, in average, 2.54 and 2.23, respectively, showing evidence for the need of a more complex generated test structure in nonconformance detection than only one modification in the object under test.

In study two, using a subset of programs from the first study (details in Section 4.3), the JET tool unveiled 9 nonconformances with Java instructions coverage of 56.97% and JML instructions coverage of 47.97%, while the JMLOK 2.0 tool detected 30 by covering 78.44% of Java instructions and 67.67% of JML instructions; on the same experimental units. We compared the tools concerning (1) the number of detected nonconformances and (2) total block instructions coverage by tests, and found that the JMLOK 2.0 tool performs better for both criteria. Furthermore, nonconformances detected by JET differ between repeated executions, maybe due to the nature of its genetic algorithm, which is not observable in JMLOK 2.0.

In the second experiment, we observed that although the automatic categorization had good results in comparison with our manual results, in some cases only a manual inspection of the code and specification is able to determine what caused the nonconformance. In this experiment we also performed two others experimental studies (studies three and four).

In study three, we propose and collect a metric to compare our automatic categorization with our manual results: *matches* – the ratio between the number of coincidences between manual and automatic categorization and the number of total categorized nonconformances –, the metric value was 0.73, indicating that there is a good ratio of coincidences between the two approaches to categorize nonconformances.

In study four, we asked voluntary JML experts to categorize 10 nonconformances randomly selected and compare their results with ours. In this study we observed that our heuristics-based approach is a little bit similar to results from manual analysis of voluntary JML experts.

1.4 Summary of Contributions

The main contributions of this work are the follows:

- An approach to suggests a categorization for nonconformances in contract-based programs;
- An implementation of this approach, and an improvement of the conformance checking performed by JMLOK – in JMLOK 2.0;
- Analysis of the most frequent nonconformance types and likely causes in our experimental units;
- Analysis of *breadth* and *depth* of test execution to detect nonconformances in our experimental units;
- A comparison of two test-based approaches to conformance checking in open-source contract-based programs;
- Analysis of *matches* between our classification model and our automatic categorization approach for nonconformances in our experimental units;
- A comparison between our automatic categorization approach and the categorization performed by JML experts in contract-based programs.

1.5 Outline of the Dissertation

The remaining parts of this document are structured as follows:

Chapter 2: Background This chapter provides the theoretical background necessary to understand this work. Some concepts from Formal Methods, Design by Contract and Software Testing are presented.

Chapter 3: An Approach for Detection and Categorization of Nonconformances in Contract-Based Programs This chapter presents in details our approach for detection and categorization of nonconformances in contract-based programs.

Chapter 4: Evaluating Random Test Generation for Detecting Nonconformances in Contract-Based Programs This chapter presents an evaluation of our approach to conformance checking together our manual categorization of nonconformances detected, we also present an analysis of *breadth* and *depth* of test execution to detect nonconformances. Furthermore, a comparison with a tool to conformance checking is performed in terms of number of detected nonconformances and test coverage. Automatic categorization of nonconformances is not considered yet.

Chapter 5: Evaluating Categorization of Nonconformances This chapter presents an evaluation of our automatic categorization approach in comparison with our manual categorization model (baseline); we also present a comparison between our categorization and the categorization performed by voluntary JML experts.

Chapter 6: Conclusions This final chapter presents conclusions, related works, and prospects for future work.

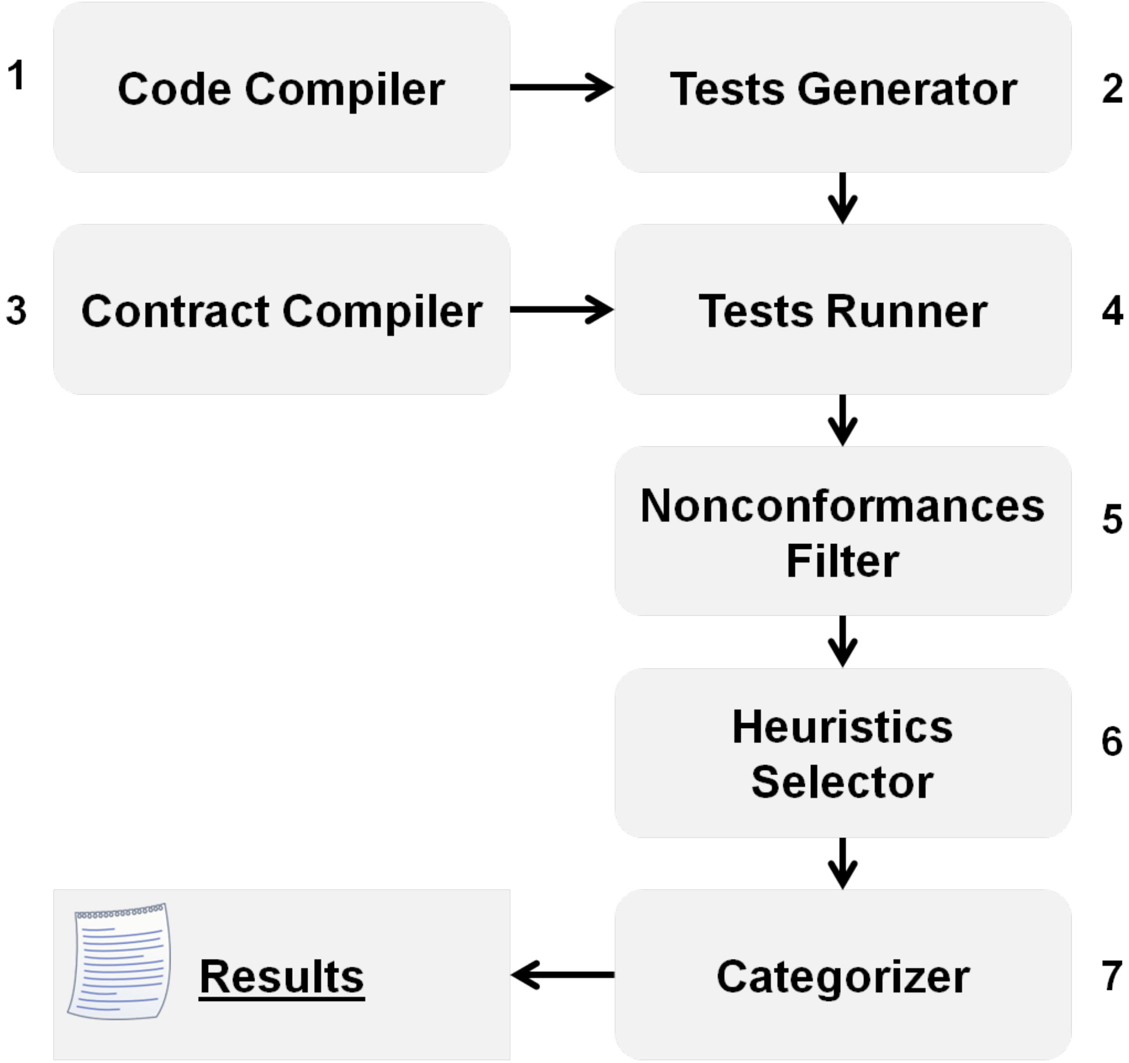


Figura 1.1: The steps performed in our approach: 1- The code of the program is compiled. 2- Tests are generated with the compiled code in step 1. 3- Oracles are produced from contracts. 4- Tests generated in step 2 are run against oracles produced in step 3. 5- A filter distinguishes from all failures which distinct nonconformances were detected. 6- A subset of heuristics to each nonconformance is selected based on its type. 7- Each subset is used together the source code to choose a likely cause for the nonconformance. And finally, all detect and categorized nonconformances are returned.

Capítulo 2

Fundamentação Teórica

Este capítulo tem como objetivo principal prover a fundamentação teórica necessária para o entendimento dos principais conceitos discutidos nesta dissertação. Serão apresentados conceitos de Métodos Formais, com ênfase na metodologia *Design by Contract* e em verificação de conformidade dos contratos; e conceitos de Teste de Software, com ênfase em geração de testes.

2.1 Formal Methods

According to Gibbins [33], a formal system is a system whose notation and manipulation rules are well-defined and based on mathematics theory. A formal method is a set of engineering rigorous practices that are based on formal systems and applied to development of engineering products, like software and hardware [33]. A formal language with a precise and unambiguous semantic is needed as base to use a formal method.

Formal methods gathered more emphasis after the 'software crisis' [68], once formal methods have a mathematical approach that allows the developers to specification, development, and verification of a system. Furthermore, with the use of logic concepts from works such as Hoare [38] or Dijkstra [27], it is possible to prove the correctness of a program using the same precision of mathematical theorems.

Despite the usefulness of formal methods, they have some limitations, that already were known by Hoare [38]. One of the main is the limitation on proofs – a proof is a demonstration that one formal statement follows from another, however, the real world is not a formal system, so a proof does not show that, in real world, things will happen as we expect [36]. Another important limitation is that mistakes can be made – even within formalism, we can make mistakes in doing proofs [36].

Concerning in languages used to formally specify systems, we present in this section some languages and discuss the main features of a language for formal specification. Formal specification languages follow basically two approaches: an algebraic approach, used to describe the system in terms of operations and their relationships; and a model-based approach, used to construct a model from the system with mathematical constructions, like set and sequences; and in which operations are defined by the way they modify the system. Some well known formal languages and methods are: Larch family [42], [35], Z specification language [72] and Vienna Development Method (VDM) [9], [52].

2.1.1 Formal Methods – Practice

Woodcock et al. [73] present a study about the use of formal methods in industry. They show the main benefits of formal methods application and show that, although formal methods are not widely used, the use has been increasing mainly after the creation of the *Verified Software Initiative*[1]. Despite the benefits of formal methods to software development, their use in industry is not widely spread yet, maybe because formal methods are hard to scale for big systems. But according to Clarke and Wing [21] this scenario is changing in the last years through the tools improvement and development to support specification and verification. Also according to Clarke and Wing [21], the use of formal methods has been intensified in companies like *Microsoft*, *Praxis*, *Intel*, *Cisco*, *Sony Co.*, *IBM*, *Rolls-Royce*, and *Cadence*. Furthermore, methodologies like Design by Contract [46] helps to increase the use of formal methods in software development.

Once testing is a common practice to get some confidence about the program behavior and, on the other hand, formal methods have been used to formally check the software correctness; Hierons et al. [37] present an investigation about the benefits from the use of formal methods together with tests. Following ideas from the Design by Contract methodology the alignment between formal methods and tests has increased mainly by the use of tools that

[1] https://sites.google.com/site/verifiedsoftwareinitiative/

generate tests to conformance check between programs and their contracts (formal specifications).

2.2 Design by Contract

In this section we present Design by Contract in details and discuss about this methodology for Eiffel, Java, and C# languages. Furthermore, we present the concept of conformance.

2.2.1 DBC – Concept

Contract-based programs [34] incorporate a language-based solution that integrates contracts and code into a single artifact. In this scenario, the Design by Contract (DBC) [46] is an approach that arose from formal methods, more specifically from Hoare's triples [38]. This methodology was developed with the aim of helping to construction of reliable and quality software. The fundamental idea from this methodology is the existence of contracts (invariants, pre- and postconditions) between modules of a system that establishes rights and obligations for both parts: clients and suppliers. So, clients have to guarantee some conditions before call a supply; suppliers, on the other hand, have to guarantee some properties (results from their executions) for their clients.

With DBC methodology we have the advantage that contracts are executable; so, they can be executed to conformance check, allowing the responsibility attribution of contract violation – if the violation occurred in client side or in supplier side. Furthermore, contracts present abstractions of the methods behavior. In general, contracts are written in the same language of program source code.

The notation of Hoare's triple used in contracts: **P {Q} R**, means that there is a required connection between a precondition (P), a program (Q) and a description of the result of its execution (R). According to Hoare [38], this may be interpreted as: "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion".

The main languages that implements the DBD methodology are: Eiffel [47], Spec# [3] and JML [43]. In the followings sections we present some details about each language.

2.2.2 Eiffel

The Eiffel language [47] was created by Bertrand Meyer in mid-1980s, as an object-oriented programming language focused on development of software quality. The language was used by Meyer [48] to illustrate the fundamental concepts from Design by Contract methodology - like pre- and postconditions and invariants. The language syntax is very similar to Pascal[2] and ALGOL[3] syntax. Eiffel method contracts are declared with keywords *require* and *ensure*, specifying pre- and postconditions, respectively. A class *invariant* clause must hold after constructor execution, and before and after every method call. The *old* clause is used to refer to pre-state of some value.

In Source Code 2.1 we present an example of a program written in Eiffel to specify the method `updateCount` from class `GenCounter`, one class from the motivating example (Section 1.1.1) – the remainder of the example is specified in a similar way to JML specification. In this example, the postcondition – clause *ensure* declares that the value of `cntGen` field must be increased in one if `b` parameter is equals `True`, and the *old* clause used in the postcondition refers to pre-state value of `cntGen`.

Código Fonte 2.1: Example of DBC in Eiffel

```
class GenCounter
  feature
  cntGen: INTEGER is 0

  updateCount(b: BOOLEAN) is
    do
      if b = True
        then cntGen := cntGen + 1
    ensure
      b = True implies cntGen = old cntGen + 1
end
```

Concerning tool support, AutoTest [49] is a collection of tools that automate the testing

[2] http://www.pascal-programming.info/index.php

[3] http://groups.engin.umd.umich.edu/CIS/course.des/cis400/algol/algol.html

process for Eiffel programs. In AutoTest the contracts are used as oracles to expected outputs for conformance checking of the programs; furthermore, AutoTest uses a randomly-guided tests generation (ARTOO [20]) and supports mixing manual and automated test. Besides AutoTest, there is the EiffelStudio[4]. EiffelStudio is an Integrated Development Environment (IDE), a software application that provides comprehensive facilities to computer programmers for software development, powered by the Eiffel language.

2.2.3 Spec#

Spec# [3] is an extension to the object-oriented language C# to support Design by Contract. Spec# extends the system of types from C# to include non-null types and verified exceptions. Furthermore, provides constructions to contract specifications for methods as preconditions, postconditions, and invariants. Spec# method contracts are declared with keywords *requires* and *ensures*, specifying pre- and postconditions, respectively. A class *invariant* clause must hold after constructor execution, and before and after every method call. The *old* clause is used to refer to pre-state of some value.

The development of the language had a similar purpose of the development of Eiffel, the main motivation to Spec# development was quality software construction with a viable cost.

In Source Code 2.2 we present the implementation of `updateCount` method from `GenCounter` class using Spec# language. The postcondition – clause *ensures* declares that the value of `cntGen` field must be increased in one if `b` parameter is equals `true`; the *old* clause used in the postcondition refers to pre-state value of `cntGen`.

Concerning tool support, Spec# compiler [3] is integrated into the Microsoft Visual Studio development environment for the .NET platform. The compiler statically enforces non-null types, emits run-time checks for method contracts and invariants, and records the contracts as metadata for consumption by downstream tools. And Boogie [2] is the Spec# static program verifier, this tool generates logical verification conditions from a Spec# program. Internally, Boogie uses an automatic theorem prover that analyzes the verification conditions to prove the correctness of the program or find errors in it.

[4] `https://www.eiffel.com/eiffelstudio/`

Código Fonte 2.2: Example of DBC in Spec#

```
class GenCounter{
  int cntGen = 0;
  public void updateCount(bool b)
  ensures (b == true) ==> (cntGen == old(cntGen) + 1);
  {
    if(b){
      cntGen++;
    }
  }
}
```

2.2.4 JML

In the context of Java development, the Java Modeling Language (JML) [43] is a DBC-enabling notation (and corresponding toolset), with contracts as comments within Java code. JML has syntax very similar to Java, furthermore, extends some Java expressions (e.g. the use of quantifiers) to specify behaviors and has some restrictions about Java constructions like: side-effects, generic types, and Java annotations. JML mixes DBC approach from Eiffel [47] with the specification model-based approach from Larch family [35] of programming languages, and some elements of calculus of refinement.

JML method contracts are declared with keywords `requires` and `ensures`, specifying pre- and postconditions, respectively. A class *invariant* clause must hold after constructor execution, and before and after every method call. A history constraint – *constraint* clause is similar to invariants, but constraints define relationships that must hold for the combination of each visible state and the next in the program's execution. A \old clause is used to refer to pre-state of some value.

In Source Code 2.3 we present the implementation of the `updateCount` method from `GenCounter` class using JML language. In JML the postcondition is represented with the clause `ensures`; the `\old` clause used in the postcondition refers to pre-state value of `cntGen`. The *spec_public* clause is used to declares that the private field `cntGen` is publicly visible in the specification context. JML has many other elements in addition to preconditions, postconditions, invariants, and constraints; the complete list of JML elements is available at JML Reference Manual [44].

Código Fonte 2.3: Example of DBC in JML

```
public class GenCounter {
  private /*@ spec_public @*/ int cntGen = 0;
  //@ ensures (b == true)==>(cntGen == \old(cntGen+1));
  public updateCount(boolean b){
    if(b){
      cntGen++;
    }
  }
}
```

Concerning tool support for JML, there are basically three kinds of tools: runtime assertion checkers (RAC) or JML compilers – like *jmlc* [17], *jml4c* [64], *OpenJML* [22], or *ajmlc* [61]; dynamic and static conformance checking tools – like JMLUnit [18], JMLUnitNG [74], JET [16], for dynamic conformance checking; and ESC/Java [29], ESC/-Java2 [23], LOOP [7], JACK [4] for static conformance checking. With regard to JML compilers, *jmlc* is like a Java compiler but add assertions into bytecode from contracts in source code (contracts like preconditions, postconditions, invariants and history constraints). *jml4c* is a JML compiler built by extending the Eclipse Java compiler; this compiler supports Java 5 features such as generics. And *OpenJML* is the new compiler of JML, this compiler is in development yet and intends to support the new features of Java language. As another point of view, *ajmlc* is a seamless aspect-oriented extension to the JML design by contract language, compatible with AspectJ. *ajmlc* cleans modularization/specification of crosscutting contracts, such as preconditions and postconditions, while preserving documentation and modular reasoning.

2.2.5 Conformance

With the DBC methodology arises the concept of conformance [16],[18] – when the code does what the contract declares, in other words, the code satisfies its contract. When the conformance is broken there is a contract violation – called in the literature as nonconformance [16]. Once in DBC contracts, invariants, preconditions, and postconditions can be specified in a way that can be verified by a compiler, any contract violation between client and supplier modules can be detected immediately, allowing the construction of more reliable systems.

A nonconformance can occurs in two cases:

1. When the client guarantees the preconditions and the supplier does not guarantee their postconditions – a nonconformance in supplier side;

2. When the client does not guarantee the precondition from the supplier – a nonconformance in client side.

For example, the code presented in Source Code 1.1 is not in conformance with its contract (Case 1 of nonconformance – the supplier does not return the expected result to its client). Consider the following sequence of calls (Source Code 2.4):

Código Fonte 2.4: Test case that shows a nonconformance in updateCount method.

```
MapMemory mm = new MapMemory();
mm.updateMap(true);
mm.updateMap(true);
mm.updateMap(true);      // Contract violation here.
```

After three calls to `updateMap` method with *true* as parameter the invariant from GenCounter class is violated. The problem occurs because the supplier allows that the client to call `updateMap` several times and does not check anything about this call – in this case the precondition from `updateCount` is *true*, meaning that all clients are accepted, and nothing is required from clients.

As an example of nonconformance from Case 2, consider the Source Code 2.5. In this Source Code we present a class that provides a function to divide two numbers (lines 2 to 7) in the supplier side and a instantiation and method call in the client side (lines 10 and 11). Line 11 shows a nonconformance in the client side, the precondition of `div` method is broken.

In this work, we are considering only nonconformances from Case 1 – in the supplier side, but we intend to extend our approach to work with client-side checking [59].

Código Fonte 2.5: Code that presents a nonconformance in the client side – Case 2.

```
// supplier side
public class MathOperations{
  //@ requires y > 0.0;
  public double div(double x, double y){
    return x/y;
  }
}

// client side
MathOperations mo = new MathOperations();
mo.div(3.5,0.0); // Contract violation here.
```

2.3 Software Testing

Testing is an activity intended to discover system defects before it is put into use [68]. Regardless test cannot guarantee that the system is defect free [25], they can be used to increase the confidence in the system behavior; thus, tests have been used for years in software engineering.

According to Sommerville [68], traditionally the testing process has two distinct goals: to demonstrate to developer and customer that the software meets its requirements; and to discover undesirable or incorrect situations, or does not conform to its specification. The first goal leads to validation tests, checks the expected behavior from the system; the second, leads to verification tests, checks if the software meets its stated functional and non-functional requirements. In this work, the tests are used aiming the second goal: checking consistency between code and contracts.

The testing process starts as soon as a requirement becomes available and continues through all stages of the software development. In the software testing context, the concepts of *error*, *failure*, and *fault* are widely used. According to Binder [8] and IEEE [24], an *error* is a human action that results in a software fault; a *failure* is a manifested inability of a system to perform a function; and a *fault* is defined as the absence of code or the presence of incorrect code in a system software that causes the failure. In this work, a nonconformance is considered as a fault.

Testing may be carried out at three levels of granularity [68]:

1. Unit testing, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.

2. Component testing, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.

3. System testing, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

Depending on the system under test, an approach to conformance checking can be used in any level of granularity: unit testing, component testing, or system testing. The remainder of this section presents some concepts about software testing that are needed to understand this dissertation, like test cases, oracles in DBC context, an approach to tests generation, and a test technique – conformance testing.

2.3.1 Test Cases

A test case is composed by a set of inputs, execution conditions, and expected results chosen to test a behavior of the system under test [8]. Each test case has basically two information: inputs – conditions that must be satisfied before test execution (preconditions to methods or objects under test), and the data chosen to test the system; outputs – postconditions that must be satisfied before the test execution, and the output produced by the system.

In the context of this work, a test case is a set of instantiations and modifications (method calls with their parameters) in an object under test. The expected results are given by the contracts present in the code. Source Code 2.4 presents an example of test case, in line 1 there is an object instantiation, lines 2 to 4 present modifications (method calls).

2.3.2 Oracles in DBC Context

A test oracle determines if the result of a program *p1* using a test case *tc1* is correct [69]. There are several methods of oracle creation, including manually specifying expected outputs for each test, monitoring user-defined assertions during test execution, and verifying if the outputs match those produced by some reference implementation.

According to the Design by Contract methodology [48], contracts (class invariants and pre- and postconditions) express elements of the specification from the software being developed. If they are executable, they can be monitored at runtime and any contract violation signals a fault into the program. In DBC the oracles are highly dependent from the quality and coverage of the contracts, if the programs have few contracts, their oracles can assert few properties about these systems.

In this work we are considering the contracts present into the code as oracles. Those contracts are used to conformance checking between source code and specification. For example, to `updateCount` method (Source Code 2.3) we show in Source Code 2.6 the oracle generated for this method (using a JML compiler – *jmlc* in this case). We omitted some details to simplify understanding. The assertions were transformed in try-catch, and if-control structures – assertion checkers in runtime. First there is a check for invariant violations (line 4 – once that an invariant must be hold before and after every method call), then there is a precondition checking (line 7), if the precondition is respected, the method is executed (line 11) and after method execution there is a postcondition checking (line 13); if there are some contract violation, lines 16 to 23 try to catch, if no contract violation occurs, the invariant is checked again (line 29).

2.3.3 Tests Generation

Considering the approach to test generation there are basically two fundamental approaches: white box testing and black box testing [39]. White box testing (such as Control Flow Testing, Branch Testing, and Loop Testing [41]) is a kind of test where the implementation of the system under test is considered. On the other hand, black box testing (such as Equivalence Class Partitioning, Decision Tables, State Transition Diagrams, and Use Case Testing [6]) is a kind of test performed to verify whether, for a given input, the system produces the correct output; correct based on the specification of the system – the system oracle. There is yet, a hybrid approach that mixes features from black and white box approaches; the Gray-box testing. In this work we are considering a gray box testing approach.

Furthermore, the test generation can be performed in two ways: manually – when the

Código Fonte 2.6: Oracle generated to updateCount method

```
public void updateCount(boolean arg0){ {...}
  try{
  // checks invariant before method execution
    if (JMLChecker.isActive(JMLChecker.INVARIANT)){
      try{
          // checks precondition
         if (JMLChecker.isActive(JMLChecker.PRECONDITION)){
            boolean rac$ok = true;
            boolean rac$inv = true;
         try{
           internal$updateCount(boolean);
           // checks normal postcondition
           if (JMLChecker.isActive(JMLChecker.POSTCONDITION) && rac$dented
              ()){...}
           } } } }
  }
  catch(JMLInvariantError rac$e) {...}
  catch(JMLEntryPreconditionError rac$e) {...}
  catch(JMLInternalPreconditionError rac$e) {...}
  catch(JMLInternalNormalPostconditionError rac$e){...}
  catch(JMLAssertionError rac$e){...}
  catch(java.lang.Throwable rac$e){ rac$inv = false;
    try{
      //checks exceptional postcondition
      if(JMLChecker.isActive(JMLChecker.POSTCONDITION) && rac$dented())
         {...}
    } catch(JMLAssertionError rac$er){...}
  }
  finally{ if(rac$ok && rac$inv){
    // checks invariant
    if(JMLChecker.isActive(JMLChecker.INVARIANT) && rac$dented()) {...} }
        }
  {...}
}
```

tests are written by a tester; or automatically – when generated by a tool. In this work we are considering tests generated automatically, in a Randomly-Generated Tests (RGT) approach by means of Randoop [55]. Randoop is a feedback-directed random test generation tool for Java programs, the tool randomly generates tests for a set of methods given a time limit. The tool generates tests in JUnit[5] format. Figure 2.1 presents the process that Randoop uses to tests generation. Source Code 2.7 presents a test case generated by Randoop for the motivating example (Section 1.1.1).

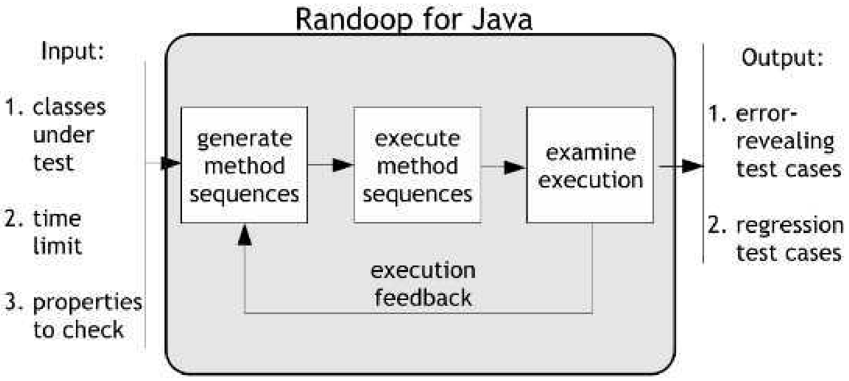


Figura 2.1: The Randoop test generation process. The process starts when are given the following inputs: a list of classes under test, a time limit, and optionally a set of properties to check. Then sequences of method calls are generated, executed and examined; the feedback from the execution feeds back the process until the time limit be reached.

