Federal University of Campina Grande

Electrical Engineering and Informatics Center

Graduate Program in Computer Science

Fostering Design By Contract by Exploiting the Relationship between Code Commentary and Contracts

Alysson Filgueira Milanez

Thesis submitted to Coordination of Graduate Program in Computer Science of the Federal University of Campina Grande - Campus I in partial fulfillment of the requirements for the degree of Ph.D. in Computer Science.

Research Area: Computer Science

Research Line: Computing's Methodology and Techniques

Tiago Massoni and Rohit Gheyi

(Advisors)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

M637f Milanez, Alysson Filgueira.
Fostering design by contract by exploiting the relationship between code commentary and contracts / Alysson Filgueira Milanez. – Campina Grande, 2018.
129 f.: il. color.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2018.
"Orientação: Prof. Dr. Tiago Lima Massoni, Prof. Dr. Rohit Gheyi".
Referências.

1. Programação por Contratos. 2. Documentação - Programação Java. 3. Verificação em Tempo de Execução. 4. Javadoc. I. Massoni, Tiago Lima. II. Gheyi, Rohit. III. Título.

CDU 004.42(043)

"FOSTERING DESIGN BY CONTRACT BY EXPLOITING THE RELATIONSHIP BETWEEN CODE COMMENTARY AND CONTRACTS"

ALYSSON FILGUEIRA MILANEZ

TESE APROVADA EM 25/05/2018

CAMPINA GRANDE - PB

Resumo

Embora contratos no estilo de programação por contratos (DBC) tragam precisão para expressar o comportamento do código, desenvolvedores são resistentes ao seu uso. Há várias razões para isto, tais como a dificuldade na produção de contratos ou o trabalho de manter os contratos consistentes com o código em evolução. Por outro lado, *Javadoc* é uma abordagem comumente usada para documentar programas Java. Mesmo assim, comentários *Javadoc* não servem para a checagem automática de conformidade devido à ambiguidade inerente à linguagem natural. Neste trabalho, procuramos minimizar a distância entre contratos e *Javadoc*, estimulando a adoção de DBC a partir de duas contribuições principais; primeiro, propomos uma extensão ao sistema de *tags* do *Javadoc* (CONTRACTJDOC) para possibilitar a integração de contratos na notação de comentários; então, propomos uma abordagem para geração de contratos a partir de comentários em linguagem natural (CONTRACTSUGGESTOR). Nós realizamos três avaliações: primeiro, avaliamos a aplicabilidade e a compreensibilidade de CONTRACTJDOC. Como resultados, detectamos inconsistências entre a documentação *Javadoc* e o código fonte. A maioria dos contratos que escrevemos foram checagens de valores limítrofes para parâmetros e repetições de expressões de retorno de métodos. Além disso, a legibilidade dos comentários percebida pelos desenvolvedores não diferiu significativamente entre as abordagens, o que é promissor, dado que contratos são usualmente classificados como difíceis de ler. Segundo, avaliamos a qualidade dos contratos gerados por CONTRACTSUGGESTOR verificando a taxa de falsos positivos gerados. Como resultado, são gerados mais contratos corretos para *non-null* do que para *relational*, devido a quantidade de instâncias de comentários para cada propriedade. Por fim, realizamos estudos de caso com JMLOK2 e CONTRACTOK – CONTRACTOK é uma extensão da abordagem de JMLOK2 para o contexto C#/Code Contracts. Primeiro, usamos JMLOK2 para verificar os contratos gerados automaticamente por CONTRACTSUGGESTOR; depois usamos as ferramentas para verificar 24 sistemas de código aberto (12 para cada ferramenta). As ferramentas detectaram 188 não-conformidades, sendo 72 problemas de pós-condição e 61 de invariante; as causas prováveis mais comuns foram Pré-condição fraca (91) e Erro de código (56). Com isso, objetivamos motivar a adoção de DBC como forma de aprimorar o projeto dos programas, e por consequência, sua qualidade geral.

Palavras-chave: programação por contratos, documentação, verificação em tempo de execução, *Javadoc*.

Abstract

Contracts in Design by Contract style bring about preciseness for expressing the code behavior; however, developers are resistant to their use. There are several likely reasons for this, such as the trouble to conceive good, useful contracts, or the burden of maintaining contracts consistent with the evolving code. On the other hand, Javadoc is a common way of documenting Java programs. Nevertheless, Javadoc comments do not serve to an automated conformance checking due to ambiguity issues inherent to the natural languages. In this work, we try to minimize the distance between contracts and Javadoc, fostering DBC adoption by means of two main contributions; first, we propose an extension to the Javadoc tagging system (CONTRACTJDOC) for allowing the integration of contracts into the comments notation; then, we propose an approach for automatically generating contracts based on natural language code commentary (CONTRACTSUGGESTOR). We perform three evaluations: first, we evaluate the applicability and comprehensibility of CONTRACTJDOC. As results, we detected inconsistencies between the documentation available by means of Javadoc comments and the source code. The majority of the contracts we could write from the comments remains between common-case and repetitive with the code. Moreover, developers' impression about the readability of comments did not differ significantly, which is promising, as contracts are usually regarded as hard to read – one reason for its non-adoption. Then, we evaluate the quality of contracts generated by CONTRACTSUGGESTOR by analyzing the false positives rate. As result, the approach generates more correct contracts for *non-null* than for *relational*, due to the number of comment instances for each property. Finally, we perform case studies with JMLOK2 and CONTRACTOK – CONTRACTOK is an extension of JMLOK2 for C#/Code Contracts context. First, we used JMLOK2 for conformance checking the contracts automatically generated; then, we run the tools over 24 open-source systems (12 with each tool). The tools detected 188 nonconformances. From those, 72 are postcondition and 61 are invariant problems; with respect to likely causes manually established, Weak precondition (91) and Code error (56) are the most commons. With this, we aim to promote DBC adoption as a way for improving the design of the projects, and consequently, their quality in general.

Keywords: design-by-contract, documentation, runtime checking, Javadoc.

Acknowledgments

I would like to express here my sincere gratitude to each one that helped me throughout this Ph.D. research.

To God, that is always with me in all moments, either good or bad. Without His support I would never be able to overcome some daily challenges or achieve my goals.

To my family for their love and support. My parents, Severino do Ramo and Rozane that are my life mentors. Since childhood they have taught me the importance of education and, with life lessons, have showed me how to be a decent human being and good citizen. My sisters Maxwellia and Laryssa, my nieces Emilly and Evellyn, and my brother-in-law Guilherme, for their support and care.

To all my friends, for showing that life cannot be complete without people to share moments.

To professors Tiago Massoni and Rohit Gheyi, that provided me the foundation for an academic career. Their guidance, trust, supervision and directions have