2.3.4 Conformance Testing

Conformance testing is an approach used to verify whether the implementation of a system is in conformance with its specification (its contract), in other words, if the code satisfies its specification. This kind of test uses as oracle the specifications in the source code and the conformance is determined by the adequacy of the test results with the contracts. There are basically two ways to automatically check conformance: dynamically and statically. Dynamic checking of conformance between code and specification is done by runtime assertion checking, that is, simply running the code and testing for violations of assertions from the

[5] JUnit is a programmer-oriented testing framework for Java, available online `http://junit.org/`.

Código Fonte 2.7: Test case generated by Randoop

```
public void test1() throws Throwable {
  if (debug) { System.out.print("RandoopTest0.test1"); }
  MapMemory var0 = new MapMemory();
  var0.updateMap(false);
  var0.updateMap(true);
  var0.updateMap(true);
  var0.updateMap(false);
  var0.updateMap(false);
  var0.updateMap(true);
  var0.updateMap(true);
  var0.updateMap(true);
  var0.updateMap(false);
}
```

specification. Tools like JMLUnit [18], JMLUnitNG [74], JET [16], and JMLOK 2.0 combine runtime assertion checking with unit testing for dynamic checking of JML programs. Another way is verifying that the code satisfies its specification statically. This can give more assurance in the correctness of the code as it establishes the correctness for all possible execution paths, whereas dynamically only exercises the execution paths according to the test suite being used. Nevertheless, correctness of a program with respect to a given specification is not decidable in general. Tools like ESC/Java [29], ESC/Java2 [23], and LOOP [7] performs static checking for JML programs. Any verification tool (static or dynamic) must trade off the level of automation it offers and the complexity of the properties and code that it can handle [12].

In a formal point of view, conformance testing relates a specification and an implementation under test (IUT) by the relation **conforms–to** $\subseteq$ *IMPS x SPECS*, where *IMPS* represents the implementations and *SPECS* represents specifications. Therefore, *IUT* **conforms–to** s if and only if *IUT* is a correct implementation of specification s [70].

The **conforms–to** relation is harder to be checked by tests than by static analysis; so in this work we are considering an informal definition of conformance testing: to us, conformance is when the code satisfies its specification; the code does what the specification declares. The ideas of **satisfies** relation arose from Hoare's triple [38]. A code satisfies its specification if the code results are expected by its specification. In our work, we use a dynamic approach to conformance checking, tests are used to compare code execution results with oracles generated by the specification.

Capítulo 3

Uma abordagem para Detecção e Categorização de Não-conformidades em Programas Baseados em Contratos

Em programas baseados em contratos, a prévia detecção de não-conformidades é desejável, para que os desenvolvedores estejam mais confiantes na corretude e robustez do sistema de software sendo escrito [48]. As não-conformidades detectadas em programas baseados em contratos devem ser corrigidas para garantir a qualidade do software. Neste capítulo apresentamos uma abordagem dinâmica para verificação de conformidade em programas baseados em contratos e uma abordagem baseada em heurísticas para categorizar as não-conformidades detectadas nestes tipos de programas. Primeiro, apresentamos a abordagem de verificação de conformidade (Seção 3.1), então apresentamos a abordagem de categorização de não-conformidades (Seção 3.2). Por fim, na Seção 3.3 descrevemos uma implementação dessas abordagens no contexto de programas Java/JML – por meio da ferramenta JMLOK 2.0.

3.1 Randomly-Generated Tests Approach for Conformance Checking

In this section we present the Randomly-Generated Tests (RGT) approach for checking conformance in contract-based programs, implemented by our tool, JMLOK 2.0. The approach benefits from automatic generation of test data and cases; it is promising to conformance checking as it provides a considerable number of ready-to-run test cases that can be used to detect contract assertion violations in compiled contract-based programs. Our approach is an improvement of the approach presented by Oliveira [53]. While JMLOK was able to generate tests and displays the test results between *meaningless* and *relevant*; JMLOK 2.0 is able to detect and categorize nonconformances and to display for the user only the distinct nonconformances detected and categorized. Our detection does not present for the user false positives; whereas JMLOK presents. Furthermore, the user interface was improved in the new version of the tool, providing to the user more information about the nonconformance that was detected.

The approach determines a straightforward process, which starts when a contract-based program is given as input; then the following steps are performed: (1) generation of unit tests (using a RGT engine, like Randoop [55]) composed of sequences of constructor and method calls over a class under test, as originally yielded by a conventional compiler (e.g. Java JDK); (2) generation of test oracles from contracts by a specific contract-aware compiler (e.g. *jmlc* [17]); (3) execution of the generated test suite, against the oracles, asserting the equivalence between test output and the assertions generated by the oracles; (4) after execution, a filter distinguishes passed tests over failed, and into the subset of failed tests, the filter groups from all failures which are distinct nonconformances (faults); (5) finally the list of nonconformances resultants is returned. Figure 3.1 presents this approach that is implemented in JMLOK 2.0 tool.

3.1.1 Step 1 - Tests Generation

This step starts with the compilation of the contract-based program, using a compiler from the source language (like Java compiler). After compilation, if no errors occurred, test cases

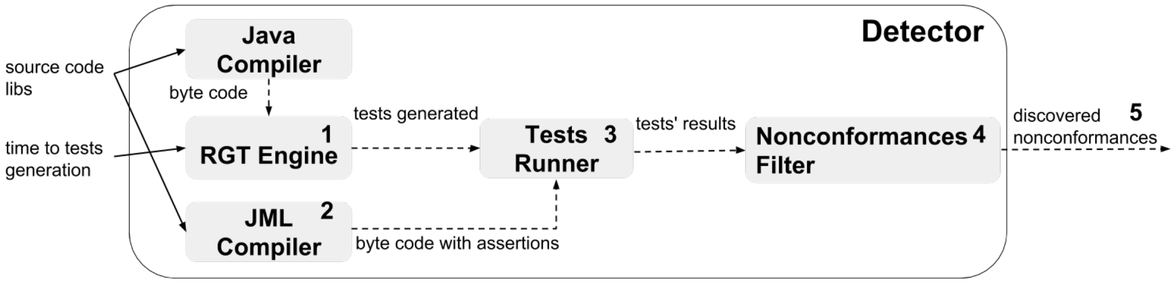


Figura 3.1: The internal structure of the proposed RGT-based approach for nonconformances detection. As inputs are given the contract-based program, a directory of external libs (optional) and the time to tests generation (also optional, default value = 10 seconds). 1- Tests are generated into the RGT engine. 2- Oracles are produced from contracts. 3- Tests generated in step 1 are run against oracles produced in step 2. 4- A filter distinguishes from all failures which distinct nonconformances were detected. 5- The list of detected nonconformances is returned.

and test data are generated using a RGT engine.

Depending on the stage of development from the system, we can have three kinds of tests: unit tests, component tests, or system tests [68]. In our approach we can have any of these types of test. The test generation is based only on the implementation, contract are used as oracles (Section 3.1.2); otherwise the contract violations would expected by the test in the oracle. The tests are composed by several calls to all public methods and constructors from classes under test. Source Code 3.1 exemplifies a test generated in our approach for test the `updateMap` from class `MapMemory` (Source Code 1.1).

Código Fonte 3.1: Test generated by our approach from motivating example

```
public void test5() throws Throwable {
  if (debug) { System.out.println(); System.out.print("RandoopTest0.test5
     "); }
  MapMemory var0 = new MapMemory();
  var0.updateMap(false);
  var0.updateMap(true);
  var0.updateMap(true);
  var0.updateMap(false);
  var0.updateMap(false);
  var0.updateMap(false);
10    var0.updateMap(true);
11    var0.updateMap(false);
12    var0.updateMap(true);
13    var0.updateMap(false);
14    // The following exception was thrown during execution.
15    // This behavior will recorded for regression testing.
16    try {
17      var0.updateMap(false);
18      fail("Expected exception of type java.lang.
          ArrayIndexOutOfBoundsException");
19    } catch (java.lang.ArrayIndexOutOfBoundsException e) {
20      // Expected exception.
21    }
22  }
```

In some cases, only several calls to the methods under test can reveals a nonconformance (for example, in our motivating example – Source Code 1.1 –, only after three calls to `updateMap` with *true* as parameter reveals a nonconformance – lines 5, 6 and 10 from Source Code 3.1). The automatic tests generation allows the creation of scenarios that performs consecutive calls to a method under test, increasing the variety of the tests and hence the possibility of nonconformances detection.

3.1.2 Step 2 - Oracle Generation

The second step is the oracle generation. This step consists in use a specific contract-aware compiler to produce oracles. To us, the test oracles are assertions generated by the contract-aware compiler from contracts present into the system.

Some contract-based languages like Eiffel [47], Spec# [3], and JML [43] have compilers that generate oracles from contracts in runtime. An example of oracle generated by *jmlc* compiler to `updateCount` method is presented in Source Code 2.6. We use the RGT engine to generate tests in the bytecode without assertions because otherwise the contract violations would expected by the test in the oracle. We use the bytecode with assertions only in tests execution.

3.1.3 Step 3 - Tests Execution

In this step, the tests generated in the first step are executed considering the oracles generated in the second step. After the comparison between test results and oracles these tests are classified in *failure* and *success*; *failure* when there is a contract violation, and *success* otherwise.

When a test is classified as *failure*, a new classification is performed. If the contract violation is from *precondition* type, we check if the precondition problem occurs in the entry of a method under test – directly into the test case; in affirmative case, the test is classified as *meaningless*[1] [19], otherwise the *failure* is relevant. This distinction occurs only with violations of *precondition*, the others: *postcondition*, *invariant*, *constraint*, and *evaluation* do not have this needed, all contract violations of these types are relevant.

3.1.4 Step 4 - Grouping distinct Nonconformances

In this step, all failures are grouped in distinct nonconformances. The grouping algorithm consists on gathering all failures presented in the different test cases according to the detected nonconformance. Thus, only the distinct nonconformances are returned. Therefore, JMLOK 2.0 presents detected nonconformances in a meaningful way, whereas previous version of JMLOK tool [53] presents the whole set of failures (possibly repeated) revealed by the tests.

In our approach two failures are equal when they have the same type and location – they were detected in the same part of the contract-based program, considering package, class, and method; otherwise, they are distinct. We compare all failures revealed by the tests and list only those that represent distinct nonconformances. Figure 3.2 shows a high-level of this step.

3.1.5 Step 5 - Results from Detection Phase

This is the last step from conformance checking phase. In this step the list of distinct nonconformances is returned within test cases that reveals the nonconformances. The result of this

[1] meaningless are tests that violate preconditions in methods entry (preconditions violated directly by the test case) because the test generator approach does not consider specifications in the process of test generation.

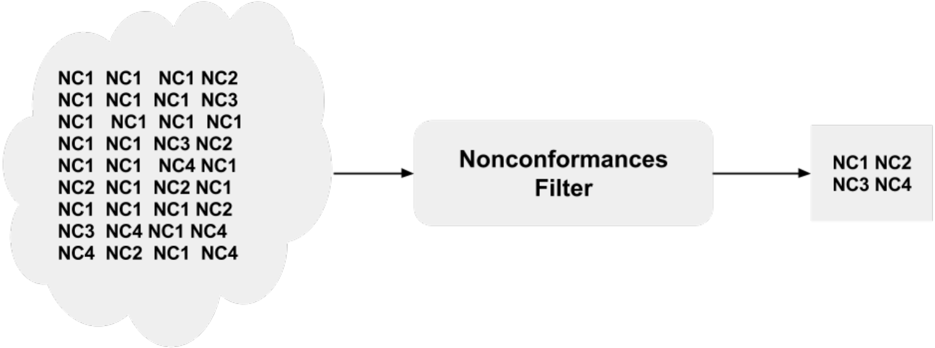


Figura 3.2: Grouping nonconformances – this process receives as input a list of several failures (some possibly equal), then the failures are compared and the filter returns a set of distinct nonconformances (distinct faults).

step is used as input for the automatic categorization approach, that automatically suggests causes for nonconformances. So, the developers can analyse the test case and the likely cause for each nonconformance, and uses this information as a starting point to nonconformances correction.

3.2 Heuristics-based Approach for Categorize Nonconformances

In this section we present the approach for categorize nonconformances in contract-based programs. First we present the categorization model for nonconformances (Section 3.2.1), then the overview of this model is shown in Section 3.2.2. Finally, Section 3.2.3 describes in details the heuristics proposed to categorize nonconformances.

3.2.1 Categorization Model

In order to categorize nonconformances, we propose a three-level model to categorization[2]: each nonconformance has a category, a type, and a likely cause (see Table 3.1). An error that apparently occurs in the contract is regarded as specification error; in contrast, apparent error in the body of the problematic method(s) is a code error; it is undefined when it is not possible

[2] This model is an extension of the categorization model proposed in our previous work [50].

Tabela 3.1: Categorization model for nonconformances in contract-based programs.

Category	Type	Likely Cause
Specification error	precondition	Strong precondition
		Weak postcondition
	postcondition	Weak precondition
		Strong postcondition
	invariant	Weak precondition
		Strong invariant
	constraint	Weak precondition
		Strong constraint
	evaluation	Weak precondition
		Strong postcondition
Code error	–	Code error
Undefined	–	Undefined

– considering a non-expert in the application domain – to determine whether the problem is in the contract or in the source code (this category is used only for manual categorization purposes). The type is given automatically by the assertion checker, and corresponds to the part of JML that was violated – considering only visible behavior from the systems.

Each error may present several likely causes, which cannot be deterministically diagnosed – debugging can be aided, however, by specific heuristics. For example, regarding an invariant error, we suggest a likely cause following the heuristics aforementioned: first check whether there is the default precondition, or nothing, or whether there is at least one field modified on method body; in either case, the likely cause is determined as *Weak precondition*; otherwise *Strong invariant* is the suggestion. Regarding the example in Section 1.1.1, in which there is a nonconformance of invariant type, once the method `GenCounter.updateCount` does not have an explicit precondition (the method receives the default *true*) and the likely cause suggested is *Weak precondition*. The complete set of heuristics for each type is available in Section 3.2.3.

As an example for the category of Specification error, the class `ArrayUtils` (project

JAccounting [60]) has a method getMaxIntArrayIndex, with a postcondition violation that occurs after a creation of an array of integers and one call to the method. The nonconformance occurs in the body of getMaxIntArrayIndex with an invalid access to an empty array, causing an exception (ArrayIndexOutOfBoundsException). This problem possibly occurs by reason of the precondition that does not check the array size. Therefore, we have a nonconformance with type Postcondition, category Specification error and likely cause weak precondition; these information may be used by the developer in the process of nonconformances correction. Source Code 3.2 presents the test case that reveals this nonconformance.

Código Fonte 3.2: Test case that reveals a nonconformance that we categorize as specification error

```
public void testGetMaxIntArrayIndex(){
  int[] var0 = new int[] { };
  int var1 = ArrayUtils.getMaxIntArrayIndex(var0);
}
```

As an example of Code error, the class Personal_Impl (project HealthCard [62]) there is an invariant violation that occurs after the creation of an object from class. The nonconformance occurs due to the default constructor does not initialize the field birthplace, violating the default JML invariant, that requires all fields as non-null. This problem possibly occurs by a code error – the lack of initialization of all fields of this class. Source Code 3.3 presents the test case that reveals this nonconformance.

Código Fonte 3.3: Test case that reveals a nonconformance that we categorize as code error

```
public void testPersonal_Impl(){
  Personal_Impl var0 = new Personal_Impl();
}
```

Finally, as an example of Undefined, the class Common (project Bomber [60]) has a method div, with a precondition violation that occurs in the call to distancePointToLine method. distancePointToLine calls the methods sqr, sqrt and mul to construct parameters to call div. The nonconformance apparently occurs for the reason that sqr, sqrt, and mul – called to create the parameters for the call to div – perform a shift of several bits (in some cases 10); as in this test case the generated values are a little bit small (usually amount of 10), the shift results in zero, violating the precondition of `div`. Once the `Bomber` domain is mobile games, maybe the generated values are large enough; therefore, we categorize this nonconformance as Undefined, because we cannot say if the problem stays in the code or in the specification. Source Code 3.4 presents the test case that reveals this nonconformance.

Código Fonte 3.4: Test case that reveals a nonconformance that we categorize as undefined error

```
public void testCommon(){
  int var4 = Common.distancePointToLine(32, (-4), 10, 1);
}
```

3.2.2 Categorization Overview

The approach for categorization determines a straightforward process, which starts when a list of nonconformances and the source code are given as input; then, the following steps are performed: (1) the process starts when the heuristics selector receives a set of nonconformances, then a subset of heuristics to each nonconformance is selected based on its type; (2) next, each subset is used together the source code to choose a likely cause for the nonconformance; (3) finally the list of categorized nonconformances resultant is returned. Figure 3.3 presents this approach.

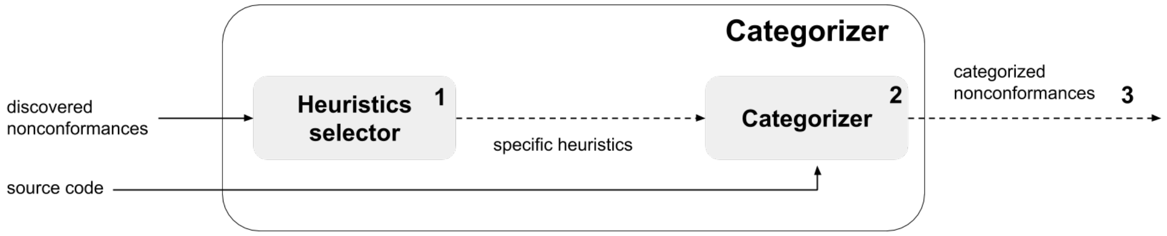


Figura 3.3: Internal structure of the approach for nonconformances categorization. As input are given the list of nonconformances and the source code. 1- a subset of heuristics to each nonconformance is selected based on its type. 2- Each subset is used together the source code to choose a likely cause for the nonconformance. 3- The list of categorized nonconformances is returned.

The categorization may help the programmer given an idea about the probable cause of the problem and reducing the search scope, and it is a first step ahead of the completely automated categorization of nonconformances.

3.2.3 Heuristics

In this section, we show our set of heuristics to categorize each one type of nonconformances that we are considering (precondition, postcondition, invariant, constraint, and evaluation).

For a nonconformance from precondition type, we proposed to check whether there is at least one parameter on precondition. In positive case, the likely cause is suggested as *Strong precondition*; otherwise *Weak postcondition* is the suggestion. If there is a parameter or field in the precondition, the method can become a little restrictive – indicating a possible strong precondition; otherwise, the postcondition of one method used in the call to the method with precondition problem can be weak and allows return values that violates the precondition of the called method. Figure 3.4 shows the heuristic for nonconformances of precondition.

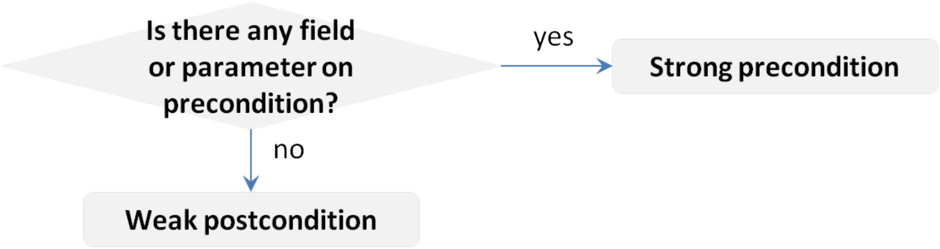


Figura 3.4: Heuristic for nonconformances from precondition type.

For a nonconformance from postcondition or evaluation types, whether there is the default precondition, or nothing, or whether there is at least one field modified on method body; in either case, the likely cause suggested is *Weak precondition*; otherwise *Strong postcondition* is the suggestion. If a method has the default precondition, it means that all clients are allowed and this fact can cause problems in the method exit – the method's body can be unable to produce the desired result; so, weak precondition is proposed; furthermore, if there is a field modified on method body, and the precondition does not check anything about the field, weak precondition is also suggested. Finally, if neither of these cases occurs, the suggestion is strong postcondition – a postcondition so strong that possibly cannot be satisfied by the method's body. Figure 3.5 shows the heuristics for nonconformances of postcondition and evaluation.

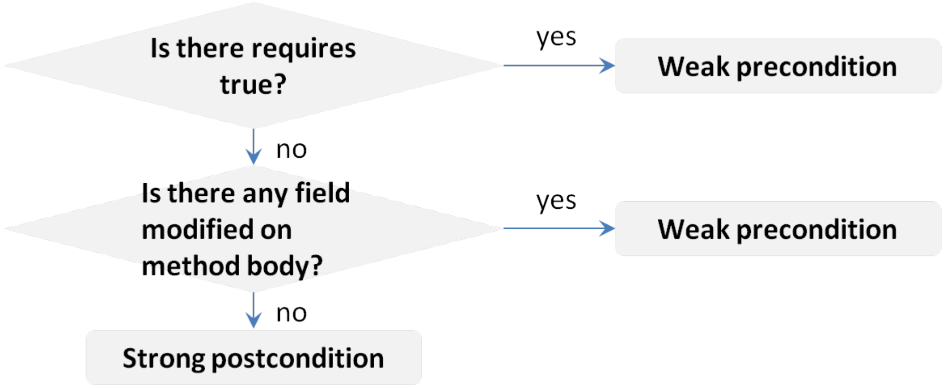


Figura 3.5: Heuristics for nonconformances from postcondition and evaluation types.

For a nonconformance from invariant type, whether there are some field from the class that is not initialized into the constructor, the likely cause proposed is *Code error*; otherwise, whether there is the default precondition, or nothing, or whether there is at least one field modified on method body; in either case, the likely cause suggested is *Weak precondition*; otherwise *Strong invariant* is the suggestion. If a class has some field not initialized, the default JML invariant is violated (invariant that determines all fields must be non-null); so, in this case we suggest as likely cause Code error (null-related) due to the fact that the code does not initialize all field from the class. Case all field are non-null and the method has the default precondition or nothing, it means that all clients are allowed and this fact can cause problems in the method exit (violating an invariant, for example) – the method's body can be unable to produce the desired result; so, weak precondition is proposed; furthermore, if there is a field modified on method body, and the precondition does not check anything about the field, weak precondition is also suggested. Finally, if neither of these cases occurs, the suggestion is strong invariant – an invariant so strong that possibly cannot be satisfied by the method's body. Figure 3.6 shows the heuristics for nonconformances of invariant.

Finally, for a nonconformance from constraint type, whether there are some field from the class that is not initialized into the constructor, the likely cause suggested is *Code error*; otherwise, whether there is the default precondition, or nothing, or whether there is at least one field modified on method body; in either case, the likely cause suggested is *Weak precondition*; otherwise *Strong constraint* is the suggestion. If a class has some field not initialized can be that a field manipulated into a constraint had null value, so, the likely cause proposed is Code error (null-related); due to the code does not initialize all fields from the class.

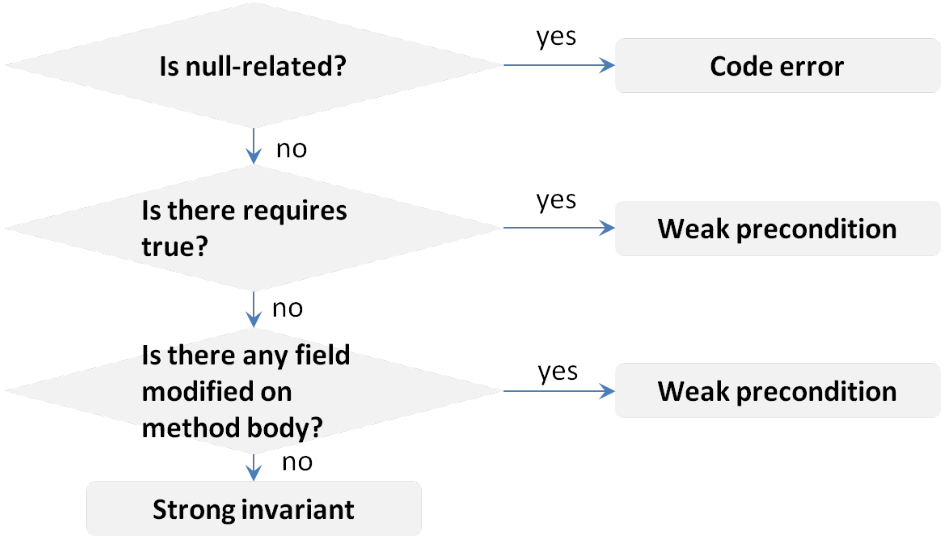


Figura 3.6: Heuristics for nonconformances from invariant type.

Case all field are non-null and the method has the default precondition or nothing, it means that all clients are allowed and this fact can cause problems in the method exit (violating a constraint, for example) – the method's body can be unable to produce the desired result; so, the suggested likely cause is weak precondition; furthermore, whether there is at least one field modified on method body, and the precondition does not check anything about the field, weak precondition is also suggested. Finally, if neither of these cases occurs, the suggestion is strong constraint – a constraint so strong that possibly cannot be satisfied by the method's body. Figure 3.7 shows the heuristics for nonconformances of constraint.

These set of heuristics were implemented into a module for nonconformances categorization in JMLOK 2.0 tool (Section 3.3.4).

3.3 JMLOK 2.0

JMLOK 2.0 is our implementation for the RGT-based and heuristics-based approaches, and is an improvement of Oliveira [53] work. Figure 3.8 shows its architecture. The tool is composed by four modules: UI, Controller, Detector (internal structure of this module is shown in Figure 3.1) and Categorizer (internal structure of this module is shown in Figure 3.3). Execution steps are indicated as follows: (1) the contract-based program, some external library needed to the program, and the time limit for test generation (the time to stop the test

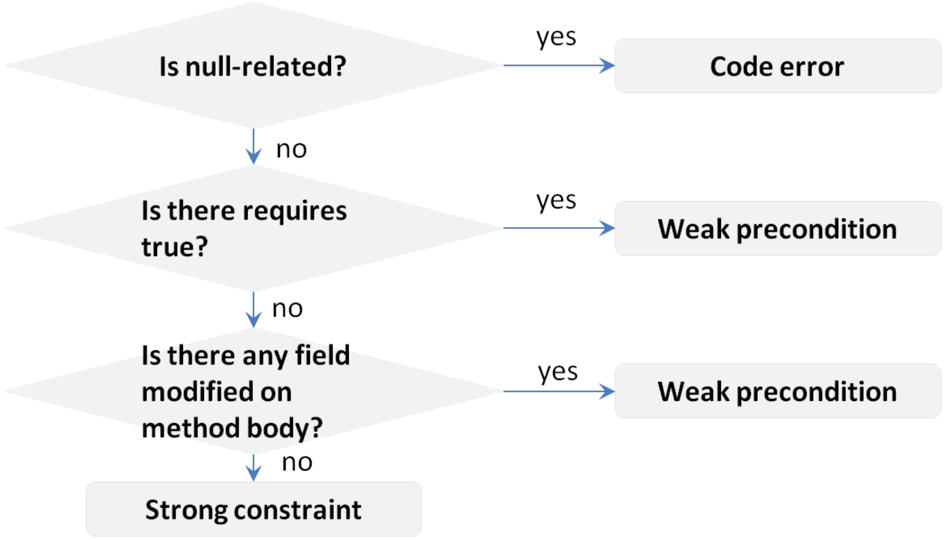


Figura 3.7: Heuristics for nonconformances from constraint type.

generation) are given as input; (2) the Controller module forwards the data to Detector and Categorizer modules; (3) the Detector module runs and returns a list of nonconformances to Categorizer module; (4) the Categorizer module runs and returns to Controller module a list of categorized nonconformances, which are presented on the UI.

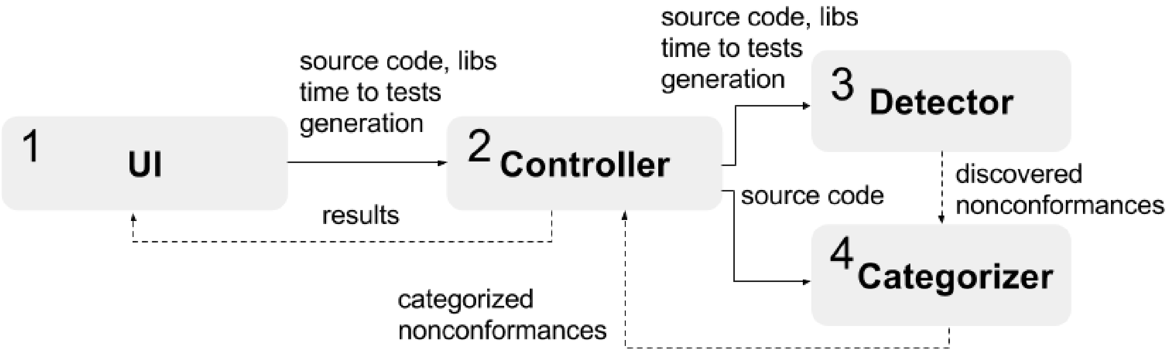

1 UI
source code, libs time to tests generation
results
2 Controller
source code, libs time to tests generation
3 Detector
source code
discovered nonconformances
4 Categorizer
categorized nonconformances

Figura 3.8: The architecture of JMLOK 2.0 tool. There are 4 modules in this tool: UI, Controller, Detector, and Categorizer.

In Sections 3.3.1 to 3.3.4, we present the details about the implementation of the JMLOK 2.0. Section 3.3.5 shows some details about the tool architecture. Finally, Section 3.3.6 describes some limitations of this approach.

3.3.1 Step 1 - User interaction

The first step performed in the tool is the user interaction: the contract-based program, some external library needed to the contract-based program, and the time limit for test generation (the time to stop the test generation) are given as input by the user and the button *Run* is pressed. Figure 3.9 shows the initial user interaction screen from JMLOK 2.0. In this screen the user can give the contract-based program, the folder to external libraries (needed in the contract-based program), and the time to tests generation. Only the contract-based program is mandatory; by default the tool uses its libraries and the time limit of 10 seconds to tests generation. The time limit depends on the contract-based programs being checked. We recommend starts with a low value (e.g. 2 seconds) for the time limit and increases the value within resources for conformance checking of the project. Soares et al. [67] present some experiences with the use of time limit for Randoop.

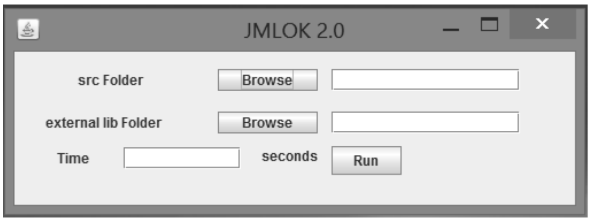


Figura 3.9: The initial screen of the tool. In this screen the user can give the inputs needed to use the tool. If no input is given, a message dialog warns the user about the needed of at least the contract-based program to be given as input.

3.3.2 Step 2 - The Controller module

The Controller module receives the data from UI and sends to Detector and Categorizer. After this, the module wait until receive data from Categorizer or until receive some information about problems in tool execution. When the Controller receives the results from Categorizer, the module sends to UI these results, to UI presents the results to the user.

3.3.3 Step 3 - The Detector module

This module was improved from the previous approach [53]. In the previous version, there were not modularization neither generalization. We improved some parts, first updating the versions of Randoop [55] and JUnit[3], Randoop version 1.3.4 and JUnit version 4.0. Another improvement that we performed, was to allow the user chooses the JML compiler that he wants to use (the current *jmlc* [17] or OpenJML [22]). Furthermore, we added a filter that allows to return only the distinct detected nonconformances and we developed an approach for automatically categorize nonconformances (Section 3.3.4).

Tests Generation

The tests generation starts when the Controller sends the data for Detector. Then, the contract-based program is compiled using the Java compiler and the bytecode is past to Randoop [55] tool, to tests generation. We chose Randoop as RGT engine because this tool generates many sequences of calls to the object under test in a given time limit. The time limit is used to bound the test generation process. So, we suggest starts with a low value for the time limit and increases, like performed in Soares et al. [67]. For tests generation, Randoop uses as input public methods or constructors (operations under test) for testing. It generates sequences of invocations, and uses the sequences to create tests and test data for further invocations. As a consequence, the number of generated test cases depends both on the number of available operations and the behavior of the experimental unit, which provides feedback to Randoop.

We investigate the Korat [11] tool. Korat is a tool based on the JML-JUnit approach, which allows exhaustive testing of a method for all objects of a bounded size. The tool automatically constructs all non isomorphic test cases and execute the method on each test case. However, test cases constructed by Korat only consist of one object construction and one method call on this object. Furthermore, Korat requires an imperative predicate that specifies the desired structural constraints and a finitization that bounds the desired test input size. So, we prefer continue to use Randoop as RGT engine. An example of test case generated by Randoop is presented in Source Code 3.1.

Oracle Generation

Once *jmlc* [17] compiler is a deprecated project and does not support some new features of Java language – like generics and other features from Java 5+; the JML community is trying

[3] `http://junit.org/`

to update tools to support the new compiler, OpenJML [22]. For that reason, aiming improve our tool and use OpenJML, we created some test scenarios to investigate the change feasibility. As a result of our tests with the OpenJML compiler, we found that its infrastructure is still incipient to use in tools to conformance checking. This compiler could lead us to report false positives; once OpenJML does not thrown exceptions, only warnings are thrown. We contacted the compiler's developer but, unfortunately, the solution proposed to this problem was costly[4]; so, we prefer continue to use the old compiler (*jmlc*).

So, in this step, we continue to use *jmlc* as the JML compiler. The procedure to conformance checking consists in check the assertions generated by *jmlc*; if an assertion is violated, an specific exception is thrown. This compiler has some limitations (the lack of support to some new Java features, like Java generics and features from Java 5+), but the new JML compiler (OpenJML [22]) is still in development and does not have the same features of *jmlc*. We are using *jmlc* version 5.6_rc4. We do not use the Randoop oracles because they are created without consider the JML contracts.

Although JMLOK 2.0 does not use OpenJML yet, we implemented this tool to work with this compiler as soon as a more stable version becomes available.

Tests Execution

In this step, the tool runs the test suite against the oracles. JUnit is the tool used to run the tests, once Randoop generates tests in JUnit format. After the comparison between tests results and oracles, those tests are classified in *failure* and *success*; *failure* when there is a contract violation, *success* otherwise.

When a test is classified as *failure*, a new classification is performed. If the contract violation is from *precondition* type, we check if the precondition problem occurs in the entry of a method under test – directly into the test case; in affirmative case, the test is classified as *meaningless*[5] [19], otherwise the *failure* is relevant. This distinction occurs only with violations of *precondition*, the others: *postcondition*, *invariant*, *constraint*, and *evaluation*

[4] The solution proposed was set a runtime property (to say for the compiler throws exceptions) in each place that a call to a method on a class under test. Once our approach generate several test methods and each test method has some method calls to the object under test, this would be very costly for our detection phase.

[5] meaningless are tests that violate preconditions in methods entry (preconditions violated directly by the test case) because the test generator approach does not consider specifications in the process of test generation.

do not have this needed, all contract violations of these types are relevant.

Grouping Distinct Nonconformances

In this step, all failures are grouped in distinct nonconformances. The grouping algorithm consist on gathering all failures presented in the different test cases according to the detected nonconformance. Thus, only the distinct nonconformances are returned. Therefore, JMLOK 2.0 presents detected nonconformances in a meaningful way, whereas previous version of JMLOK tool [53] presents the whole set of failures (possibly repeated) revealed by the tests.

In our approach two failures are equal when they have the same type and location – they were detected in the same part of the contract-based program, considering package, class, and method; otherwise, they are distinct. We compare all failures revealed by the tests and list only those that represent distinct nonconformances.

Results from Detection Phase

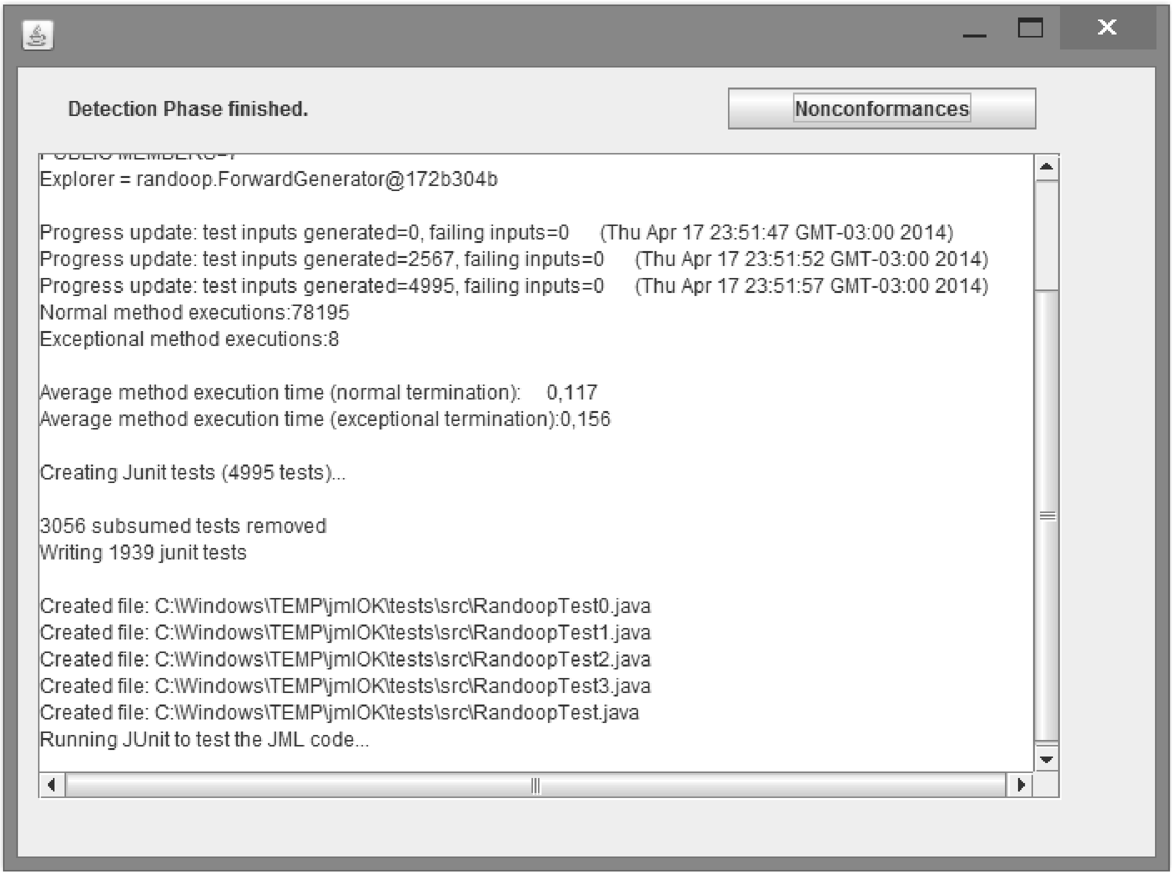
As result of Detection step, a list of distinct nonconformances is returned. Each element of the list contains information about type of nonconformance, and its location. This list will be used in the process of categorization performed by the Categorizer.

Figure 3.10 presents the intermediate screen that shows to user that Detection phase has finished.

3.3.4 Step 4 - The Categorizer module

In this step, the categorization of the detected nonconformances is performed. After performs the categorization, this module returns to Controller module a list of categorized nonconformances. Then the nonconformances are sent by Controller to the UI, and the UI presents the results of the tool to the user.

This module executes the steps presented in Figure 3.3. First, the list of discovered nonconformances is given as parameter to the heuristics selector (Section 3.3.4). Then, a set of heuristics, the set of discovered nonconformances, and the source code is past to the categorizer (Section 3.3.4). Finally, the list of categorized nonconformances is returned, and



```
PUBLIC MEMBERS=7
Explorer = randoop.ForwardGenerator@172b304b

Progress update: test inputs generated=0, failing inputs=0     (Thu Apr 17 23:51:47 GMT-03:00 2014)
Progress update: test inputs generated=2567, failing inputs=0     (Thu Apr 17 23:51:52 GMT-03:00 2014)
Progress update: test inputs generated=4995, failing inputs=0     (Thu Apr 17 23:51:57 GMT-03:00 2014)
Normal method executions:78195
Exceptional method executions:8

Average method execution time (normal termination):     0,117
Average method execution time (exceptional termination):0,156

Creating Junit tests (4995 tests)...

3056 subsumed tests removed
Writing 1939 junit tests

Created file: C:\Windows\TEMP\jmlOK\tests\src\RandoopTest0.java
Created file: C:\Windows\TEMP\jmlOK\tests\src\RandoopTest1.java
Created file: C:\Windows\TEMP\jmlOK\tests\src\RandoopTest2.java
Created file: C:\Windows\TEMP\jmlOK\tests\src\RandoopTest3.java
Created file: C:\Windows\TEMP\jmlOK\tests\src\RandoopTest.java
Running JUnit to test the JML code...
```

Figura 3.10: The screen resultant of Detection phase. In order to get the results of the categorization, the user have to press the button Nonconformances.

the programmer can analyze the source code and contract to correct the nonconformances detected and categorized by the tool.

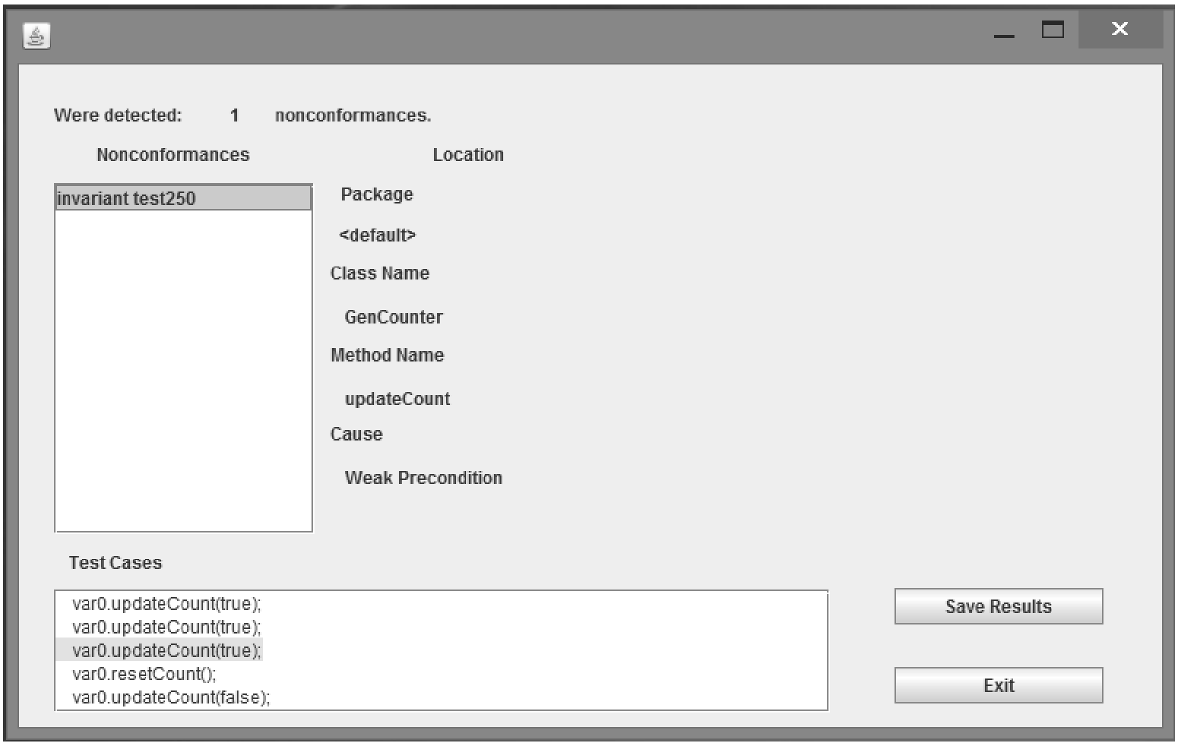
Heuristics selector

The first step of the categorization module is the selection of a subset of heuristics for a nonconformance. Each type of nonconformance has a set of heuristics, that were presented in Section 3.2.3. So, in this step a switch directs each type for its set of heuristics. As a result of this first step, a list of set of heuristics is past to the categorizer step. For example, to the nonconformance presented in our motivating example (see Source Code 1.2), the heuristics used to categorize the nonconformance is presented in Figure 3.6 – heuristics for invariant.

Categorizer

The second step is the categorization of the nonconformance, based on its set of heuristics and the contract-based program corresponding. Each heuristic of the set of heuristics is checked with the program; and a resultant likely cause is returned. Finally, a list of categorized nonconformances is returned.

Figure 3.11 shows the result of the categorization process in JMLOK 2.0 tool, for the contract-based program presented in Source Code 1.1.1.



Were detected: 1 nonconformances.

Nonconformances

invariant test250

Location

Package

<default>

Class Name

GenCounter

Method Name

updateCount

Cause

Weak Precondition

Test Cases

```
var0.updateCount(true);
var0.updateCount(true);
var0.updateCount(true);
var0.resetCount();
var0.updateCount(false);
```

Figura 3.11: Categorization screen presenting the results of JMLOK 2.0 tool. For each nonconformance are presented: the type, the complete location (including information of package, class, and method), the suggested likely cause, and the test case with highlight in the line that reveals the nonconformance.

3.3.5 JMLOK 2.0 Architecture

In our improvement of JMLOK tool, we use an adaptation of the Model-View-Controller (MVC) pattern [13]. In this adaptation, the controller mediates communication between the model and the view. The view is responsible by the interaction with the user; and the model is responsible by the conformance checking and the categorization of the detected nonconformances. The MVC architecture of JMLOK 2.0 tool is the follows: the View is represented by UI module; the Model is represented by Detector and Categorizer (Detector is responsible by the conformance checking; and Categorizer is responsible by the categorization of detected nonconformances); and the Controller is represented by Controller module – this module mediates the communication between model and view. Figure 3.8 shows the architecture of JMLOK 2.0 tool.

As an improvement in relation to the previous version, in JMLOK 2.0, all features were modularized allowing that internal changes do not affect other modules. Furthermore, now the tool has a module that treats with categorization of nonconformances; and the detection performed includes a filter to differentiate between several test results which represent distinct nonconformances.

Although our approach has some limitations (see Section 3.3.6), we have a tool that can be used to detect and categorize contract violations; furthermore, the tool results may help the programmer in the process of nonconformances correction.

3.3.6 Limitations

Our conformance checking approach is based on tests, so we cannot argue about its completeness in finding all nonconformances that could be found in a given experimental unit. Furthermore, we can have a set of test cases that are unable to reveal nonconformances in a given project. Therefore, any generalization about the types of nonconformances, categories and likely causes should be disregarded.

Once our tool uses the *jmlc* as JML compiler, our tool cannot be used to check conformance in programs with new features of Java language (like generics and other features from Java 1.5+). Moreover, in our approach we are considering only the publicly visible behavior of methods, so we cannot check the internal behavior of the methods.

Our approach is limited to dynamic checking of the code and contract so, sometimes, we cannot suggest the same likely cause that the manual categorization. Nevertheless, our automatic results may help the developer as a start point to the nonconformance correction. Additionally, our approach does not consider the domain of each contract-based program being checked, which sometimes is important for the suggestion of likely cause from the nonconformance. For example, in Source Code 3.4 our automatic categorization approach suggests as likely cause for the precondition problem *Strong precondition*, whereas a manual categorization can suggests a *Code error* in the body of `distancePointToLine` or in the body of the methods called by `distancePointToLine` before the call to `div` method. Or yet, domain knowledge can indicate that the problem occurs in the test data generated by the test case, because in the real use of this class, values given to `distancePointToLine` will be much greater than zero and the precondition of `div` method will be respected.

Although our categorization module cannot determine the real cause of the nonconformance, the results can help the programmer giving an idea about the probable cause of the problem and bounding the search scope. Moreover, the categorization approach is a first step ahead of the completely automated categorization of nonconformances.

Capítulo 4

Avaliação da Geração Aleatória de Testes para Detecção de Não-Conformidades em Programas Baseados em Contratos

Neste capítulo apresentamos dois estudos experimentais [5]: primeiro avaliamos nossa abordagem RGT (JMLOK 2.0) com respeito a detecção de não-conformidades; então, comparamos duas abordagens RGT (JMLOK 2.0 e JET [16]) para verificação de conformidade em programas JML. Primeiro, apresentamos as questões de pesquisa utilizadas (Seção 4.1), então os resultados e as discussões dos estudos experimentais (Seções 4.2 e 4.3). Na Seção 4.4 descreve algumas ameaças à validade. Por fim, na Seção 4.5 são respondidas as nossas questões de pesquisa.

4.1 Research Questions

The goal of the first study is to analyze our RGT approach (JMLOK 2.0) with respect to conformance checking and manual categorization of nonconformances from the point of view of the developer in the context of contract-based programs. This study addresses the following research questions:

Q1. Is JMLOK 2.0 able to detect nonconformances in contract-based programs?

We measure the number of detected nonconformances in original sample and open-source contract-based programs.

Q2. What are the most common types of nonconformances, and their likely causes?

We analyze the types of nonconformances, investigating their likely causes.

Q3. What is the context, within the execution, in which we found nonconformances?

We measure and summarize metrics B and D (Equations 4.1 and 4.2, respectively). For each nonconformance (nc) we manually collected two metrics: *breadth* (B) and *depth* (D). The first measures the number of top calls within the test method until the failure occurs. This metric is defined in Equation 4.1; the $calls(tm)$ returns the sequence of method calls into the test method (tm). The second is the call *depth* (Equation 4.2) needed to find a given nonconformance – the internal calls performed until the contract is violated. For the method call that corresponds to the position on which the nonconformance was revealed, if the latter is in the body of this method, D receives 1, otherwise, its value is recursively increased until the method that reveals the nonconformance is called. For the example of Section 1.1.1 the metric values are: B = 4 – there are four calls until the failure occurs (see Source Code 1.2)) –, and D = 2 – the nonconformance is into `GenCounter.updateCount` method that is called into `MapMemory.updateMap` method.

$$B(nc, tm) = position(nc, calls(tm)) \tag{4.1}$$

$$\begin{aligned}
&D(nc, tm) = \\
&\quad let\ p = position(nc, calls(tm)) \\
&\quad let\ m = method(calls(tm)[p]) \\
&\qquad if\ (nc \in body(m))\ then\ result =\ 1 \\
&\qquad else\ result =\ 1 + D(nc, m)
\end{aligned} \tag{4.2}$$

The goal of the second study is compare two RGT approaches: JMLOK 2.0 and JET, for the purpose of evaluation with respect to their effectiveness in detecting nonconformances from the point of view of the developer in the context of contract-based programs. We chose JET for this comparison because to the best of our knowledge is the only tool for JML that does not require user inputs (as test data or implementation of functions). This study addresses the following research question:

Q4. Does the RGT-based approach perform better than the JET tool?

With this question, we intend to analyze the results from the two approaches in detecting nonconformances; the answer should provide a better view of their differences, helping developers in establishing the best scenario for each method.

4.2 First study - RGT in Isolation

In this section we present the first study, which provides evidences for the first three research questions (**Q1**, **Q2** and **Q3**).

4.2.1 Setup

The experimental units consist of sample programs available in the JML web site[1] and programs collected from several open-source JML projects; the experimental units totalize more than 29 KLOC and more than 9 K lines of JML specification (that we will refer as KLJML henceforth). The experimental unit named `Samples` is composed by 11 example programs for training purposes[2], because 2 other programs (`prelimdesign` and `jmlrefman`) could not be compiled. The `Samples` programs were written by specialists in the JML language. Furthermore, we gathered 6 open-source JML programs. While `Bomber` [60] is a mobile game, `HealthCard` [62] is an application that manages medical appointments into smart cards. `JAccounting` and `JSpider` are 2 case studies from the *ajml* compiler project [60], implementing, respectively, an accounting system and a Web Spider Engine. `Mondex` [65] is a system whose translation from original Z specification was developed in the Verified Software Repository[3] context. Finally, `TransactedMemory` [57] is a specific feature of the Javacard API. These experimental units are characterized in Table 4.1.

The study was performed in a PC with CPU Intel Core i7 2.20 GHz, RAM 8 GB, OS Windows 8 and Java 7 update 51. Once Randoop requires a time limit to generate tests – the time after which the generation process stops –, we used 10 seconds as basis, as in

[1] `http://www.eecs.ucf.edu/~leavens/JML/examples.shtml`

[2] `dbc`, `digraph`, `dirobserver`, `jmlkluwer`, `jmltutorial`, `list`, `misc`, `reader`, `sets`, `stacks`, `table`, and an adaptation of the subpackage `stacks – BoundedStack`

[3] `http://vsr.sourceforge.net/mondex.htm`

Tabela 4.1: Programs characterization in terms of Lines of Code (LOC) and Lines of JML specification (LJML).

Experimental Unit	LOC	LJML
Samples	3,400	5,200
Bomber	6,400	255
HealthCard	1,700	2,400
JAccounting	6,500	331
JSpider	8,800	386
Mondex	1,000	361
TransactedMemory	1,800	335
Total	**29,600**	**9,268**

previous work with JMLOK [71] and a simple bootstrap execution[4] – we suggest starts with a low value for the time limit and increases, like performed in Soares et al. [67]. For collecting data about test coverage we used EclEmma 2.3.0 (Eclipse plugin)[5] to collect Java instruction coverage, and manually collect the JML coverage – aided by EclEmma, counting the number of contracts (pre- and postconditions, invariants and constraints) covered by the tests. The categorization model from Section 3.2.1 was manually applied to each detected nonconformance. Furthermore, metrics *breadth* and *depth* were manually collected.

4.2.2 Results

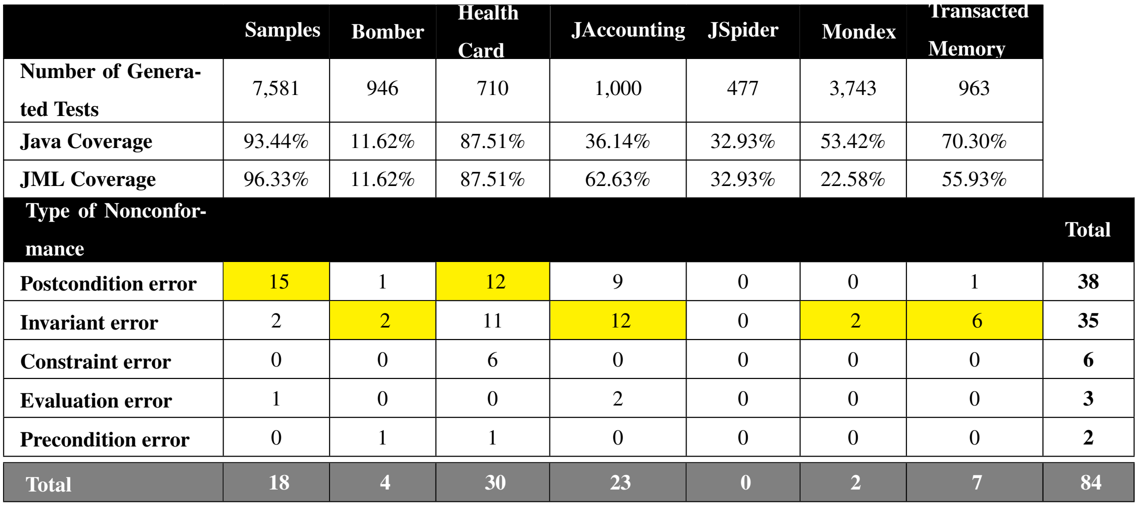
Table 4.2 presents the results of running JMLOK 2.0 for sample and open-source JML programs, including the number of generated test cases (considering the time limit of 10 seconds for tests generation), the test coverage, the number of detected nonconformances and their types. Further down, the detected nonconformances for each unit are listed.

For sample programs, 18 nonconformances were detected – 15 were categorized as *post-*

[4] In our bootstrap, we variate the time limit from 10 to 120 seconds. Once the nonconformances detected and the instruction coverage were the same for all executions (considering our experimental units), we used 10 seconds as time limit.

[5] `http://www.eclemma.org`

Tabela 4.2: For each experimental unit we present the number of generated test cases, test coverage, and all nonconformances detected, grouped by their types.



	Samples	Bomber	Health Card	JAccounting	JSpider	Mondex	Transacted Memory	
Number of Generated Tests	7,581	946	710	1,000	477	3,743	963	
Java Coverage	93.44%	11.62%	87.51%	36.14%	32.93%	53.42%	70.30%	
JML Coverage	96.33%	11.62%	87.51%	62.63%	32.93%	22.58%	55.93%	
Type of Nonconformance								**Total**
Postcondition error	15	1	12	9	0	0	1	**38**
Invariant error	2	2	11	12	0	2	6	**35**
Constraint error	0	0	6	0	0	0	0	**6**
Evaluation error	1	0	0	2	0	0	0	**3**
Precondition error	0	1	1	0	0	0	0	**2**
Total	**18**	**4**	**30**	**23**	**0**	**2**	**7**	**84**

condition errors, other 2 as *invariant*, and 1 as *evaluation*. For open-source JML projects, 66 nonconformances were detected: `Bomber` (4), `HealthCard` (30), `JAccounting` (23), `Mondex` (2), and `TransactedMemory` (7). In `JSpider` we did not discover nonconformances with the used setup (Section 4.2.1). The categorization was distributed in the following manner: 33 *invariant*, 23 *postcondition*, 6 *constraint*, 2 *evaluation*, and 2 *precondition*. The pie chart in Figure 4.1 shows the distribution of nonconformance types among all experimental units.

Additionally, the likely causes of the detected nonconformances are shown in Table 4.3. Most of the 84 detected nonconformances were categorized as *weak precondition* (38), followed by *Code error* (23). Three experimental units (`Bomber`, `JAccounting`, `Mondex`) presented *Code error* as the most frequent problem. *Strong postcondition* is the main cause of nonconformances in the sample programs. The pie chart in Figure 4.2 shows the distribution of causes among all experimental units.

Table 4.4 presents the mean of metrics *breadth* (B) and *depth* (D) for each experimental unit and for all nonconformances in general. The mean of *breadth* metric vary from 1.50 (in Bomber and Mondex experimental units) to 5.43 (in TransactedMemory). The mean of *depth* metric vary from 1.43 (in JAccounting) to 2.94 (in Samples).

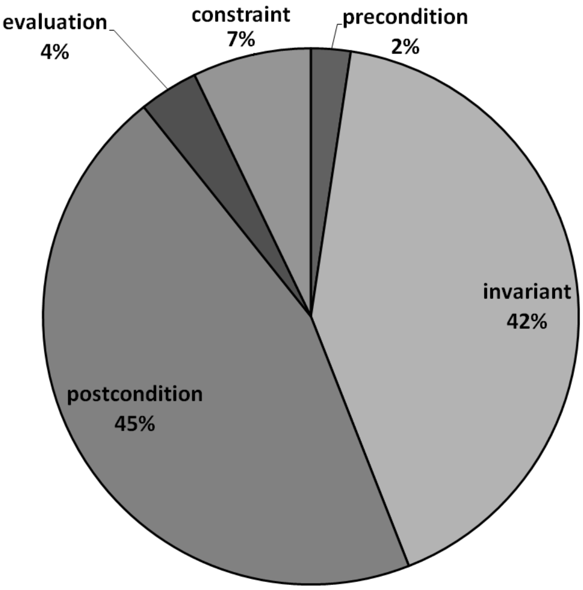


Figura 4.1: Nonconformances distributed between their types, for all experimental units.

4.2.3 Discussion

Discussion Q1. The RGT-based approach – applied by means of JMLOK 2.0 – was able to detect 84 nonconformances. The number of generated tests varied significantly for each experimental unit, maybe due to the test generator's approach. Randoop uses as input public methods or constructors (operations under test) for testing. It generates sequences of invocations, and uses the sequences to create tests and test data for further invocations. As a consequence, the number of generated test cases depends both on the number of available operations and the behavior of the experimental unit, which provides feedback to Randoop. For instance, `JSpider` has the lowest number of generated tests, maybe due to its nature (Web Engine), which requires some user interaction (same for `Bomber`); in contrast, `Mondex` had, among open-source JML programs, the highest number of generated test cases, as user interaction is not necessary. The generated sequences become a benefit of the approach, as several nonconformances were only detected by running a particular sequence of constructor and method calls. For instance, a postcondition error in `AbstractTransactedMemory` (a class from `TransactedMemory`) is only revealed after 32 specific method calls. In addition, test coverage results are also varying. While `Bomber` showed a very low value (maybe due to the need of user interaction), `Samples` and `HealthCard` presented the highest coverage rates.

Tabela 4.3: Likely Causes from detected nonconformances in the experimental units.

Nonconformance's Likely Cause	Samples	Bomber	Health Card	JAccounting	JSpider	Mondex	Transacted Memory	Total
Weak precondition	6	0	15	11	0	0	6	**38**
Code error	0	3	5	12	0	2	1	**23**
Strong postcondition	8	0	8	0	0	0	0	**16**
Undefined	4	1	0	0	0	0	0	**5**
Strong precondition	0	0	1	0	0	0	0	**1**
Strong constraint	0	0	1	0	0	0	0	**1**
Strong invariant	0	0	0	0	0	0	0	**0**
Weak postcondition	0	0	0	0	0	0	0	**0**
Total	**18**	**4**	**30**	**23**	**0**	**2**	**7**	**84**

Tabela 4.4: Results of mean for *breadth* and *depth* metrics to each experimental unit and for all nonconformances detected.

Experimental Unit	*breadth* (B)	*depth* (D)
Samples	2.22	**2.94**
Bomber	**1.50**	2.25
HealthCard	3.00	2.47
JAccounting	1.57	**1.43**
Mondex	**1.50**	1.50
TransactedMemory	**5.43**	2.14
Mean	**2.54**	**2.23**

The approach detected nonconformances to the set of example programs, written by specialists in the JML language. Despite their best efforts, subtle nonconformances remained in the contract and/or programs; some of those were indeed hard to catch only with visual analysis or simple tests. For instance, we detected 8 nonconformances in methods that invoke, in their contract, `JMLDouble.approximatelyEqualTo`; `JMLDouble` is imported from the standard JML API, being only visible on the contract level; its method `approximatelyEqualTo` performs precise comparison between two values. The tolerance value (constant related to error rate) can be inappropriately small (only 0.005); or, this postcondition is too strong; or, the implementation of type `JMLDouble` is too restrictive. All these possible reasons show how hard it is to detect those kinds of nonconformances.

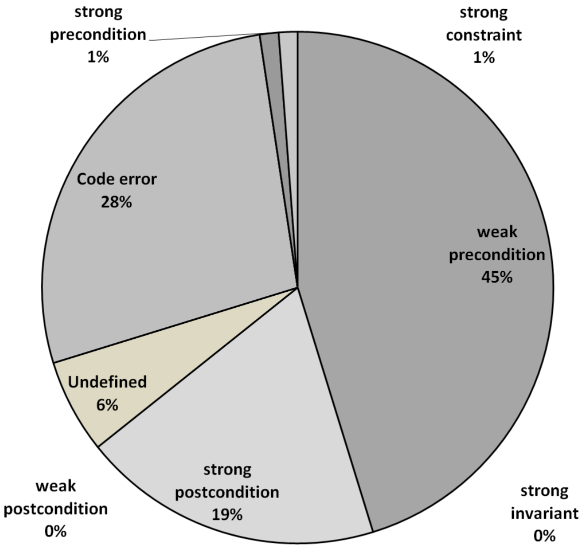


Figura 4.2: Nonconformances distributed between their likely causes, for all experimental units.

Concerning open-source JML programs, we contacted all authors to report the results. From those, two authors (the author of `Bomber, JAccounting` and `JSpider`; and the author of `TransactedMemory`) responded positively, confirming that the results were indeed previously-undetected nonconformances. Concerning `Mondex`, the authors reported that the nonconformances detected were already known by developers; however, the version with those nonconformances remained available. In `HealthCard` study, the author answered to our contact informing that he had not worked with JML for some years, and did not want to discuss the nonconformances.

Discussion Q2. From the 84 detected nonconformances, the most frequent type was the *postcondition* – 38, followed by *invariant* violations – 35. These numbers indicate that most violations occur at the exit of operations, even if the cause is not in the operation itself. There are several possible explanations: specifiers expect certain behavior to which the code fails to comply, or a chain of previous calls fails to avoid certain undesirable states. More severe contract errors were not significantly frequent (only three were evaluation errors). A few nonconformances were related to history constraints (six) in the only program that actually uses those constraints (`HealthCard`). This result could indicate that using those constraints is not trivial, with questionable usefulness – they often can be replaced by an invariant.

Regarding likely causes, our manual classification found *Weak precondition* as the main cause for the experimental units (with 38 instances). In this case, a method allows values that should be denied for the correct execution of the method – as we could infer from the information available in the program. For instance, the default precondition (*requires true*) is often used in the experimental unit `JAccounting`. Maybe *Weak precondition* has been the most common case of likely cause because it is complex to specify preconditions; once the specifier does not previously know the clients of his system, an overly strong precondition deny access to several clients; on the other hand, if the precondition is too weak, several clients will not be able to get the expected result. So this represents an important trade-off for contract-based programs. However, *Code error* is also very recurrent, with 23 instances. This problem is probably related to different levels of abstraction between programming language and contract language, in which the coder may not understand a contract written by someone else, or even the contract may be too complex to be implemented; or there can be a synchronization issue between code and contract evolution. For instance, in `HealthCard` we observed that a change in the code concerning dates made a class invariant obsolete, which resulted in a nonconformance.

Discussion Q3. The results presented in Table 4.4 suggest that *breadth* and *depth* are, in average, higher than 1, for all experimental units. Those numbers suggest that tests have a higher chance of success in detecting nonconformances when the test cases present higher horizontal (call sequences) and vertical (internal calls) complexity. The method of test generation used in our approach favors that property. The highest values were obtained on `HealthCard`. For instance, *breadth* = 32 for class `AbstractTransactedMemory`, where several modifications into objects under test were performed until the nonconformance emerged; and *depth* = 8 into the class `GenGenbyte` where several calls are made until the invariant problem arises. This experimental unit also had the biggest mean for *breadth* (5.43), whereas `Samples` presented the biggest mean for *depth* metric (2.94).

The smallest mean for *breadth* (1.5) and *depth* (1.43) metrics were obtained on `Bomber`, `Mondex` and `JAccounting`; this result indicates that the nonconformances in these units were simpler to detect. It seems reasonable to conclude that there is a relationship between those metrics and the most frequent type of nonconformance present in those units – invariant, which specifies that all fields of the class must be non-null (JML default behavior). A single execution of a constructor that does not initialize all fields conflicts with the assertion based on this invariant. The complete manual categorization performed is available in our technical report [51].

4.3 Second Study - JMLOK 2.0 versus JET

In this section we present the second study, which provides evidences for the fourth research question (**Q4**).

4.3.1 Setup

JET aims at applying dynamic testing to conformance checking in JML, by randomly generating test cases using contracts as test oracles. Genetic algorithms are applied for automatically building all test data that exercise runtime assertions. This choice is promisingly effective, although it raises the risk of nondeterminism in generating test cases and data on successive executions of the tool. We chose JET because was the only functional tool that we have found for Java not requiring any additional inputs.

In this study, we evaluate one factor with two treatments (JMLOK 2.0 and JET). We chose a paired comparison design for the study, that is, the experimental units were applied to all treatments. This comparison considered only a subset of the experimental units from the first study, due to JET requirements [16] (no public fields can be assigned, and object sharing is not allowed). The units were `Samples`, `JAccounting`, `Mondex` and `TransactedMemory`, totalizing over 6 KLOC and 5 KLJML; from `JAccounting`, only the main class of the system, `Account`, was considered, once this class does not have dependencies, so can be used with JET tool. This study was performed in the same machine setup of the first study (Section 4.2.1). We used the JMLOK 2.0 and JET tools with their default configurations.

4.3.2 Results

Table 4.5 presents the results of the experimental study considering JMLOK 2.0 and JET. The total number of nonconformances detected by JET was 9, against 30 nonconformances detected by JMLOK 2.0. Only for the `JAccounting` experimental unit JET detected a nonconformance that was not detected by JMLOK 2.0, the others 8 nonconformances were also detected by JMLOK 2.0. In relation to test coverage, only for `JAccounting` JET presented higher coverage for both Java and JML coverage.

Tabela 4.5: Comparison between JMLOK 2.0 and JET. For each experimental unit is presented the number of tests generated by each tool, test coverage (Java and JML), number and types of nonconformances.

		Samples	**JAccounting**	**Mondex**	**TransactedMemory**
JET	Tests	8,306	1,787	700	1,958
	Java Coverage	62.86%	100%	7.50%	21.53%
	JML Coverage	63.70%	100%	11.60%	52.59%
	# NCs	4	4	0	1
	Types	2 invariant 2 postcondition	4 postcondition	—	1 invariant
JMLOK 2.0	Tests	7,581	1,000	3,743	963
	Java Coverage	93.44%	96.60%	53.42%	70.30%
	JML Coverage	96.33%	95.83%	22.58%	55.93%
	# NCs	18	3	2	7
	Types	15 postcondition 2 invariant 1 evaluation	1 postcondition 2 evaluation	2 invariant	6 invariant 1 postcondition

4.3.3 Discussion

With objective of perform a statistic comparison of the tools, to discover which is the more effective (in relation to number of nonconformances detected and tests coverage) we analyze the data profile to choose the more suitable test to be used.

Firstly, we performed an investigation about the possibility of use the *Paired t-test* [40] because the test is the better parametric test to compare two samples; to use the *Paired t-test* our data need have two features: Normality and Homoscedasticity. Thus, we performed a normality test *Shapiro-Wilk* [66] and get the following results:

- To Java coverage data:
  - With JMLOK 2.0 we obtained a p-value = 0.44, do not being possible reject the null hypothesis of data Normality;

  - With JET we obtained a p-value = 0.67, too do not being possible reject the null hypothesis.

- To JML coverage data:

  - With JMLOK 2.0 we obtained a p-value = 0.29, do not being possible reject the null hypothesis of data Normality;
  - With JET we obtained a p-value = 0.94, too do not being possible reject the null hypothesis.

- To number of nonconformances data:

  - With JMLOK 2.0 we obtained a p-value = 0.04, rejecting the null hypothesis of data Normality;
  - With JET we obtained a p-value = 0.16, do not being possible reject the null hypothesis of data Normality.

Since only the number of nonconformances data of JMLOK 2.0 was not Normal, the use of *Paired test-t* was discarded only for compare the number of nonconformances detected by the tools. We used the Homoscedasticity test *Bartlett* and we obtained the following results:

- To Java coverage data we obtained a p-value = 0.27, do not rejecting the null hypothesis of data Homoscedasticity;
- To JML coverage data we obtained a p-value = 0.97, do not rejecting the null hypothesis of data Homoscedasticity;
- To number of nonconformances data we obtained a p-value = 0.01, rejecting the null hypothesis of data Homoscedasticity.

Once data of coverage Java and JML are Normal, we use *Paired test-t* to compare the two tools; and to compare the data of number of detected nonconformances, we use *Wilcoxon signed-rank*. The analysis statistic revealed that JMLOK 2.0 is better than JET in relation to number of nonconformances detected and the coverage (Java and JML) from tests generated by the tools using *Paired test-t* and *Wilcoxon signed-rank* tests, obtain that the JMLOK 2.0

tool is more effective than JET tool with 95% of confidence level and with p-value = 0.94 to the number of nonconformances; p-value = 0.96 to Java coverage; and p-value = 0.86 to JML coverage (considering the null hypotheses that the number of nonconformances detected by JMLOK 2.0 is greather than equal the number of nonconformances detected by JET and that the coverage (Java and JML) of JMLOK 2.0 tests are greather than equal the JET tests coverage (Java and JML) - because the null hypotheses that the values were equals were rejected). Thus, the answer of question **Q4.** is: the JMLOK 2.0 tool is more effective than JET tool with 95% of confidence level.

Discussion Q4. JET was able to reveal unseen nonconformances, specially for the `Samples` and `JAccounting` experimental units. However, we observed an important drawback: the tool is inconstant about the nonconformances discovered; for instance, on `JAccounting`, different executions found different nonconformances: JET often detects four nonconformances for a given unit, then in the next execution finds no nonconformances; for the same unit JMLOK 2.0 always finds three nonconformances. Furthermore, the nonconformances detected by the tools are the same in the majority of the cases; but in the `JAccounting`, there were two cases in which JET and JMLOK 2.0 differ in the type assigned to the nonconformances: while JMLOK 2.0 assigned Evaluation, JET assigned Postcondition; maybe this difference can be related to the compilers used by tools (*jmlc*, in JMLOK 2.0, and an extension of *jmlc* in JET). The genetic algorithm in the backend makes JET differ between repeated executions. This property was not observed in the RGT-based approach (JMLOK 2.0), despite its randomness.

Considering the test coverage, in general JMLOK 2.0 performed better than JET. The only case in which JET showed better results was `JAccounting`. This result can be related to JET constraints: no public fields can be assigned, and object sharing is not allowed; the tests miss several parts from the programs that do not fulfill those requirements, which does not occur with JMLOK 2.0. Considering the number of nonconformances detected, the only case where JET performs better than JMLOK 2.0 was also `JAccounting`: four against three. Therefore, considering number of nonconformances detected and tests coverage, JMLOK 2.0 performs better than JET for our experimental units, with a confidence level of 95% by means of *Paired test-t* and *Wilcoxon signed-rank* tests. In other units, these results may vary, although JET limitations prevent tests with dependencies between classes and packages, and the use of external libraries. We believe that an approach that uses the best features of both tools would be suitable for the purpose of nonconformance detection.

4.4 Threats to validity

In the context of external validity, the first threat is the number of JML programs analyzed: six in the first study, and three in the second; however the total size of experimental units (more of 29 KLOC and more of 9 KLJML to evaluate JMLOK 2.0 and over 6 KLOC and more of 5 KLJML to evaluate JET and compare it with JMLOK 2.0) is higher than other studies with JML tools [71] [16]. Other threat is the categorization model used to manually classify the nonconformances discovered; to address this limitation, the nonconformances discovered, its categories and likely causes were reported to experimental units authors, and most authors agreed with our categorization model.

For conclusion validity, once the tools both use randomness, we used five runs for each unit and collected the best result (as JET differs between executions in relation to the nonconformances detected); on the other hand this problem was not observed in JMLOK 2.0 (the same setup detects the same nonconformances in several runs). Most importantly, JMLOK 2.0 (and also JET) is test-based. We thus cannot argue about its completeness in finding all nonconformances that could be found in the experimental units. Therefore, any generalization about the types of nonconformances, categories and likely causes is out of the question. Still, we believe that the results show reasonable trends that could be the starting point for similar studies involving contract-based programs.

4.5 Answers to the research questions

From our results, we made the following observations:

- **Q1.** Is JMLOK 2.0 able to detect nonconformances in contract-based programs?

  Yes, the JMLOK 2.0 tool was able to detect 84 nonconformances. From those, 18 nonconformances were detected in sample programs and 66 in open-source JML programs.

- **Q2.** What are the most common types of nonconformances, and their likely causes?

  From the 84 detected nonconformances, the most frequent type was the *postcondition* – 38, followed by *invariant* violations – 35. Regarding likely causes, our manual classification found *Weak precondition* as the main cause for the experimental units, with 38 instances. Maybe *Weak precondition* has been the most common case of likely cause because it is complex to specify preconditions; once the specifier does not previously know the clients of his system, an overly strong precondition deny access to several clients; on the other hand, if the precondition is too weak several clients will not be able to get the expected result. So this represents an important trade-off for contract-based programs.

- **Q3.** What is the context, within the execution, in which we found nonconformances?

  The results presented in Table 4.4 suggest that *breadth* and *depth* are, in average, higher than 1, for all experimental units. Those numbers suggest that tests have a higher chance of success in detecting nonconformances when the test cases present higher horizontal (call sequences) and vertical (internal calls) complexity. The method of test generation used in our approach favors that property.

- **Q4.** Does the RGT-based approach perform better than the JET tool?

  Yes, considering number of nonconformances detected and tests coverage, JMLOK 2.0 performs better than JET for our experimental units, with a confidence level of 95%. In other units, these results may vary, although JET limitations prevent tests with dependencies between classes and packages, and the use of external libraries. We believe that an approach that uses the best features of both tools would be suitable for the purpose of nonconformance detection.

Capítulo 5

Avaliação da Categorização de Não-Conformidades

Neste capítulo, também apresentamos dois estudos experimentais [5]: no primeiro avaliamos a abordagem de categorização automática proposta (por meio da ferramenta JMLOK 2.0) em comparação com a categorização manual feita para o estudo da Seção 4.2; no segundo, comparamos a categorização automática com a categorização realizada por JML experts voluntários. Na Seção 5.1 são apresentadas as questões de pesquisa, então nas Seções 5.2 e 5.3 são apresentados os resultados e discussões dos estudos experimentais. Na Seção 5.4 são descritas algumas ameaças à validade. Por fim, na Seção 5.5 são respondidas as nossas questões de pesquisa.

5.1 Research Questions

The goal of the first study is to analyze our automatic categorization approach (by means of JMLOK 2.0 tool) with respect to coincidences with our manual categorization (the baseline) from the point of view of the developer in the context of contract-based programs.

This study addresses the following research questions:

Q1. How many answers from tool are coincident with our manual analysis performed previously?

We measure *matches* (Equation 5.1) and perform a hypothesis testing for the mean value.

$$matches(x) = \frac{Total\ of\ Agreements(x)}{Total\ of\ Categorized\ Nonconformances(x)} \tag{5.1}$$

where x is an experimental unit; $Total\ \ of\ \ Agreements$ is the total of coincidences between automatic and manual categorization of nonconformances; and $Total\ of\ Categorized\ Nonconformances(x)$ is the total of categorized nonconformances for both approaches – manual (baseline) and automatic.

Hypothesis null and alternative In order to answer this research question we formulate the following hypothesis:

$$H_0 : \mu(matches) = 0 \qquad\qquad H_1 : \mu(matches) \neq 0 \tag{5.2}$$

Regarding statistical tests, we use the *Shapiro-Wilk test* [66] to test data normality, if the data are normal we use the *One Sample t-test* [40], otherwise the *Wilcoxon signed-rank test* [40]; in either cases, with a confidence level of 95%. If we reject the null hypothesis, we will use a two-tailed hypothesis test to verify whether the mean of *matches* is greater than zero.

Q2. What is the relationship between *matches* and *depth* of test execution?

This complementary investigation aims at characterizing whether the coincidences between manual and automatic categorization approaches are related to the deeply of the nonconformance revelation. Therefore, we compare the values of *matches* and *depth* metrics.

The goal of the second study is compare our categorization approach with the results from voluntary JML experts, from the point of view of the researcher in the context of contract-based programs. This study addresses the following research question:

Q3. How many answers from tool are coincident with JML experts categorization?

We asked some voluntary JML experts[1] to categorize 10 nonconformances randomly selected. Then we compare their results with the automatic categorization.

[1] The JML experts have worked with JML and DBC for more than two years.

5.2 First Study - Comparison between Manual and Automatic Categorization

In this section we present the first study, which provides evidences for the first two research questions (**Q1** and **Q2**).

5.2.1 Setup

The population considered in this comparison is the 84 nonconformances manually categorized in Section 4.2. This study was performed using the same machine setup of Section 4.2.1. In the automatic categorization, we use the JMLOK 2.0 with time limit of 10 seconds, considering the same experimental units from Section 4.2. We use the R[2] statistical tool [10] to perform hypothesis testing for **Q1**.

5.2.2 Results

Table 5.1 presents the results of *matches* for each experimental unit[3]. We used the *Shapiro-Wilk test* to check if the data came from a normally distributed population. The *p-value* resultant was 0.0031, indicating that our data did not come from a normally distributed population. As the use of the *One Sample t-test* was discarded; we compare the median of *matches* with zero by means of the *Wilcoxon signed-rank test*. The *p-value* resultant was 0.0071, rejecting our null hypothesis (Equation 5.2) that the mean of *matches* is equal to zero, with a confidence level of 95%. Then, we reformulated our hypothesis (Equation 5.3) and calculate the *p-value*, considering as alternative hypothesis that the mean of *matches* metric is greater than 0:

$$H_0 : \mu(matches) = 0 \qquad\qquad H_1 : \mu(matches) > 0 \tag{5.3}$$

With the two-tailed test, the resultant *p-value* was 0.0035, also rejecting the null hypothesis, that mean of *matches* is equal to zero, with a confidence level of 95%.

[2] R is a free software environment for statistical computing and graphics. Available online at `http://www.r-project.org/`

[3] once `Samples` is composed by 11 packages, we considered each package as an experimental unit

Tabela 5.1: *matches* results for each experimental unit and the mean of this metric. The metric was obtained using the Equation 5.1.

Experimental Unit	***matches***
Samples.BoundedStack	1.00
Samples.stacks	1.00
Samples.dbc	0.00
Samples.misc	1.00
Samples.list	0.20
Bomber	0.50
HealthCard	0.63
JAccounting	1.00
Mondex	1.00
TransactedMemory	1.00
Mean	**0.73**

Interested in investigate whether there is a relationship between metrics *matches* and *depth* (number of internal calls until a nonconformance occurrence), we performed a correlation test by means of *Spearman's* rank correlation coefficient. This test was specially designed for nonparametric distributions [40]. We obtained ρ = -0.77, allowing us assert at 95% of confidence level, that there is a strong negative relationship between these metrics. So, when *depth* increases, *matches* tend to decreases, and vice versa; in other words, we found that to our experimental units, when a nonconformance is more internal – considering the AST of the language, the coincidence (*matches*) between our automatic approach and the baseline (our manual categorization) is smaller. Table 5.2 shows *matches* and *depth* metrics and the value of ρ.

The complete results of the manual categorization are available in Appendix A. And in Appendix B we present the results of the automatic categorization.

Tabela 5.2: Metrics *matches* and *depth* for each experimental unit. For Samples we calculate the metrics for each package used – BoundedStack, stacks, dbc, misc, and list. The *Spearman's* coefficient (last line of table) indicates a strong negative relation between these metrics.

Experimental Unit	***matches***	***depth***
Samples.BoundedStack	1.00	2.00
Samples.stacks	1.00	2.00
Samples.dbc	0.00	3.25
Samples.misc	1.00	3.00
Samples.list	0.20	3.20
Bomber	0.50	2.25
HealthCard	0.63	2.47
JAccounting	1.00	1.43
Mondex	1.00	1.50
TransactedMemory	1.00	2.14
ρ	**-0.77**	

5.2.3 Discussion

Discussion Q1. The mean of *matches* metric – used to compare the results from manual and automatic categorization – was 0.73. Nevertheless, there were two cases in which the metric was very low. These cases occurred in `Samples` experimental units, in which there were *matches* = 0.00 (`dbc`) and *matches* = 0.20 (`list`).

In the `dbc` package, this result occurred because the lack of a semantic analysis that could give a more precise result. Using our set of heuristics, the automatic approach assigned to all nonconformances as likely cause *Weak precondition*, because there are no preconditions to the methods in which the nonconformances were detected. On the other hand, the manual analysis assigned *Strong postcondition*, as a result of the low tolerance value used in the postcondition (0.005), to compare two decimal values using the `JMLDouble.approximatelyEqualTo`[4], that can be the responsible to these noncon-

[4] `JMLDouble` is imported from the standard JML API, being only visible on the contract level

formances.

In the `list` package, the low *matches* metric is due to the fact of manual categorization assigned *Undefined* as likely cause, whereas the automatic categorization assigned *Weak precondition*. This difference occurred because the manual analysis was not able to understand whether the problem arises from a code or a specification problem; on the other hand, the automatic categorization always returns a likely cause to the nonconformance.

On the other hand, there were six experimental units (`BoundedStack`, `stacks`, `misc`, `JAccounting`, `Mondex`, and `TransactedMemory`) from which the highest possible *matches* were obtained. Additionally, in the other two units (`Bomber`, and `HealthCard`) *matches* = 0.50, and *matches* = 0.63, respectively. Those results show that, although we are using a heuristics-based approach, our automatic categorization has good results in comparison with the baseline (our manual categorization). In order to get a more reliable result about our categorization approach, we perform an experimental study (Section 5.3) to compare our results with the results from voluntary JML experts.

In contract-based methods with tricky postconditions; if the precondition is default, our categorization resolves *Weak precondition* as the likely cause. This is a limitation of our approach, but we believe that if a method has a complex postcondition, this method needs some precondition to guarantee properties needed to satisfy the postcondition. Concerning in differences related to *Undefined* problems, we believe that inserting our approach in development phase of a project this problem could be avoided, once that, possibly the developer can find the real source of the nonconformance using our automatic categorization approach. In this context, the developer can use the outcome from JMLOK 2.0 as a starting point to the process of nonconformances correction.

Those results are important to discuss the limitations of our approach: once that context knowledge may be important to understand the real source of the contract violations, a heuristics-based approach hardly is able to find the real source of the contract violations. Furthermore, our heuristics-based approach returns as likely cause the first match found; and we observed that a more detailed analysis of the contract-based program may be necessary. Nevertheless, our approach is a step ahead of contract violations correction aided by automation.

Discussion Q2. The *Spearman's* coefficient indicates, at 95% of confidence level, a strong negative relation between *matches* and *depth* metrics. So, when a metric increases the other decreases, and vice versa. This result allows assert, at 95% of confidence level, that a greater number of internal calls until a nonconformance occurrence decreases the coincidence between manual and automatic categorization, maybe this occurs because in those cases the semantic knowledge affects directly on manual result.

Maybe this result is related to the fact that when the nonconformance is revealed directly on method body, the manual categorization process is more similar to heuristics-based approach, basically analyzes the contract from current class and suggests a likely cause based on this analysis; on the other hand, when the nonconformance is revealed in a deeper level, the manual analysis can observe some properties not detectable by the heuristics-based approach, resulting in a difference between their results. We believe that this result may be generalized for others projects, once that in projects from different domains we had a high *matches* when *depth* was smaller.

5.3 Second Study - Comparison between Automatic and JML Experts Categorization

In this section we present the second study, which provides evidences for the third research question (**Q3**).

5.3.1 Setup

From the 84 nonconformances that we discovered in our experimental units (Section 4.2) we randomly selected 10 (using the *sample* command from the R statistical tool (version 3.0.1) [10]). These nonconformances are presented on Table 5.3. The detected nonconformances were ordered in the following manner: first we ordered the detected in sample programs – starting with `BoundedStack` package, then `stacks`, `dbc`, `misc` and finally `list`; then `Bomber`, `HealthCard`, `JAccounting`, `Mondex` and `TransactedMemory`. In each experimental unit the nonconformances were ordered alphabetically by the name of the methods where the nonconformances were detected. All numbered nonconformances are available in Appendix C.

Tabela 5.3: Randomly selected nonconformances released to JML experts categorize. The nonconformance number corresponds to the position of the nonconformance considering our 84 nonconformances – the counting starts in sample programs (`BoundedStack`) and continues until the last nonconformance discovered in `TransactedMemory` unit. Experimental Unit gives the name of the experimental unit. Class and Method columns give information about location of the nonconformance into the experimental unit. Finally, column Type gives the type of the nonconformance.

NC number	Experimental Unit	Class	Method	Type
1	**Samples**	BoundedStack.BoundedStack	Constructor	postcondition
4		stacks.BoundedStack	Constructor	invariant
10		dbc.Rectangular	imaginaryPart	postcondition
18		list.list3.TwoWayIterator	next	postcondition
26	**HealthCard**	allergies.Allergies_Impl	setAllergyDesignation	postcondition
47		treatments.Treatment_Impl	setTreatmentID	invariant
48		vaccines.Vaccine_Impl	setDesignation	precondition
49		vaccines.Vaccines_Impl	getVaccineDesignation	postcondition
57	**JAccounting**	com.spaceprogram.util.ArrayUtils	getMaxIntArrayIndex	postcondition
64		com.spaceprogram.util.CookieUtils	getDeleteCookie	postcondition

The form that we use to ask the JML experts is available in Appendix D. The form has the following structure: first, we present the proposed three-level model for categorization and an overview of this model; next, a methodology to performs the manual categorization of nonconformances is suggested; then, for each nonconformance selected we present some details about the nonconformance – information about location (experimental unit, package, class, and method) and the nonconformance's type –, a link for the contract-based program corresponding, and a test case that reveals the nonconformance; finally, we ask to JML expert give a categorization for the nonconformance, choosing a category for the nonconformance – between Specification error, Code error or Undefined; and suggesting a likely cause.

5.3.2 Results

We had three voluntary JML experts (that we will refer as Subjects henceforth). Once that was used as a text field for likely cause, we mapped the Subjects answers to our set of likely causes. Comments like "the precondition should be stronger" were mapped to *Weak precondition*; "not identified", and "Couldn't get hold of the cause" were mapped to *Unde-*

fined; “code should set the position pointer correctly”, and “Code seems to not prepare the argument to call method correctly” were mapped to *Code error*. Table 5.4 shows Subjects answers and our automatic results.

Tabela 5.4: Column # NC displays the nonconformance number (the number is the same presented on Table 5.3). Column Type shows the nonconformances type. For each Subject we present the Category and Likely Cause for each categorized nonconformance. Finally, Automatic results shows the results of our automatic categorization for each categorized nonconformance. The line *matches* presents the *matches* metric for each Subject in comparison with tool results. We use the following acronyms for type: pre for precondition, post for postcondition, and inv for invariant; and for category: Spec for Specification error, Code for Code error, and Undef for Undefined.

		Subject 1		Subject 2		Subject 3		Automatic results	
# NC	Type	Category	Likely Cause	Category	Likely Cause	Category	Likely Cause	Category	Likely Cause
1	post	**Spec**	**weak pre**	**Spec**	**weak pre**	**Spec**	**weak pre**	Spec	weak pre
4	inv	**Spec**	**weak pre**	**Spec**	**weak pre**	Code	Code	Spec	weak pre
10	post	Undef	Undef	Spec	Undef	**Spec**	**weak pre**	Spec	weak pre
18	post	Code	Code	Spec	weak post	Undef	Undef	Spec	weak pre
26	post	Undef	Undef	Undef	Undef	Code	Code	Spec	weak pre
47	inv	**Spec**	**weak pre**	Undef	Undef	Code	Code	Spec	weak pre
48	pre	Code	Code	Undef	Undef	Undef	Undef	Spec	strong pre
49	post	Undef	Undef	**Spec**	**weak pre**	**Spec**	**weak pre**	Spec	weak pre
57	post	**Spec**	**weak pre**	Spec	weak post	Code	Code	Spec	weak pre
64	post	Code	Code	Spec	Undef	Undef	Undef	Spec	weak pre
matches		**0.40**		**0.30**		**0.30**			

According to Table 5.4, there were four coincidences between our automatic categorization approach and Subject 1, for both category and likely cause; that corresponds to a *matches* = 0.40. Concerning the Subject 2, our automatic categorization had seven coincidences considering category, and three coincidences considering likely cause; corresponding a *matches* = 0.30. Finally, with the Subject 3, our automatic categorization had three coincidences for both category and likely cause; corresponding a *matches* = 0.30.

5.3.3 Discussion

We had few respondents to our form because manual categorization of nonconformances is time consuming (a people need much time to understand the code and the contract, and need some time to suggest a likely cause for the nonconformance). Nevertheless, the answers were important to highlight the difficulty of manually categorize nonconformances and the usefulness of our automatic categorization approach, as a first step to help the developer in the process of nonconformances correction.

Concerning the *matches* metric we found that our approach is a little bit similar to JML experts categorization; furthermore, the most similar result occurred with Subject 1 – *matches* = 0.40; with Subject 2 and Subject 3 the *matches* (0.30) was the same.

Moreover, only for nonconformances numbers 26 (postcondition problem at the method `setAllergyDesignation` from class `Allergies_Impl` – in `HealthCard`), and 48 (precondition problem at the method `setDesignation` from `Vaccine_Impl` – also in `HealthCard`) we did not have any JML expert answer equal to our automatic categorization. Maybe this had occurred because these nonconformances are complex to understand (e.g. in the contract of `setAllergyDesignation` there are 10 JML clauses – for specify pre- and postconditions to normal and exceptional behaviors, to declare which field can be assigned in the method and to specify which types of exceptions can be thrown. Furthermore, there is the use of a model method from `Common` interface that manipulates elements from JML API (`JMLValueSequence` and `JMLByte`). Therefore, we believe that only a manual analysis of this system could give a more precise categorization. Two Subjects could not assign a likely cause for both nonconformances, assigning *Undefined* for category and likely cause. Considering likely cause, there were four cases in which we did not have any coincidence between the automatic approach and Subjects answers: for nonconformances 18 (postcondition problem at the method `next` from class `TwoWayIterator` – in package `list` from `Samples`), 26, 48, and 64 (postcondition problem at the method `getDeleteCookie` from class `CookieUtils` – in `JAccounting`).

These differences between the automatic categorization and the results from Subjects, are justified by the fact that we have used a heuristics-based approach to categorize nonconformances. Moreover, by the fact that our automatic approach does not have semantic information about the systems and their contracts; so, in some cases we expected differences between automatic and manual results (conforms to discussion from Section 5.2). Thus, we believe that our approach may be used in the process of nonconformances correction, as a first step to find the actual problem that occasioned the nonconformance.

5.4 Threats to Validity

In the context of external validity, our results are valid only for our experimental units and can change considerably in others experimental units. Additionally, our categorization approach consider only external behavior from the methods; but, we intend to improve our model to consider internal behaviors as a future work.

In the context of conclusion validity, once the JML expert that proposes the manual categorization process was the developer of the automatic categorization approach, we perform a validation with others JML experts to evaluate the heuristics used and the approach in general. Unfortunately, we had only three answers to our form. We believe that we did not have more answers because manually categorize nonconformances is time consuming and the experts invited to answer the form possibly did not have enough time. Despite that, the answers that we received were similar to the automatic categorization. Moreover, the differences between the answers demonstrate that an automatic approach to categorize nonconformances can be useful.

Furthermore, our categorization approach is heuristics-based, so not always the result from automatic and manual categorization will be the same (as were shown in Sections 5.2 and 5.3). Nevertheless the approach had promising results and is a step ahead of help the programmer in the nonconformances correction process; once that, using our approach the programmer already will have an idea about the source of the problem.

5.5 Answers to the research questions

From our results, we made the following observations:

- **Q1.** How many answers from tool are coincident with our manual analysis performed previously?

The mean of *matches* metric – used to compare the results from manual and automatic categorization – was 0.73.

- **Q2.** What is the relationship between *matches* and *depth* of test execution?

  The *Spearman's* coefficient indicates, at 95% of confidence level, a strong negative relation between *matches* and *depth* metrics. So, when a metric increases the other decreases, and vice versa. This result allows assert, at 95% of confidence level, that a greater number of internal calls until a nonconformance occurrence decreases the coincidence between manual and automatic categorization, maybe this occurs because in those cases the semantic knowledge affects directly on manual result.

- **Q3.** How many answers from tool are coincident with JML experts categorization?

  For Subject 1, we have a *matches* 0.40; for Subjects 2 and 3 the *matches* was 0.30.

Capítulo 6

Considerações Finais

Neste capítulo são apresentados os principais resultados deste trabalho, os trabalhos relacionados e as sugestões para trabalhos futuros.

6.1 Conclusions

In this work, we present an approach for detect and categorize nonconformances in contract-based programs, aiming help the programmer in the process of nonconformances correction. We performed four experimental studies. In the first two studies we evaluated JMLOK 2.0 tool in relation to nonconformances detection and our manual categorization model for nonconformances; and compared JMLOK 2.0 with JET concerning nonconformances detection. In the last two studies we evaluated our categorization model by means of our implementation as a module to JMLOK 2.0 tool, comparing the results of the tool with our manual results (baseline); and also by comparing our results with the categorization performed by voluntary JML experts.

In the first study, the RGT-based approach – by means of the JMLOK 2.0 implementation – is applied to sample and open-source JML projects, in order to demonstrate the applicability of the approach in detecting overlooked nonconformances in those programs. Second, we compare the effectiveness of JMLOK 2.0 with the results of JET [16], the most similar detection tool for JML.

The RGT-based approach presents promising results, as, in more of 29 KLOC and more of 9 KLJML, 84 nonconformances were detected. We reported those nonconformances and their classification to authors, and answers were mostly positive. Only `HealthCard`'s developer did not want to answer about our categorization model, because he has not worked with JML for some years. Furthermore, we classified the nonconformances and established likely causes – postcondition violations were the most frequent detected type, and most causes stay between *Weak preconditions* (mostly related to the absence of preconditions for the methods) and *Code errors* (mostly related to null fields) in our experimental units. We also found that, in the context of our experimental units, most nonconformances are hard to detect without sequences of modifications into the object under test, with the results of metrics *breadth* and *depth*; showing evidence for the need of a more complex test structure in nonconformance detection than only one modification in the object under test.

When comparing JMLOK 2.0 with JET, the first detected 30 nonconformances with Java instructions coverage of 78.44% and JML instructions coverage of 67.67%, while JET detected 9 nonconformances by covering 47.97% of Java instructions and 56.97% of JML instructions; for the same experimental units (a subset from the first study, totalizing approximately 6 KLOC and 5 KLJML). These numbers suggest that JMLOK 2.0 performs better than JET considering the number of nonconformances detected and test coverage (block instructions coverage), for the experimental units. In addition, we observed that the nonconformances detected by JET differ between repeated executions, maybe due to the nature of its genetic algorithms – this property was not observed in the proposed approach. So we can conclude that the proposed approach is more stable than JET, considering the same setup and the same experimental unit, JMLOK 2.0 always found the same nonconformances.

Third, we compared the coincidences – *matches* – between the automatic categorization (by means of JMLOK 2.0 tool) and our manual categorization (baseline), and we got a mean *matches* of 0.73. Additionally, we verify that knowledge of the context may be important to assign likely cause; so, this lack of knowledge can lead an automatic approach to differ of a manual approach. Nevertheless, our approach is a step ahead of contract violations correction aided by automation.

In the last study, we compared the categorization performed by voluntary JML experts with the automatic categorization. The results showed that our heuristics-based approach is a little bit similar to results from manual analysis of JML experts; the mean of *matches* was 0.33.

The results are promising for applying specific test techniques in the context of contract-based programs, fostering a more widespread adoption of such methodology by lowering the costs of conformance checking. Furthermore, we believe that a heuristics-based approach can be useful to suggest a categorization for nonconformances in contract-based programs.

6.2 Related Work

Our work is related to three types of research results: researches that investigate the conformance checking between programs and their formal contracts; researches about some kind of categorization for nonconformances, aiming help the programmer in the process of nonconformances correction; and researches concerning to automatic tests generation.

6.2.1 Conformance Checking

Several efforts on verified software [54; 18; 16; 26; 49; 20] have been carried out in the context of source code specification with contract-based languages [47; 43; 3], and the Design by Contract (DBC) methodology [48]. Dynamic checking of contracts, despite its incompleteness, gives immediate feedback for programmers, even if they write partial contracts. Detecting nonconformances is, in this case, strictly dependent on the quality of the test cases that exercise the runtime assertions produced out of contracts. For DBC, a related approach proposes auto tests [49], where contracts are used as oracles to expected outputs, an the test generation is performed automatically. The AutoTest tool is an implementation of conformance checking to the Eiffel language [47]. This tool is similar to our approach: both aim at conformance checking, and use randomly-guided tests generation (ARTOO [20] for AutoTest and Randoop [55] for JMLOK 2.0). However, AutoTest supports mixing manual and automated test, while our approach focuses on complete automation. Our approach is directed to JML, which is relatively simple to apply to existent Java programs; in addition, our approach is also concerned with automatic categorization of nonconformances.

Concerning Spec# language, Boogie [2] is the Spec# static program verifier. This tool generates logical verification conditions from a Spec# program. Internally, Boogie uses an automatic theorem prover that analyzes the verification conditions to prove the correctness of the program or find errors in it. This tool is similar to our approach: both aim at conformance checking; but differ in approaches used to do this: dynamic conformance checking – in JMLOK 2.0 – and static checking – in Boogie.

There are a number of tools that apply dynamic checking for detecting nonconformances between JML programs. JMLUnit [18] is a semi-automatic tool to check conformance, generating test case skeletons by combining calls to the methods under test, lacking test. On the other hand, JMLOK 2.0 is completely automatic and provides an automatic categorization for nonconformances.

In order to handle some JMLUnit limitations, JMLUnitNG [74], an improved version of JMLUnit, automatically generates test data for non-primitive types; however, it does not exempt users from providing their test data in some situations. Whereas, JMLOK 2.0 is completely automatic and provides an automatic categorization for nonconformances.

Korat [11] has the advantage over JMLUnit of being able to construct the objects which invoke the method under test. However, test cases constructed by Korat only consist of one object construction and one method invocation on this object; furthermore Korat requires the implementation of an imperative predicate to specify the desired structural constraints, and a bound to the desired test input size. Our approach, on the other hand, does not require implementation of functions and generates more than one call for the methods under test; furthermore, we present an automatic categorization for nonconformances.

Jartege [54] is a semi-automatic tool, inspired by JMLUnit, for generating test cases, by a random approach with assigned weights to classes and methods under test; however, the user might have to assign weights for methods under test and information about how to choose the weights is not provided. Whilst, JMLOK 2.0 is completely automatic and provides an automatic categorization for nonconformances.

JET [16] aims at applying dynamic testing to conformance checking in JML, by randomly generating test cases using contracts as test oracles. Genetic algorithms are applied for automatically building all test data that exercise runtime assertions. This choice is promisingly effective, although it raises the risk of nondeterminism in generating test cases and data on successive executions of the tool. Regarding purpose, the JET tool is closely related to JMLOK 2.0, because to the best of our knowledge is the only tool for JML that does not require user inputs (as test data or implementation of functions). However, JMLOK 2.0 presents an automatic categorization for nonconformances, not provided by JET.

On the other hand, ESC/Java2 [23] performs static verification in JML programs, applying a logical-based technique that verifies statically if no violation of JML contracts will happen at runtime. Nevertheless ESC/Java2 is neither sound nor complete, this tool presents a high rate of false positives. Whereas, JMLOK 2.0 is sound, because all nonconformances found are correct, but it is not complete, because it is not ensured that it found all nonconformances.

As a new point of view about contracts, the Option Contracts [28] idea arises as an extension to DBC, introducing notions about transfer and exercise do give more freedom to developers in the use of contracts. This methodology has the same purpose of our work: the use of contracts since development phase to improve the systems developed and to reduce the cost of faults correction.

Table 6.1 summarizes those approaches with respect to: (1) the kind of conformance checking performed; (2) whether there are some categorization of the nonconformances; (3) whether the approach is automatic; (4) the specification language used in the approach.

Tabela 6.1: Related Work about conformance checking.

	Conformance Checking	Categorization	Automation level	Specification Language
AutoTest	dynamic	–	automatic	Eiffel
Boogie	static	–	automatic	Spec#
JMLUnit	dynamic	–	semi automatic	JML
JMLUnitNG	dynamic	–	semi automatic	JML
Korat	dynamic	–	semi automatic	JML
Jartege	dynamic	–	semi automatic	JML
JET	dynamic	–	automatic	JML
ESC/Java2	static	–	automatic	JML
JMLOK 2.0	dynamic	automatic	automatic	JML

6.2.2 Categorization of Contract Violations

Regarding categorization, Rosenblum [63] presents an early study about the main assertions that reveal contract violations into C programs and a classification system for those assertions. He used App – Annotation PreProcessor for C programs, like the *jmlc* compiler. His work presents two levels at which a problem (a contract violation) may happen: *Specification of Function Interfaces*, and *Specification of Function Bodies*, the first one is related with our work – considers the external behavior of methods, their pre- and postconditions, and invariants. To *Specification of Function Interfaces* level, the author presents eight main kinds of assertion violations: Consistency Between Arguments (I1), Dependency of Return Value on Arguments (I2), Effect on Global State (I3), Context in Which Function is Called (I4), Frame Specifications (I5), Subrange Membership of Data (I6), Enumeration Membership of Data (I7), and Non-Null Pointers (I8). Those kinds of assertions are related to our types: I1, I3, and I7 are related to precondition type; I2, and I6 are related to postcondition type; and I4, I5, and I8 are related to invariant.

More recently, Polikarpova et al. [56] present three categories to classify nonconformances: *specification faults*, *inconsistency faults* and *real faults*. In this work we present a three-level model to classify nonconformances composed by a category, a type, and a likely cause for each nonconformance – the latter is a distinctive feature of our model.

Table 6.2 summarizes those approaches with respect to: (1) the categorization scope – whether the approach categorizes contract violations from external and internal behaviors of the system under test; (2) whether the approach is automatic; (3) the specification language for the categorization proposed.

Tabela 6.2: Related Work about categorization approaches for contract violations.

	Categorization scope	**Automation level**	**Language**
Rosenblum	external and internal behaviors	manual	C
Polikarpova et al.	external and internal behaviors	manual	Eiffel
JMLOK 2.0	only external behavior	automatic	JML

6.2.3 Automatic Test Generation

Software testing, although cannot guarantee that the software is error free, is a widely-used approach to get some confidence about the software behavior. In the scenario of contract-based programs, tests generated automatically is commonly used to check conformance between programs and their contracts, as verification by formal proofs is hard to scale and static analysis is limited.

In this context, test cases with automatically-generated data are important due to their low cost and high precision in detecting conformance problems that needs more than one modification into the object under test. In our work we use a random-directed tests generation approach, by means of Randoop [55]. In Randoop, the feedback from execution of sequence being constructed is used as pruning function – only valid constructions are considered in the next sequence generations. This approach is similar to Adaptive Random Testing (ART) [15] approach. In ART, the test cases generation is based on the idea of tests more distant are more probable to detect problems than test separated by smaller distances; ART uses the Euclidean distance to calculate the distance between test cases. An extension of ART ideas are presented in ARTOO [20], the adaptive random testing for object-oriented programs; in ARTOO there is a modification of distance calculation to consider properties related to object-oriented systems.

Other approach similar to the used in our work is presented on JET [16], where genetic algorithms are used in test generation process. Genetic algorithms are based on feedback to creation of new generations, similarly that occurs in the feedback-directed approach of Randoop. An another approach to automatic test generation is EvoSuite [31]. This tool uses an evolutionary search approach that evolves whole test suites with respect to an entire coverage criterion at the same time. EvoSuite approach, similar to ours, uses a guided approach to tests generation, in that case a search based approach.

Table 6.3 summarizes those approaches with respect to test generation approach.

Tabela 6.3: Related Work about automatic test generation.

	Test generation approach
ART	Based on distance between test cases
ARTOO	Based on distance – considering properties from object-oriented programs
JET	Based on the feedback – from genetic algorithms
EvoSuite	Based on evolutionary search
Randoop	Based on feedback from execution of sequence being constructed

6.3 Future Work

As future work, we intend to improve the test generation of JMLOK 2.0, considering other techniques to test generation (e.g. advanced Monte Carlo techniques - like n-factor, approaches to symbolic/concolic execution - like in Symbolic Pathfinder [58] or KLEE-like tools [14], incremental SAT-solving - like in FAJITA [1], and evolutionary algorithms - like in EvoSuite [31]), and perform new experimental studies to evaluate those approaches. Furthermore, we plan to investigate again the feasibility of the OpenJML compiler [22] use, once it is the new JML compiler and *jmlc* [17] is a deprecated project. We also expect to extend our approach to consider problems in the client side [59]. Additionally, we aim investigating other metrics for contracts, in order to further analyze the relationship between nonconformances and program properties; and to perform an evaluation of contract-based programs without nonconformances and to inject systematically nonconformances to evaluate more precisely the quality of our approach: JMLOK 2.0.

Concerning to conformance testing, we intend to investigate a formal definition of conformance for each type of conformance problem that we are considering.

Regarding categorization of nonconformances, we intend to extend our categorization model to treat with nonconformances in specifications inside of method bodies (nonconformances from *IntraconditionalErrors*, such as *assertion*, *loop variant*, *loop invariant*). Moreover, we plan to investigate the possibility of return a list of likely causes for a nonconformance, based in the correspondences found in the contract-based program. Furthermore, to the best of our knowledge, there are no automatic categorization approaches for Eiffel neither Spec#; so, we plan to examine the feasibility of extend our categorization approach for these languages.

We have identified that there are no automated tools that consider refactoring [30] in the context of conformance checking; and the preservation of behavior become a property hard to verify [45]. There are formal approaches, such as presented by Freitas [32], but these approaches have a high cost; so we intend to develop an automatic approach to consider the conformance problem in the context of refactoring, contributing to use of Design by Contract methodology and to construction of reliable programs.

Bibliografia

[1] P. Abad, N. Aguirre, V. Bengolea, D. Ciolek, M. F. Frias, J. Galeotti, T. Maibaum, M. Moscato, N. Rosner, and I. Vissani. Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving. In *Proceedings of 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 21–30. IEEE, 2013.

[2] M. Barnett, B. Chang, R. DeLine, B. Jacobs, and R. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, volume 4111, pages 364–387. Springer Berlin Heidelberg, 2006.

[3] M. Barnett, M. Fähndrich, R. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# Experience. *Communications of the ACM*, 54(6):81–91, 2011.

[4] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J. Lanet, M. Pavlova, and A. Requet. JACK – A Tool for Validation of Security and Behaviour of Java Applications. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, pages 152–174. Springer-Verlag, 2007.

[5] V. R. Basili, R. W. Selby, and D. H. Hutchens. Experimentation in Software Engineeering. *IEEE Transactions on Software Engineering*, 12(7):733 – 743, 1986.

[6] B. Beizer. *Black-box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., 1995.

[7] J. Berg and B. Jacobs. The LOOP Compiler for Java and JML. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–312. Springer-Verlag, 2001.

[8] R. V. Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[9] D. Bjørner and C.B. Jones. *The Vienna Development Method: The Meta-Language*. Springer-Verlag, 1978.

[10] V. A. Bloomfield. *Using R for Numerical Analysis in Science and Engineering*. Chapman & Hall/CRC, 2014.

[11] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 123–133. ACM, 2002.

[12] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996.

[14] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.

[15] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive Random Testing. In *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, volume 3321, pages 320–329. Springer Berlin Heidelberg, 2005.

[16] Y. Cheon. Automated Random Testing to Detect Specification-Code Inconsistencies. Technical report, In Proceedings of The 2007 International Conference on Software Engineering Theory and Practice, 2007.

[17] Y. Cheon and G. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In *Proceedings of The International Conference on Software Engineering Research and Practice*, pages 322–328. CSREA Press, 2002.

[18] Y. Cheon and G. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 231–255. Springer-Verlag, 2002.

[19] Y. Cheon and C. Medrano. Random Test Data Generation for Java Classes Annotated with JML Specifications. In *International Conference on Software Engineering Research and Practice*, pages 385–392, 2007.

[20] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: Adaptive Random Testing for Object-oriented Software. In *Proceedings of the 30th International Conference on Software Engineering*, pages 71–80. ACM, 2008.

[21] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[22] D. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In *Proceedings of the 3rd International Conference on NASA Formal methods*, pages 472–479. Springer-Verlag, 2011.

[23] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML – Progress and issues in building and using ESC/Java2. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 108–128. Springer-Verlag, 2004.

[24] IEEE Computer Society. Standards Coordinating Committee, Institute of Electrical, Electronics Engineers, and IEEE Standards Board. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std. The Institute, 1990.

[25] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press Ltd., 1972.

[26] A. Darvas and P. Müller. Faithful mapping of model classes to mathematical structures. *IET Software*, pages 477–499, 2008.

[27] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1997.

[28] C. Dimoulas, R. B. Findler, and M. Felleisen. Option Contracts. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 475–494. ACM, 2013.

[29] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[30] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[31] G. Fraser and A. Arcuri. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 416–419. ACM, 2011.

[32] G. R. F. Freitas. Refactoring Annotated Java Programs: A Rule-Based Approach. Master's thesis, Universidade de Pernambuco, 2009.

[33] P. F. Gibbins. What Are Formal Methods? *Information and Software Technology*, 30(3):131–137, 1988.

[34] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag New York, Inc., 1993.

[35] J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch Family of Specification Languages. *IEEE Software*, 2(5):24–36, 1985.

[36] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, 1990.

[37] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2):9:1–9:76, 2009.

[38] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[39] P. C. Jorgensen. *Software Testing: A Craftsman's Approach*. Auerbach Publications, 2013.

[40] G. K Kanji. *100 Statistical Tests*. Sage, 2006.

[41] M. E. Khan. Different Approaches to White Box Testing Technique for Finding Errors. *International Journal of Software Engineering and Its Applications*, 5(3):1–13, 2011.

[42] G. Leavens. Larch/C++ Reference Manual. Version 5.1. Technical report, Iowa State University, 1995.

[43] G. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[44] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. JML Reference Manual, 2013.

[45] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[46] B. Meyer. Design by Contract. In *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice Hall, 1991.

[47] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992.

[48] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[49] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs That Test Themselves. *IEEE Computer*, 42(9):46–55, 2009.

[50] A. Milanez, T. Massoni, and R. Gheyi. Categorizing Nonconformances Between Programs and Their Specifications. In *Proceedings of the 7th Brazilian Workshop on Systematic and Automated Software Testing*, 2013.

[51] A. F. Milanez. Case Study on Categorizing Nonconformances. Technical report, Software Practices Laboratory, Federal University of Campina Grande, 2014.

[52] A. Müller. VDM - The Vienna Development Method. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, 2009.

[53] C. V. S. Oliveira. Uma Abordagem para Verificar Não-conformidades em Programas Especificados com Contratos. Master's thesis, Federal University of Campina Grande, 2013.

[54] C. Oriat. Jartege: A Tool for Random Generation of Unit Tests for Java Classes. In *Proceedings of the First International Conference on Quality of Software Architectures and Software Quality, and Proceedings of the Second International Conference on Software Quality*, pages 242–256. Springer-Verlag Berlin Heidelberg, 2005.

[55] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *Proceeedings of the 29th International Conference on Software Engineering*, pages 75–84. IEEE Computer Society, 2007.

[56] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer. What Good Are Strong Specifications? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 262–271. IEEE Press, 2013.

[57] E. Poll, P. Hartel, and E. Jong. A Java Reference Model of Transacted Memory for Smart Cards. In *In Smart Card Research and Advanced Application Conference*, pages 75–86. USENIX Association, 2002.

[58] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 179–180. ACM, 2010.

[59] H. Rebêlo, G. Leavens, and R. Lima. Client-aware Checking and Information Hiding in Interface Specifications with JML/Ajmlc. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, pages 11–12. ACM, 2013.

[60] H. Rebêlo, R. Lima, M. L. Cornélio, G. Leavens, A. C. Mota, and C. Oliveira. Optimizing JML Features Compilation in ajmlc Using Aspect-Oriented Refactorings. In

Proceedings of the 13th Brazilian Symposium on Programming Languages, pages 117–130, 2009.

[61] H. Rebêlo, S. Soares, R. Lima, L. Ferreira, and M. Cornélio. Implementing Java Modeling Language Contracts with AspectJ. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 228–233. ACM, 2008.

[62] R. M. S. Rodrigues. JML-Based Formal Development of a Java Card Application for Managing Medical Appointments. Master's thesis, Universidade da Madeira, 2009.

[63] D. S. Rosenblum. Towards a Method of Programming with Assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104. ACM, 1992.

[64] A. Sarcar and Y. Cheon. A New Eclipse-Based JML Compiler Built Using AST Merging. *2010 Second World Congress on Software Engineering*, 2:287–292, 2010.

[65] P. H. Schmitt and I. Tonin. Verifying the Mondex Case Study. In *Proceedings of Fifth IEEE International Conference on Software Engineering and Formal Methods*, pages 47–58, 2007.

[66] S. S. Shapiro and M. B. Wilk. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 3/4(52):591–611, 1965.

[67] G. Soares, R. Gheyi, E. R. Murphy-Hill, and B. Johnson. Comparing Approaches to Analyze Refactoring Activity on Software Repositories. *Journal of Systems and Software*, 86(4):1006–1022, 2013.

[68] I. Sommerville. *Software Engineering*. Pearson, 2010.

[69] M. Staats, M. W. Whalen, and M. P. E. Heimdahl. Programs, Tests, and Oracles: The Foundations of Testing Revisited. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 391–400. ACM, 2011.

[70] J. Tretmans. Testing Concurrent Systems: A Formal Approach. In *Proceedings of the 10th International Conference on Concurrency Theory*, pages 46–65. Springer-Verlag, 1999.

[71] C. Varjão, R. Gheyi, T. Massoni, and G. Soares. JMLOK: Uma Ferramenta para Verificar Conformidade em Programas Java/JML. In *Proceedings of the 2nd Brazilian Conference on Software: Theory and Practice (Tools session)*, 2011.

[72] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof.* Prentice-Hall, Inc., 1996.

[73] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):19:1–19:36, 2009.

[74] D. Zimmerman and R. Nagmoti. JMLUnit: The Next Generation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*, pages 183–197. Springer-Verlag, 2010.

Apêndice A

Manual Categorization Results on Contract-based Programs

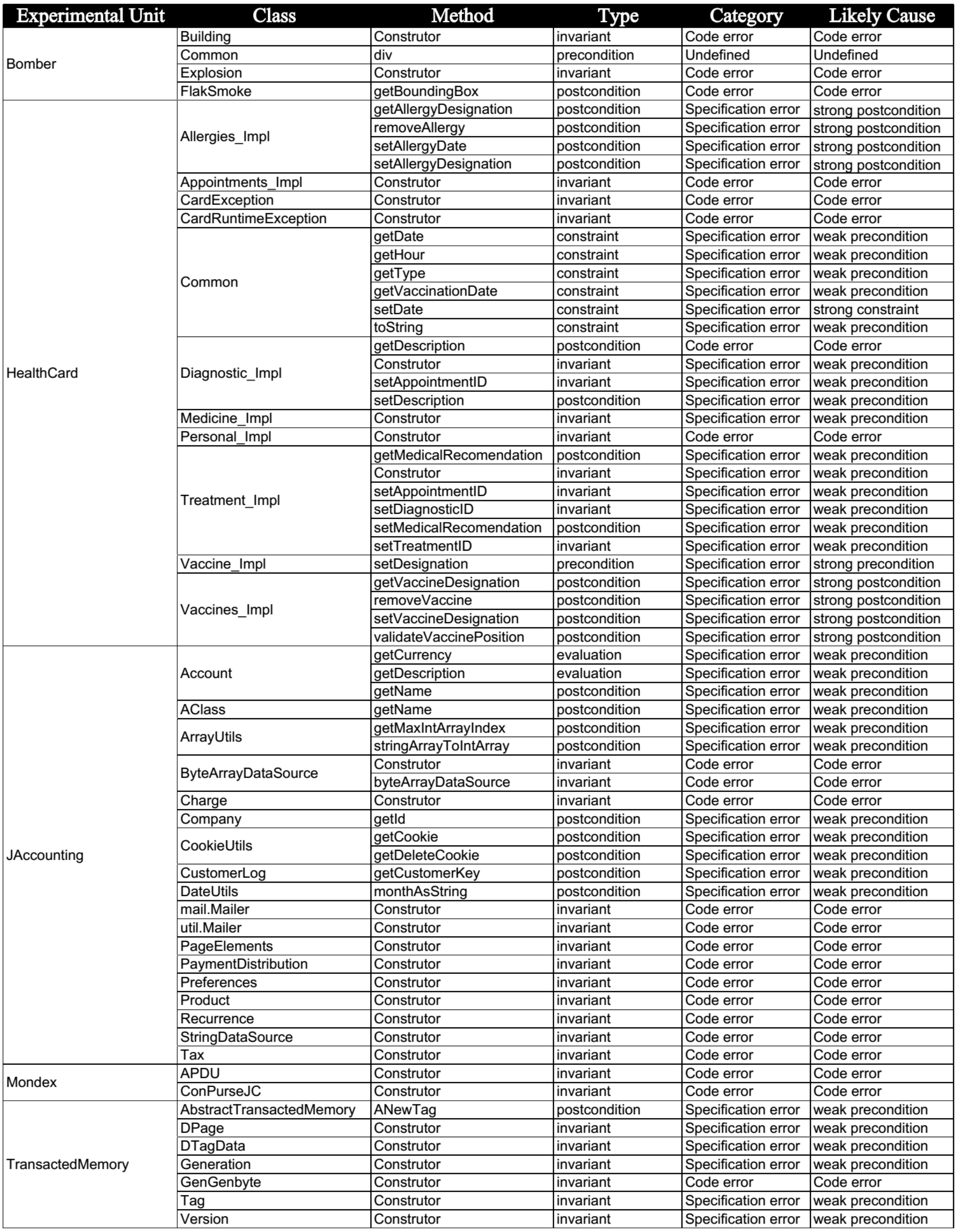
Table A.1 shows the results of the manual categorization of nonconformances on sample contract-based programs.

Tabela A.1: Results of the manual categorization process on Sample Programs. Column Experimental Unit shows the name of the experimental unit. Columns Class and Method display the names of the class and the method, respectively, where a nonconformance was detected. Column Category presents the category manually assigned to the discovered nonconformance. Column Type exhibits the nonconformance's type. Finally, column Likely Cause reveals the likely cause manually assigned.

Experimental Unit	Class	Method	Type	Category	Likely Cause
Samples	BoundedStack.BoundedStack	Construtor	postcondition	Specification error	weak precondition
			invariant	Specification error	weak precondition
	stacks.BoundedStack	Construtor	postcondition	Specification error	weak precondition
			invariant	Specification error	weak precondition
	dbc.Polar	angle	postcondition	Specification error	strong postcondition
		imaginaryPart	postcondition	Specification error	strong postcondition
		magnitude	postcondition	Specification error	strong postcondition
		realPart	postcondition	Specification error	strong postcondition
	dbc.Rectangular	angle	postcondition	Specification error	strong postcondition
		imaginaryPart	postcondition	Specification error	strong postcondition
		magnitude	postcondition	Specification error	strong postcondition
		realPart	postcondition	Specification error	strong postcondition
	misc.SingleSolution	limit	postcondition	Specification error	weak precondition
	list.Link	getEntry	evaluation	Specification error	weak precondition
	list.list2.E_OneWayList	equals	postcondition	Undefined	Undefined
	list.list3.E_OneWayList	equals	postcondition	Undefined	Undefined
	list.list3.TwoWayIterator	isDone	postcondition	Undefined	Undefined
		next	postcondition	Undefined	Undefined

Table A.2 shows the results of the manual categorization of nonconformances on open-source contract-based programs.

Tabela A.2: Results of the manual categorization process on Open-source Programs. Column Experimental Unit shows the name of the experimental unit. Columns Class and Method display the names of the class and the method, respectively, where a nonconformance was detected. Column Category presents the category manually assigned to the discovered nonconformance. Column Type exhibits the nonconformance's type. Finally, column Likely Cause reveals the likely cause manually assigned.



Experimental Unit	Class	Method	Type	Category	Likely Cause
Bomber	Building	Construtor	invariant	Code error	Code error
	Common	div	precondition	Undefined	Undefined
	Explosion	Construtor	invariant	Code error	Code error
	FlakSmoke	getBoundingBox	postcondition	Code error	Code error
HealthCard	Allergies_Impl	getAllergyDesignation	postcondition	Specification error	strong postcondition
		removeAllergy	postcondition	Specification error	strong postcondition
		setAllergyDate	postcondition	Specification error	strong postcondition
		setAllergyDesignation	postcondition	Specification error	strong postcondition
	Appointments_Impl	Construtor	invariant	Code error	Code error
	CardException	Construtor	invariant	Code error	Code error
	CardRuntimeException	Construtor	invariant	Code error	Code error
	Common	getDate	constraint	Specification error	weak precondition
		getHour	constraint	Specification error	weak precondition
		getType	constraint	Specification error	weak precondition
		getVaccinationDate	constraint	Specification error	weak precondition
		setDate	constraint	Specification error	strong constraint
		toString	constraint	Specification error	weak precondition
	Diagnostic_Impl	getDescription	postcondition	Code error	Code error
		Construtor	invariant	Specification error	weak precondition
		setAppointmentID	invariant	Specification error	weak precondition
		setDescription	postcondition	Specification error	weak precondition
	Medicine_Impl	Construtor	invariant	Specification error	weak precondition
	Personal_Impl	Construtor	invariant	Code error	Code error
	Treatment_Impl	getMedicalRecomendation	postcondition	Specification error	weak precondition
		Construtor	invariant	Specification error	weak precondition
		setAppointmentID	invariant	Specification error	weak precondition
		setDiagnosticID	invariant	Specification error	weak precondition
		setMedicalRecomendation	postcondition	Specification error	weak precondition
		setTreatmentID	invariant	Specification error	weak precondition
	Vaccine_Impl	setDesignation	precondition	Specification error	strong precondition
	Vaccines_Impl	getVaccineDesignation	postcondition	Specification error	strong postcondition
		removeVaccine	postcondition	Specification error	strong postcondition
		setVaccineDesignation	postcondition	Specification error	strong postcondition
		validateVaccinePosition	postcondition	Specification error	strong postcondition
JAccounting	Account	getCurrency	evaluation	Specification error	weak precondition
		getDescription	evaluation	Specification error	weak precondition
		getName	postcondition	Specification error	weak precondition
	AClass	getName	postcondition	Specification error	weak precondition
	ArrayUtils	getMaxIntArrayIndex	postcondition	Specification error	weak precondition
		stringArrayToIntArray	postcondition	Specification error	weak precondition
	ByteArrayDataSource	Construtor	invariant	Code error	Code error
		byteArrayDataSource	invariant	Code error	Code error
	Charge	Construtor	invariant	Code error	Code error
	Company	getId	postcondition	Specification error	weak precondition
	CookieUtils	getCookie	postcondition	Specification error	weak precondition
		getDeleteCookie	postcondition	Specification error	weak precondition
	CustomerLog	getCustomerKey	postcondition	Specification error	weak precondition
	DateUtils	monthAsString	postcondition	Specification error	weak precondition
	mail.Mailer	Construtor	invariant	Code error	Code error
	util.Mailer	Construtor	invariant	Code error	Code error
	PageElements	Construtor	invariant	Code error	Code error
	PaymentDistribution	Construtor	invariant	Code error	Code error
	Preferences	Construtor	invariant	Code error	Code error
	Product	Construtor	invariant	Code error	Code error
	Recurrence	Construtor	invariant	Code error	Code error
	StringDataSource	Construtor	invariant	Code error	Code error
	Tax	Construtor	invariant	Code error	Code error
Mondex	APDU	Construtor	invariant	Code error	Code error
	ConPurseJC	Construtor	invariant	Code error	Code error
TransactedMemory	AbstractTransactedMemory	ANewTag	postcondition	Specification error	weak precondition
	DPage	Construtor	invariant	Specification error	weak precondition
	DTagData	Construtor	invariant	Specification error	weak precondition
	Generation	Construtor	invariant	Specification error	weak precondition
	GenGenbyte	Construtor	invariant	Code error	Code error
	Tag	Construtor	invariant	Specification error	weak precondition
	Version	Construtor	invariant	Specification error	weak precondition

Apêndice B

Automatic Categorization Results on Contract-based Programs

Table B.1 shows the results of the automatic categorization of nonconformances on sample contract-based programs.

Tabela B.1: Results of the automatic categorization process on Sample Programs. Column Experimental Unit shows the name of the experimental unit. Columns Class and Method display the names of the class and the method, respectively, where a nonconformance was detected. Column Category presents the category manually assigned to the discovered nonconformance. Column Type exhibits the nonconformance's type. Finally, column Likely Cause reveals the likely cause automatically assigned.

Experimental Unit	Class	Method	Type	Category	Likely Cause
Samples	BoundedStack.BoundedStack	Construtor	postcondition	Specification error	Weak precondition
			invariant	Specification error	Weak precondition
	stacks.BoundedStack	Construtor	postcondition	Specification error	Weak precondition
			invariant	Specification error	Weak precondition
	dbc.Polar	angle	postcondition	Specification error	weak precondition
		imaginaryPart	postcondition	Specification error	weak precondition
		magnitude	postcondition	Specification error	weak precondition
		realPart	postcondition	Specification error	weak precondition
	dbc.Rectangular	angle	postcondition	Specification error	weak precondition
		imaginaryPart	postcondition	Specification error	weak precondition
		magnitude	postcondition	Specification error	weak precondition
		realPart	postcondition	Specification error	weak precondition
	misc.SingleSolution	limit	postcondition	Specification error	weak precondition
	list.Link	getEntry	evaluation	Specification error	weak precondition
	list.list2.E_OneWayList	equals	postcondition	Specification error	weak precondition
	list.list3.E_OneWayList	equals	postcondition	Specification error	weak precondition
	list.list3.TwoWayIterator	isDone	postcondition	Specification error	weak precondition
		next	postcondition	Specification error	weak precondition

Table B.2 shows the results of the automatic categorization of nonconformances on open-source contract-based programs.

Tabela B.2: Results of the automatic categorization process on Open-source Programs. Column Experimental Unit shows the name of the experimental unit. Columns Class and Method display the names of the class and the method, respectively, where a nonconformance was detected. Column Category presents the category manually assigned to the discovered nonconformance. Column Type exhibits the nonconformance's type. Finally, column Likely Cause reveals the likely cause automatically assigned.

Experimental Unit	Class	Method	Type	Category	Likely Cause
Bomber	Building	Construtor	invariant	Code error	Code error
	Common	div	precondition	Specification error	Strong precondition
	Explosion	Construtor	invariant	Code error	Code error
	FlakSmoke	getBoundingBox	postcondition	Code error	weak precondition
HealthCard	Allergies_Impl	getAllergyDesignation	postcondition	Specification error	weak precondition
		removeAllergy	postcondition	Specification error	weak precondition
		setAllergyDate	postcondition	Specification error	weak precondition
		setAllergyDesignation	postcondition	Specification error	weak precondition
	Appointments_Impl	Construtor	invariant	Code error	weak precondition
	CardException	Construtor	invariant	Code error	Code error
	CardRuntimeException	Construtor	invariant	Code error	Code error
	Common	getDate	constraint	Specification error	weak precondition
		getHour	constraint	Specification error	weak precondition
		getType	constraint	Specification error	weak precondition
		getVaccinationDate	constraint	Specification error	weak precondition
		setDate	constraint	Specification error	weak precondition
		toString	constraint	Specification error	weak precondition
	Diagnostic_Impl	getDescription	postcondition	Code error	weak precondition
		Construtor	invariant	Specification error	weak precondition
		setAppointmentID	invariant	Specification error	weak precondition
		setDescription	postcondition	Specification error	weak precondition
	Medicine_Impl	Construtor	invariant	Specification error	weak precondition
	Personal_Impl	Construtor	invariant	Code error	Code error
	Treatment_Impl	getMedicalRecomendation	postcondition	Specification error	weak precondition
		Construtor	invariant	Specification error	weak precondition
		setAppointmentID	invariant	Specification error	weak precondition
		setDiagnosticID	invariant	Specification error	weak precondition
		setMedicalRecomendation	postcondition	Specification error	Weak precondition
		setTreatmentID	invariant	Specification error	weak precondition
	Vaccine_Impl	setDesignation	precondition	Specification error	strong precondition
	Vaccines_Impl	getVaccineDesignation	postcondition	Specification error	weak precondition
		removeVaccine	postcondition	Specification error	weak precondition
		setVaccineDesignation	postcondition	Specification error	weak precondition
		validateVaccinePosition	postcondition	Specification error	weak precondition
JAccounting	Account	getCurrency	evaluation	Specification error	weak precondition
		getDescription	evaluation	Specification error	weak precondition
		getName	postcondition	Specification error	weak precondition
	AClass	getName	postcondition	Specification error	weak precondition
	ArrayUtils	getMaxIntArrayIndex	postcondition	Specification error	weak precondition
		stringArrayToIntArray	postcondition	Specification error	weak precondition
	ByteArrayDataSource	Construtor	invariant	Code error	Code error
		byteArrayDataSource	invariant	Code error	Code error
	Charge	Construtor	invariant	Code error	Code error
	Company	getId	postcondition	Specification error	weak precondition
	CookieUtils	getCookie	postcondition	Specification error	weak precondition
		getDeleteCookie	postcondition	Specification error	weak precondition
	CustomerLog	getCustomerKey	postcondition	Specification error	weak precondition
	DateUtils	monthAsString	postcondition	Specification error	weak precondition
	mail.Mailer	Construtor	invariant	Code error	Code error
	util.Mailer	Construtor	invariant	Code error	Code error
	PageElements	Construtor	invariant	Code error	Code error
	PaymentDistribution	Construtor	invariant	Code error	Code error
	Preferences	Construtor	invariant	Code error	Code error
	Product	Construtor	invariant	Code error	Code error
	Recurrence	Construtor	invariant	Code error	Code error
	StringDataSource	Construtor	invariant	Code error	Code error
	Tax	Construtor	invariant	Code error	Code error
Mondex	APDU	Construtor	invariant	Code error	Code error
	ConPurseJC	Construtor	invariant	Code error	Code error
TransactedMemory	AbstractTransactedMemory	ANewTag	postcondition	Specification error	weak precondition
	DPage	Construtor	invariant	Specification error	weak precondition
	DTagData	Construtor	invariant	Specification error	weak precondition
	Generation	Construtor	invariant	Specification error	weak precondition
	GenGenbyte	Construtor	invariant	Code error	Code error
	Tag	Construtor	invariant	Specification error	weak precondition
	Version	Construtor	invariant	Specification error	weak precondition

Apêndice C

Numbered Nonconformances

Table C.1 lists all detected nonconformances numbered for sample contract-based programs. The detected nonconformances were numbered in the following manner: starting with `BoundedStack` package, then `stacks`, `dbc`, `misc` and finally `list`. In each package, the nonconformances were ordered alphabetically by name of methods where the nonconformances were detected.

Tabela C.1: Numbered nonconformances for Sample Programs.

Conformance Number	Experimental Unit	Class	Method	Type
1	Samples	BoundedStack.BoundedStack	Construtor	postcondition
2				invariant
3		stacks.BoundedStack	Construtor	postcondition
4				invariant
5		dbc.Polar	angle	postcondition
6			imaginaryPart	postcondition
7			magnitude	postcondition
8			realPart	postcondition
9		dbc.Rectangular	angle	postcondition
10			imaginaryPart	postcondition
11			magnitude	postcondition
12			realPart	postcondition
13		misc.SingleSolution	limit	postcondition
14		list.Link	getEntry	evaluation
15		list.list2.E_OneWayList	equals	postcondition
16		list.list3.E_OneWayList	equals	postcondition
17		list.list3.TwoWayIterator	isDone	postcondition
18			next	postcondition

Table C.2 lists all detected nonconformances numbered for open-source contract-based programs. We continue the numbering starting with `Bomber`, then `HealthCard`, `JAccounting`, `Mondex` and finally `TransactedMemory`. In each experimental unit, the nonconformances were ordered alphabetically by the name of the methods where the nonconformances were detected.

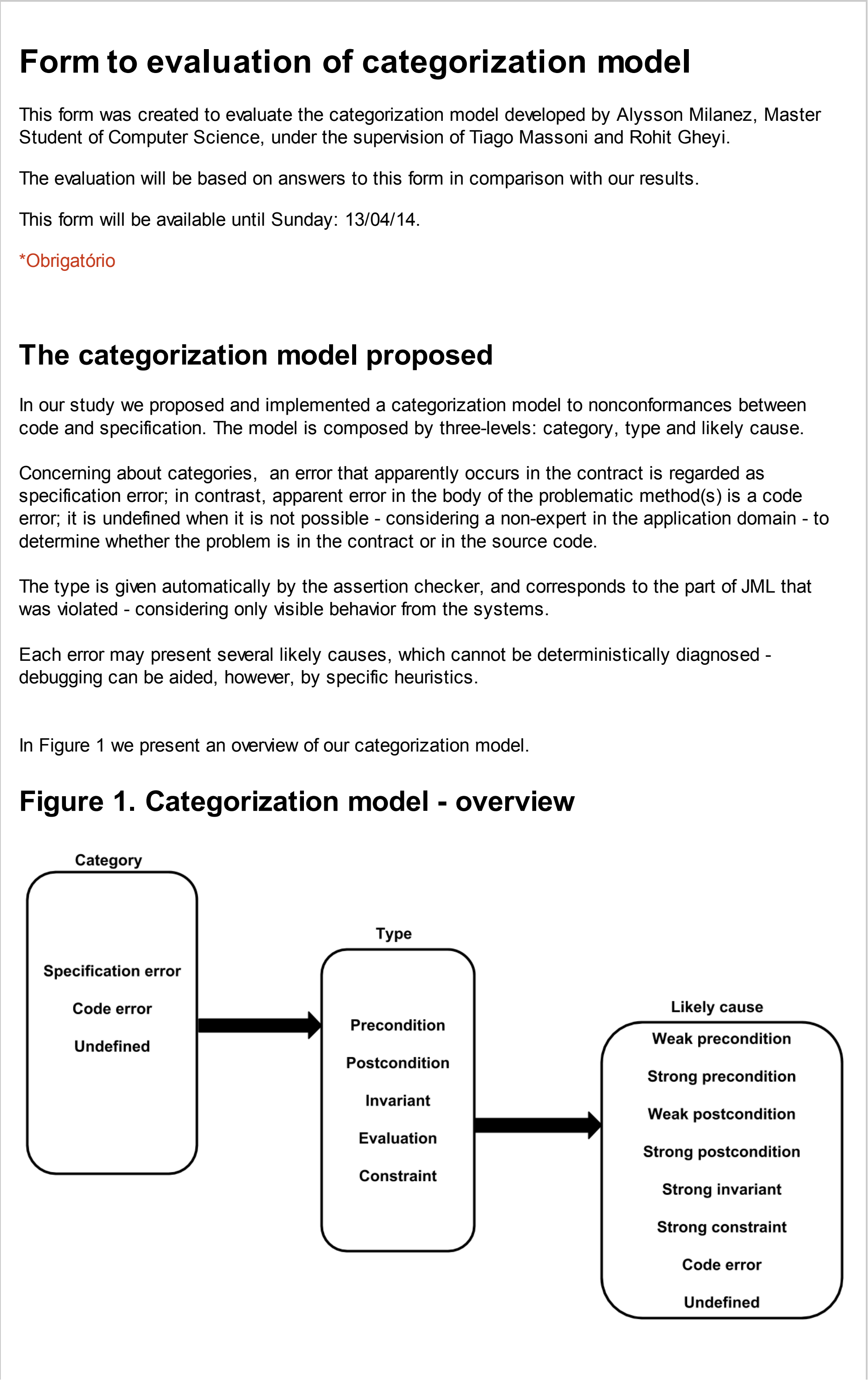
Tabela C.2: Numbered nonconformances for Open-source Programs.

Conformance Number	Experimental Unit	Class	Method	Type
19	Bomber	Building	Construtor	invariant
20		Common	div	precondition
21		Explosion	Construtor	invariant
22		FlakSmoke	getBoundingBox	postcondition
23	HealthCard	Allergies_Impl	getAllergyDesignation	postcondition
24			removeAllergy	postcondition
25			setAllergyDate	postcondition
26			setAllergyDesignation	postcondition
27		Appointments_Impl	Construtor	invariant
28		CardException	Construtor	invariant
29		CardRuntimeException	Construtor	invariant
30		Common	getDate	constraint
31			getHour	constraint
32			getType	constraint
33			getVaccinationDate	constraint
34			setDate	constraint
35			toString	constraint
36		Diagnostic_Impl	getDescription	postcondition
37			Construtor	invariant
38			setAppointmentID	invariant
39			setDescription	postcondition
40		Medicine_Impl	Construtor	invariant
41		Personal_Impl	Construtor	invariant
42		Treatment_Impl	getMedicalRecomendation	postcondition
43			Construtor	invariant
44			setAppointmentID	invariant
45			setDiagnosticID	invariant
46			setMedicalRecomendation	postcondition
47			setTreatmentID	invariant
48		Vaccine_Impl	setDesignation	precondition
49		Vaccines_Impl	getVaccineDesignation	postcondition
50			removeVaccine	postcondition
51			setVaccineDesignation	postcondition
52			validateVaccinePosition	postcondition
53	JAccounting	Account	getCurrency	evaluation
54			getDescription	evaluation
55			getName	postcondition
56		AClass	getName	postcondition
57		ArrayUtils	getMaxIntArrayIndex	postcondition
58			stringArrayToIntArray	postcondition
59		mail.ByteArrayDataSource	byteArrayDataSource	invariant
60		util.ByteArrayDataSource	Construtor	invariant
61		Charge	Construtor	invariant
62		Company	getId	postcondition
63		CookieUtils	getCookie	postcondition
64			getDeleteCookie	postcondition
65		CustomerLog	getCustomerKey	postcondition
66		DateUtils	monthAsString	postcondition
67		mail.Mailer	Construtor	invariant
68		util.Mailer	Construtor	invariant
69		PageElements	Construtor	invariant
70		PaymentDistribution	Construtor	invariant
71		Preferences	Construtor	invariant
72		Product	Construtor	invariant
73		Recurrence	Construtor	invariant
74		StringDataSource	Construtor	invariant
75		Tax	Construtor	invariant
76	Mondex	APDU	Construtor	invariant
77		ConPurseJC	Construtor	invariant
78	TransactedMemory	AbstractTransactedMemory	ANewTag	postcondition
79		DPage	Construtor	invariant
80		DTagData	Construtor	invariant
81		Generation	Construtor	invariant
82		GenGenbyte	Construtor	invariant
83		Tag	Construtor	invariant
84		Version	Construtor	invariant

Apêndice D

Form to Evaluation of the Categorization Model

Below we present the form that we used in the evaluation of our categorization model by JML experts. In this form are available the nonconformances that were selected, the link to the contract-based program, a test case that reveals the nonconformance, and some information about the nonconformance location.



Form to evaluation of categorization model

This form was created to evaluate the categorization model developed by Alysson Milanez, Master Student of Computer Science, under the supervision of Tiago Massoni and Rohit Gheyi.

The evaluation will be based on answers to this form in comparison with our results.

This form will be available until Sunday: 13/04/14.

*Obrigatório

The categorization model proposed

In our study we proposed and implemented a categorization model to nonconformances between code and specification. The model is composed by three-levels: category, type and likely cause.

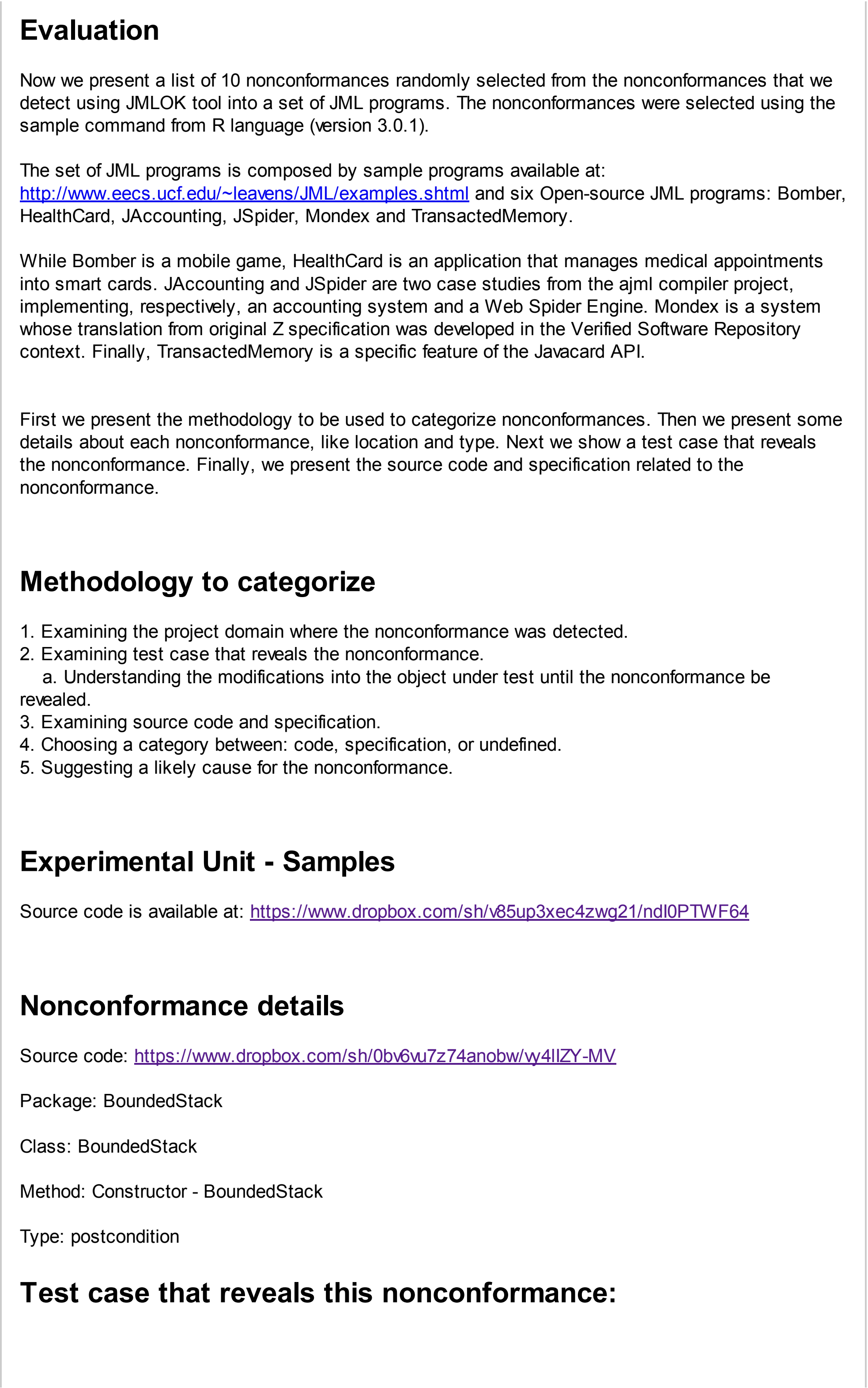
Concerning about categories, an error that apparently occurs in the contract is regarded as specification error; in contrast, apparent error in the body of the problematic method(s) is a code error; it is undefined when it is not possible - considering a non-expert in the application domain - to determine whether the problem is in the contract or in the source code.

The type is given automatically by the assertion checker, and corresponds to the part of JML that was violated - considering only visible behavior from the systems.

Each error may present several likely causes, which cannot be deterministically diagnosed - debugging can be aided, however, by specific heuristics.

In Figure 1 we present an overview of our categorization model.

Figure 1. Categorization model - overview



Evaluation

Now we present a list of 10 nonconformances randomly selected from the nonconformances that we detect using JMLOK tool into a set of JML programs. The nonconformances were selected using the sample command from R language (version 3.0.1).

The set of JML programs is composed by sample programs available at: http://www.eecs.ucf.edu/~leavens/JML/examples.shtml and six Open-source JML programs: Bomber, HealthCard, JAccounting, JSpider, Mondex and TransactedMemory.

While Bomber is a mobile game, HealthCard is an application that manages medical appointments into smart cards. JAccounting and JSpider are two case studies from the ajml compiler project, implementing, respectively, an accounting system and a Web Spider Engine. Mondex is a system whose translation from original Z specification was developed in the Verified Software Repository context. Finally, TransactedMemory is a specific feature of the Javacard API.

First we present the methodology to be used to categorize nonconformances. Then we present some details about each nonconformance, like location and type. Next we show a test case that reveals the nonconformance. Finally, we present the source code and specification related to the nonconformance.

Methodology to categorize

1. Examining the project domain where the nonconformance was detected.
2. Examining test case that reveals the nonconformance.
    a. Understanding the modifications into the object under test until the nonconformance be revealed.
3. Examining source code and specification.
4. Choosing a category between: code, specification, or undefined.
5. Suggesting a likely cause for the nonconformance.

Experimental Unit - Samples

Source code is available at: https://www.dropbox.com/sh/v85up3xec4zwg21/ndI0PTWF64

Nonconformance details

Source code: https://www.dropbox.com/sh/0bv6vu7z74anobw/vy4IIZY-MV

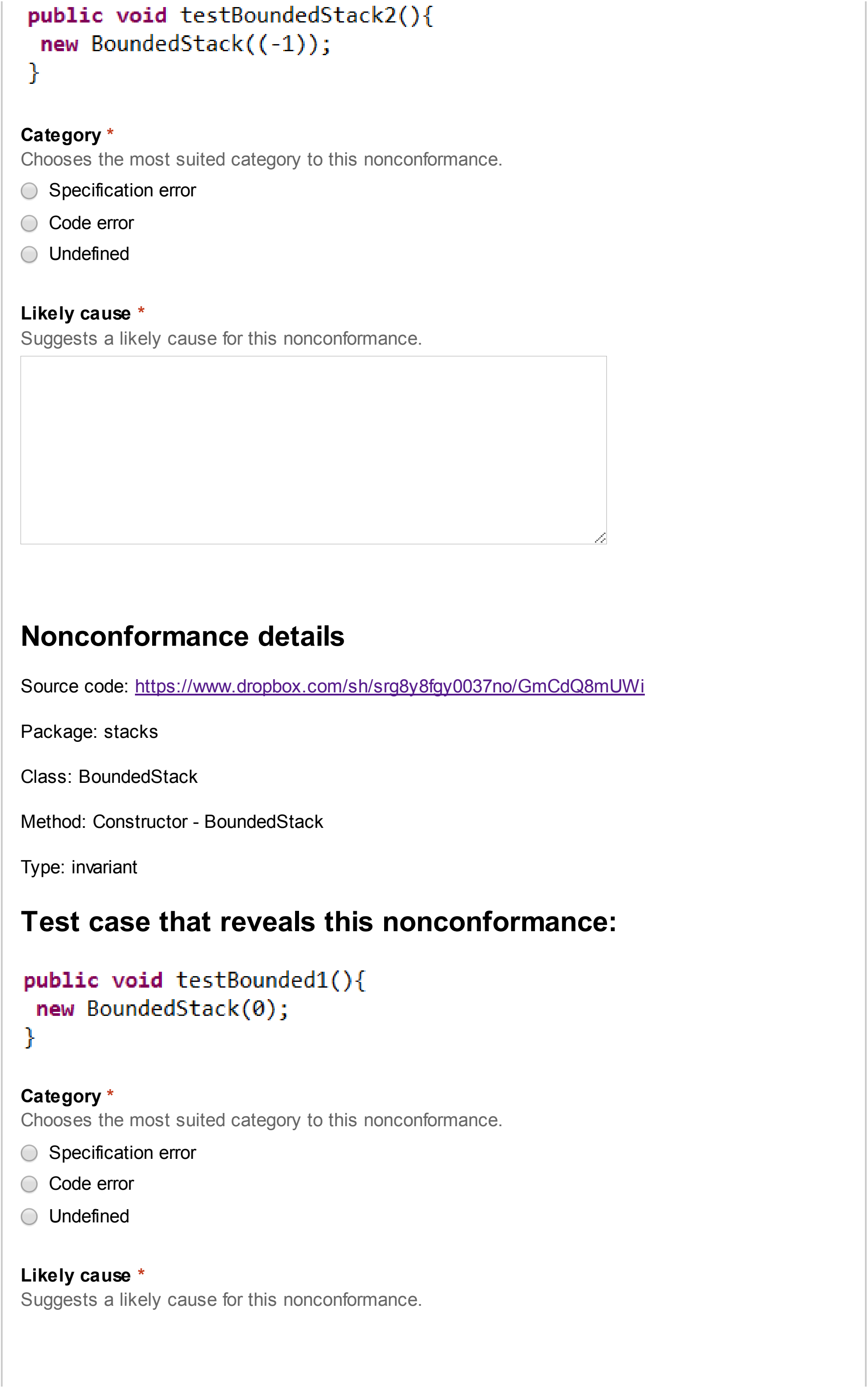
Package: BoundedStack

Class: BoundedStack

Method: Constructor - BoundedStack

Type: postcondition

Test case that reveals this nonconformance:



```
public void testBoundedStack2(){
 new BoundedStack((-1));
}
```

Category *

Chooses the most suited category to this nonconformance.

○ Specification error

○ Code error

○ Undefined

Likely cause *

Suggests a likely cause for this nonconformance.

Nonconformance details

Source code: https://www.dropbox.com/sh/srg8y8fgy0037no/GmCdQ8mUWi

Package: stacks

Class: BoundedStack

Method: Constructor - BoundedStack

Type: invariant

Test case that reveals this nonconformance:

```
public void testBounded1(){
 new BoundedStack(0);
}
```

Category *

Chooses the most suited category to this nonconformance.

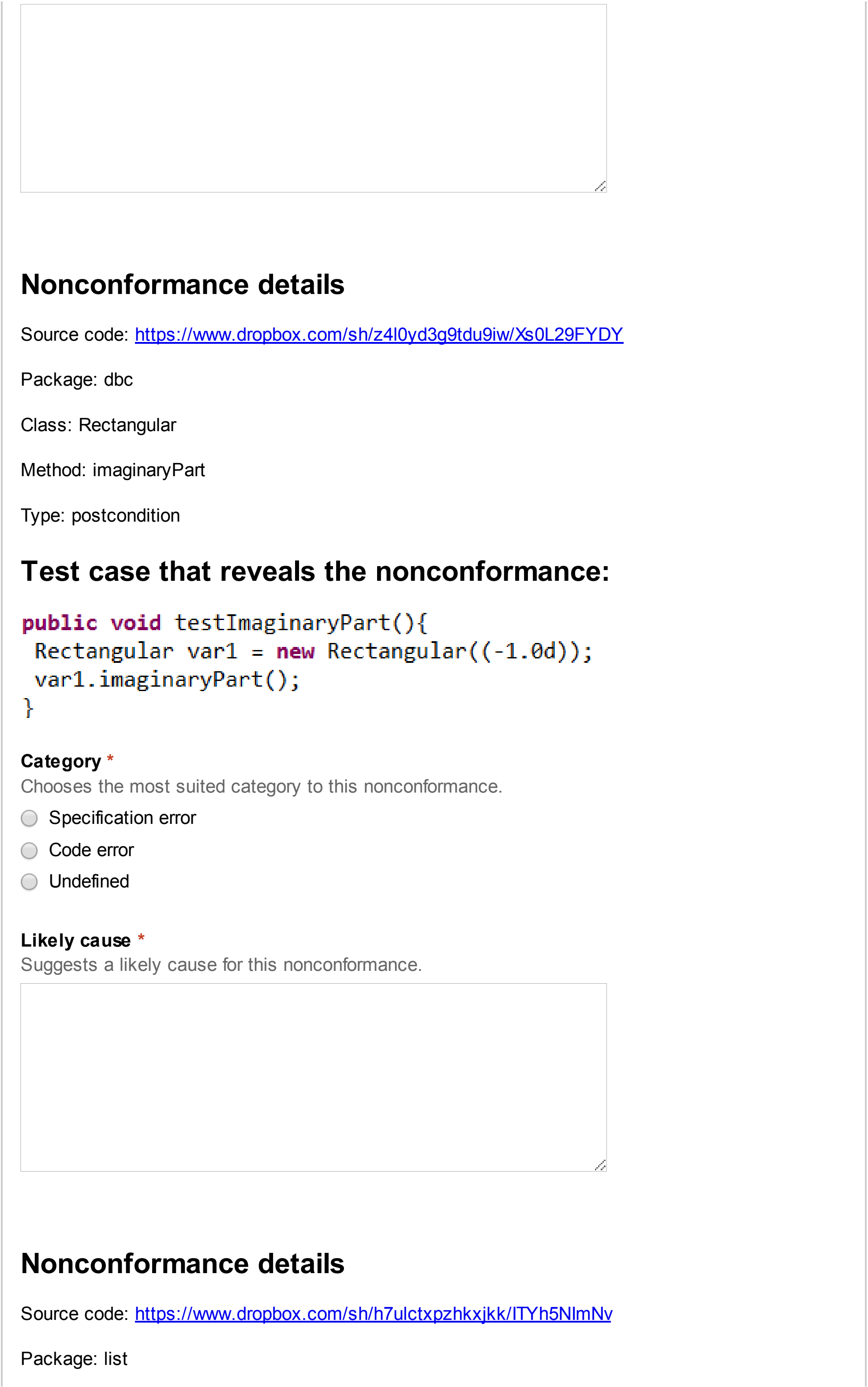
○ Specification error

○ Code error

○ Undefined

Likely cause *

Suggests a likely cause for this nonconformance.



Nonconformance details

Source code: https://www.dropbox.com/sh/z4l0yd3g9tdu9iw/Xs0L29FYDY

Package: dbc

Class: Rectangular

Method: imaginaryPart

Type: postcondition

Test case that reveals the nonconformance:

```
public void testImaginaryPart(){
 Rectangular var1 = new Rectangular((-1.0d));
 var1.imaginaryPart();
}
```

Category *

Chooses the most suited category to this nonconformance.

○ Specification error

○ Code error

○ Undefined

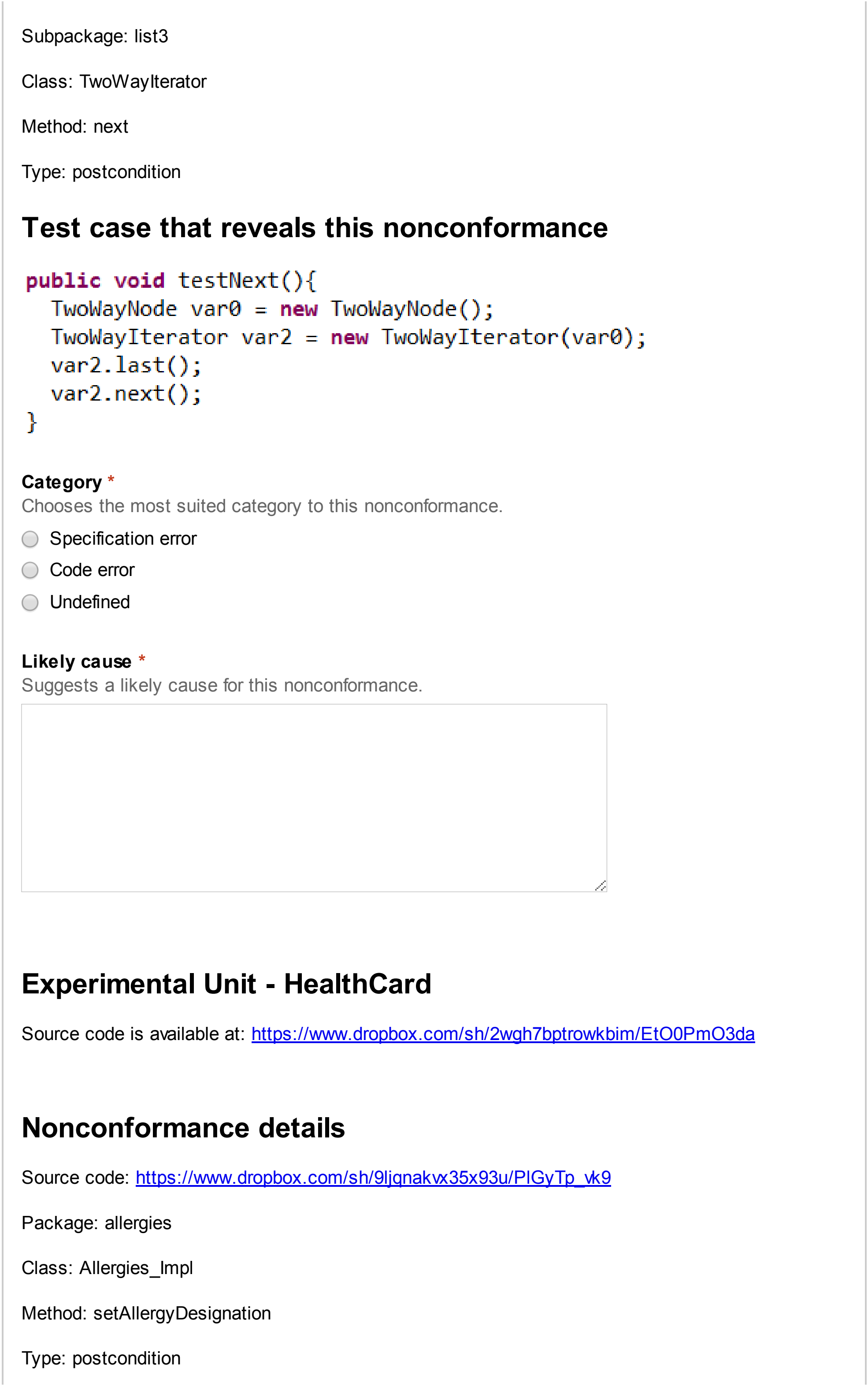
Likely cause *

Suggests a likely cause for this nonconformance.

Nonconformance details

Source code: https://www.dropbox.com/sh/h7ulctxpzhkxjkk/ITYh5NlmNv

Package: list



Subpackage: list3

Class: TwoWayIterator

Method: next

Type: postcondition

Test case that reveals this nonconformance

```
public void testNext(){
  TwoWayNode var0 = new TwoWayNode();
  TwoWayIterator var2 = new TwoWayIterator(var0);
  var2.last();
  var2.next();
}
```

Category *

Chooses the most suited category to this nonconformance.

○ Specification error

○ Code error

○ Undefined

Likely cause *

Suggests a likely cause for this nonconformance.

Experimental Unit - HealthCard

Source code is available at: https://www.dropbox.com/sh/2wgh7bptrowkbim/EtO0PmO3da

Nonconformance details

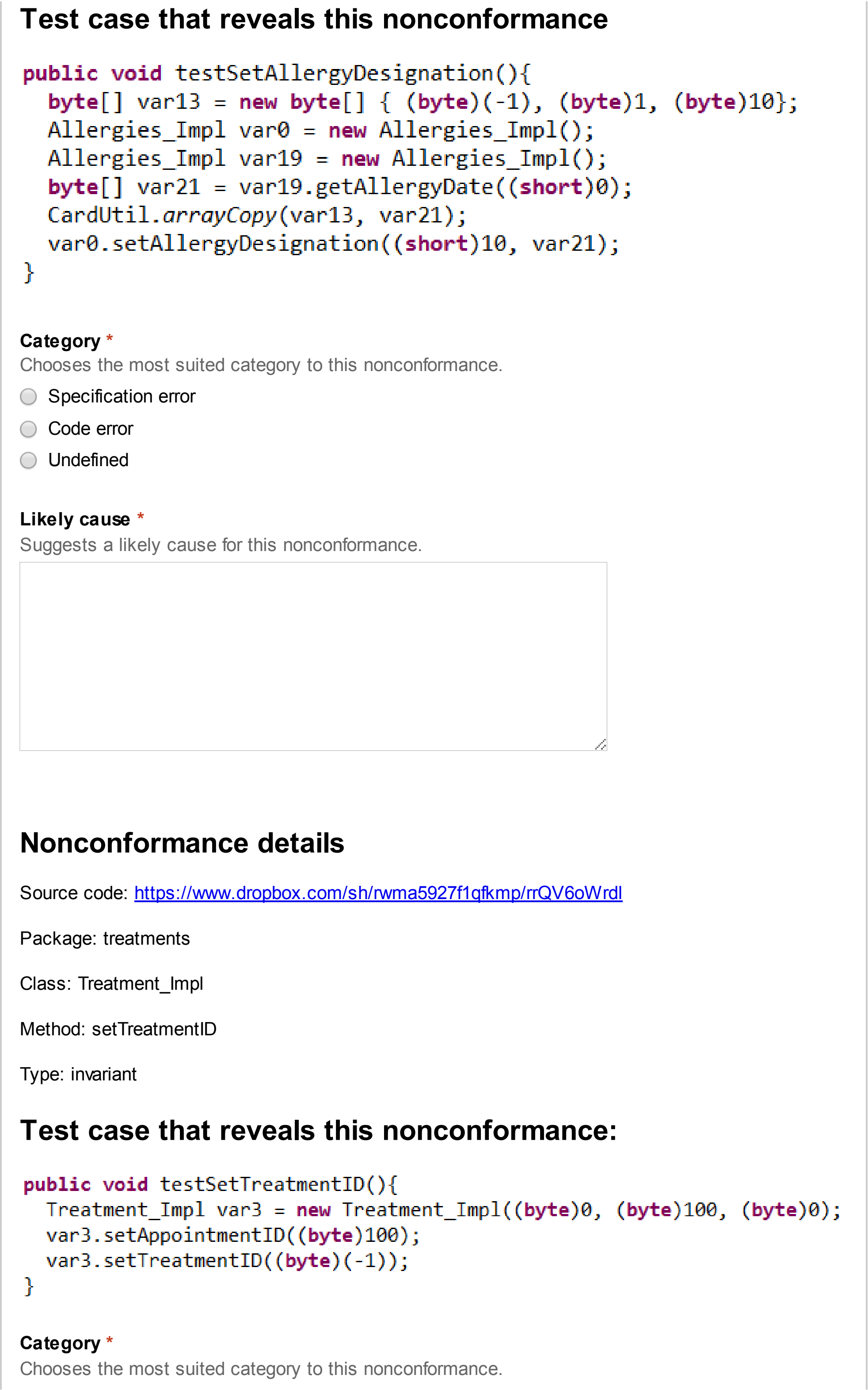
Source code: https://www.dropbox.com/sh/9ljqnakvx35x93u/PIGyTp_vk9

Package: allergies

Class: Allergies_Impl

Method: setAllergyDesignation

Type: postcondition



Test case that reveals this nonconformance

```
public void testSetAllergyDesignation(){
  byte[] var13 = new byte[] { (byte)(-1), (byte)1, (byte)10};
  Allergies_Impl var0 = new Allergies_Impl();
  Allergies_Impl var19 = new Allergies_Impl();
  byte[] var21 = var19.getAllergyDate((short)0);
  CardUtil.arrayCopy(var13, var21);
  var0.setAllergyDesignation((short)10, var21);
}
```

Category *

Chooses the most suited category to this nonconformance.

○ Specification error

○ Code error

○ Undefined

Likely cause *

Suggests a likely cause for this nonconformance.

Nonconformance details

Source code: https://www.dropbox.com/sh/rwma5927f1qfkmp/rrQV6oWrdl

Package: treatments

Class: Treatment_Impl

Method: setTreatmentID

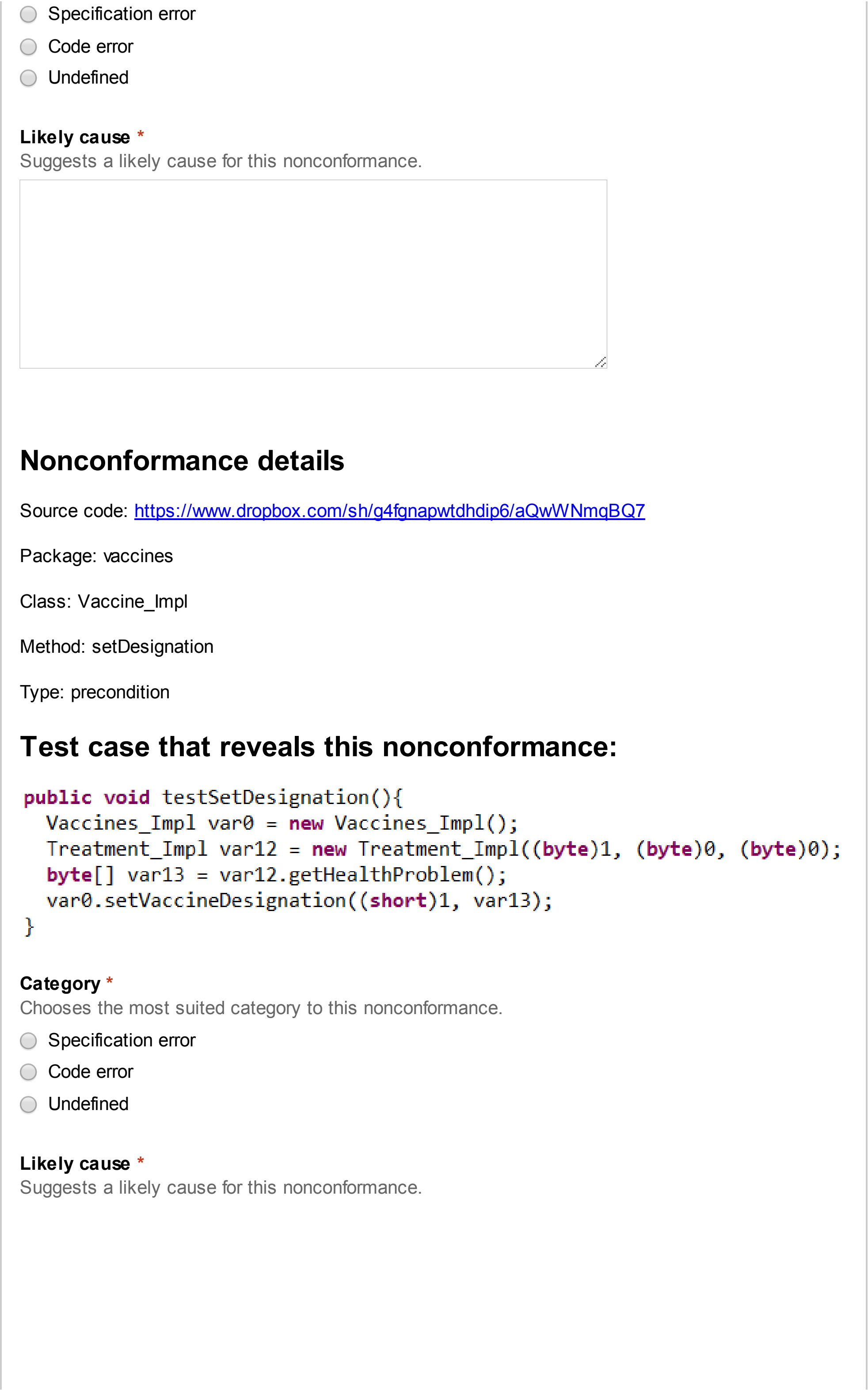
Type: invariant

Test case that reveals this nonconformance:

```
public void testSetTreatmentID(){
  Treatment_Impl var3 = new Treatment_Impl((byte)0, (byte)100, (byte)0);
  var3.setAppointmentID((byte)100);
  var3.setTreatmentID((byte)(-1));
}
```

Category *

Chooses the most suited category to this nonconformance.



○ Specification error

○ Code error

○ Undefined

Likely cause *

Suggests a likely cause for this nonconformance.

Nonconformance details

Source code: https://www.dropbox.com/sh/g4fgnapwtdhdip6/aQwWNmqBQ7

Package: vaccines

Class: Vaccine_Impl

Method: setDesignation

Type: precondition

Test case that reveals this nonconformance:

```
public void testSetDesignation(){
  Vaccines_Impl var0 = new Vaccines_Impl();
  Treatment_Impl var12 = new Treatment_Impl((byte)1, (byte)0, (byte)0);
  byte[] var13 = var12.getHealthProblem();
  var0.setVaccineDesignation((short)1, var13);
}
```

Category *

Chooses the most suited category to this nonconformance.

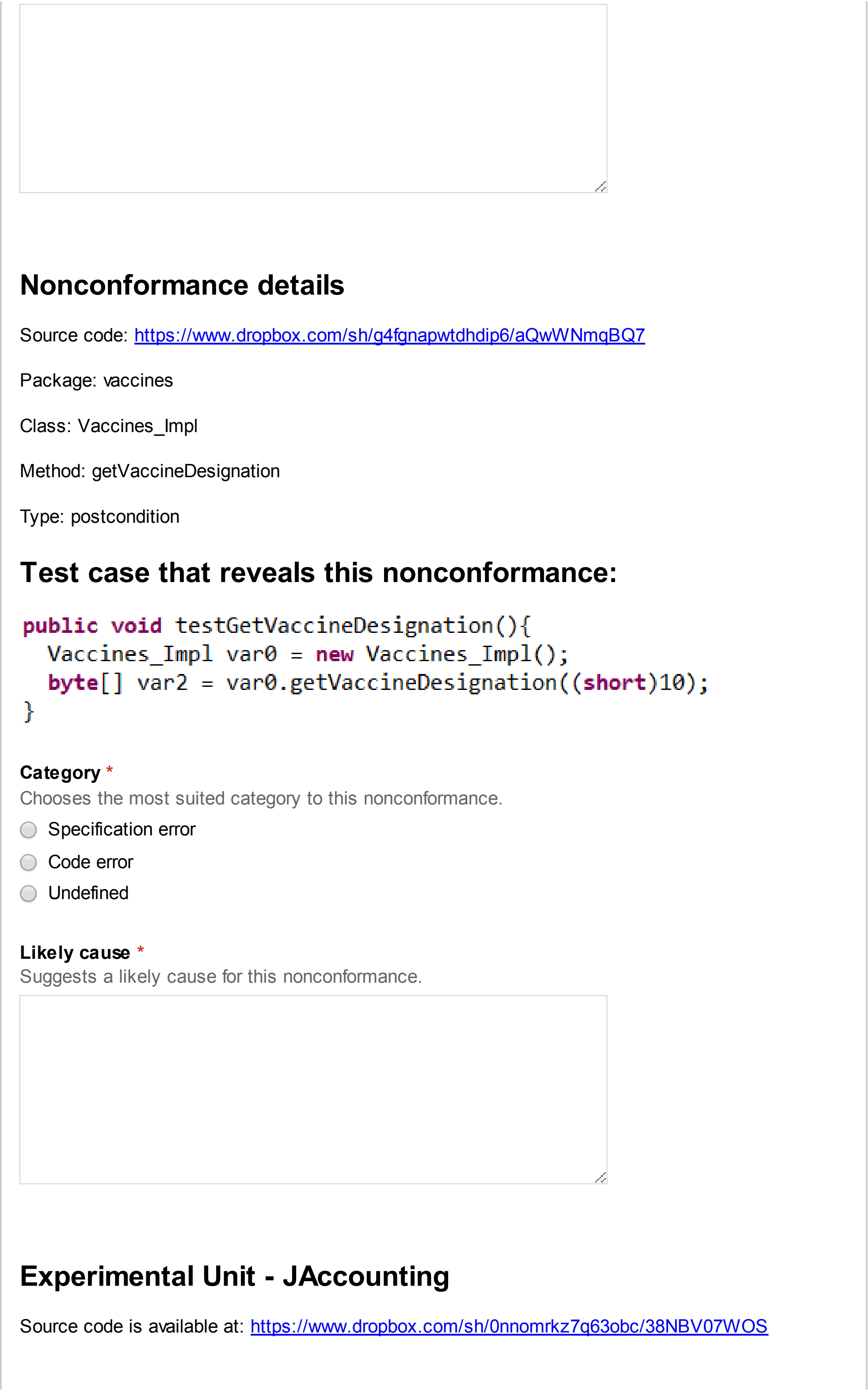
○ Specification error

○ Code error

○ Undefined

Likely cause *

Suggests a likely cause for this nonconformance.



Nonconformance details

Source code: https://www.dropbox.com/sh/g4fgnapwtdhdip6/aQwWNmqBQ7

Package: vaccines

Class: Vaccines_Impl

Method: getVaccineDesignation

Type: postcondition

Test case that reveals this nonconformance:

```
public void testGetVaccineDesignation(){
  Vaccines_Impl var0 = new Vaccines_Impl();
  byte[] var2 = var0.getVaccineDesignation((short)10);
}
```

Category *

Chooses the most suited category to this nonconformance.

○ Specification error

○ Code error

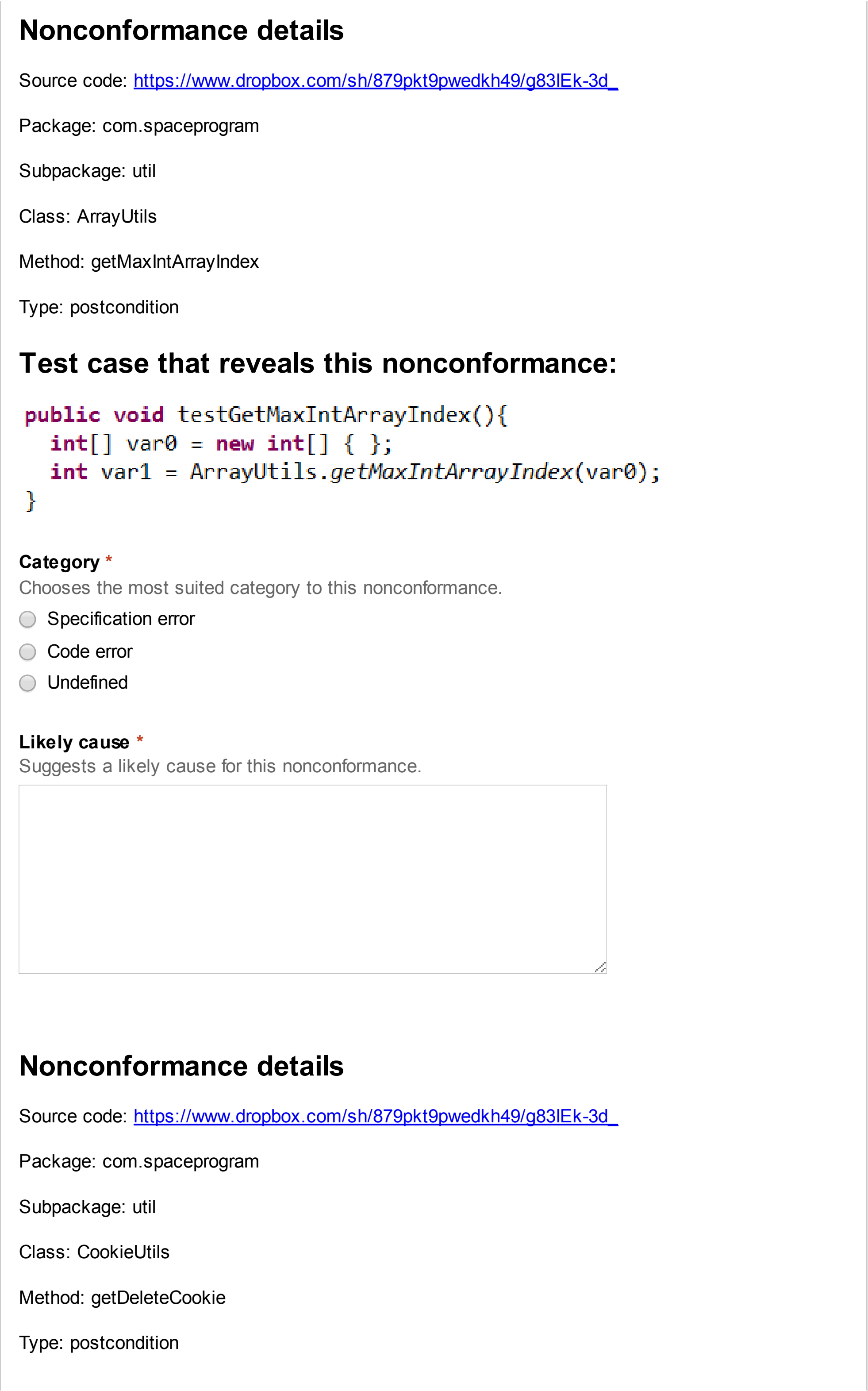
○ Undefined

Likely cause *

Suggests a likely cause for this nonconformance.

Experimental Unit - JAccounting

Source code is available at: https://www.dropbox.com/sh/0nnomrkz7q63obc/38NBV07WOS



Nonconformance details

Source code: https://www.dropbox.com/sh/879pkt9pwedkh49/g83IEk-3d_

Package: com.spaceprogram

Subpackage: util

Class: ArrayUtils

Method: getMaxIntArrayIndex

Type: postcondition

Test case that reveals this nonconformance:

```
public void testGetMaxIntArrayIndex(){
  int[] var0 = new int[] { };
  int var1 = ArrayUtils.getMaxIntArrayIndex(var0);
}
```

Category *

Chooses the most suited category to this nonconformance.

○ Specification error

○ Code error

○ Undefined

Likely cause *

Suggests a likely cause for this nonconformance.

Nonconformance details

Source code: https://www.dropbox.com/sh/879pkt9pwedkh49/g83IEk-3d_

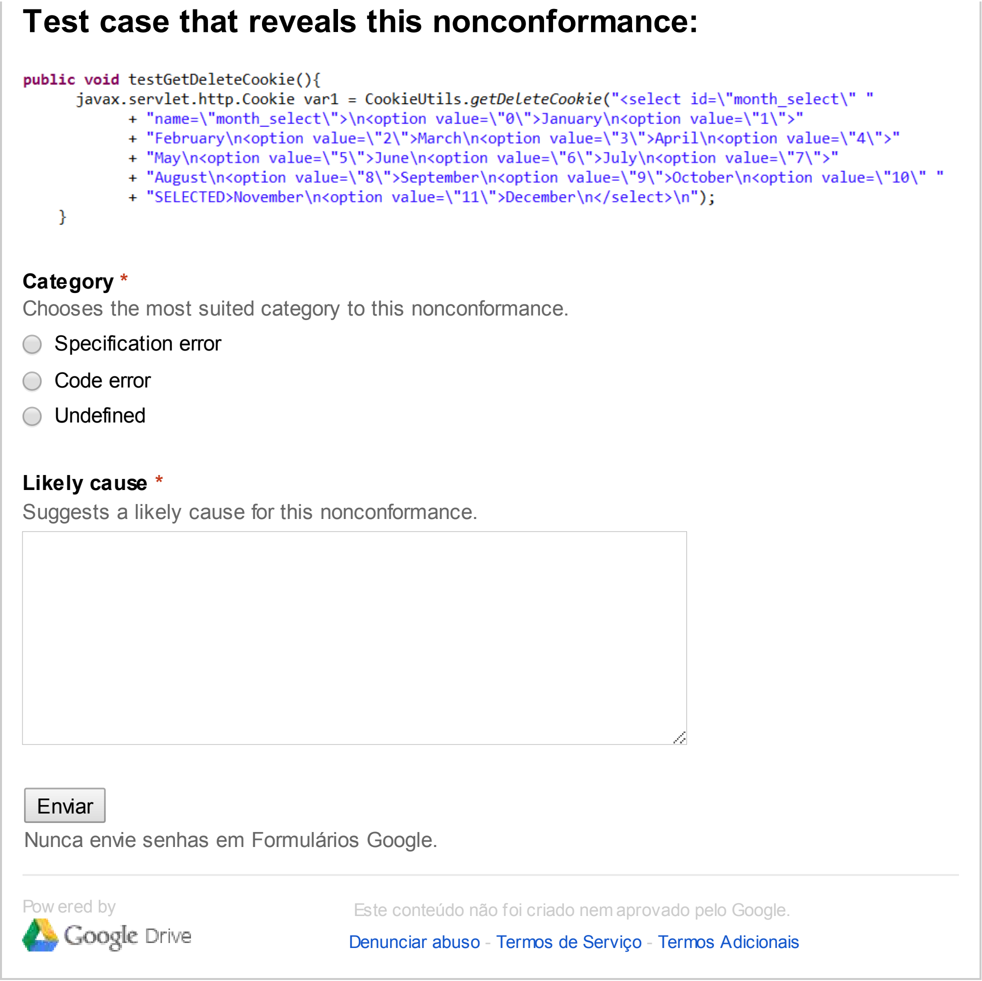
Package: com.spaceprogram

Subpackage: util

Class: CookieUtils

Method: getDeleteCookie

Type: postcondition



Test case that reveals this nonconformance:

```
public void testGetDeleteCookie(){
      javax.servlet.http.Cookie var1 = CookieUtils.getDeleteCookie("<select id=\"month_select\" "
            + "name=\"month_select\">\n<option value=\"0\">January\n<option value=\"1\">"
            + "February\n<option value=\"2\">March\n<option value=\"3\">April\n<option value=\"4\">"
            + "May\n<option value=\"5\">June\n<option value=\"6\">July\n<option value=\"7\">"
            + "August\n<option value=\"8\">September\n<option value=\"9\">October\n<option value=\"10\" "
            + "SELECTED>November\n<option value=\"11\">December\n</select>\n");
    }
```

Category *

Chooses the most suited category to this nonconformance.

○ Specification error

○ Code error

○ Undefined

Likely cause *

Suggests a likely cause for this nonconformance.

Nunca envie senhas em Formulários Google.

Powered by
Google Drive
Este conteúdo não foi criado nem aprovado pelo Google.
Denunciar abuso - Termos de Serviço - Termos Adicionais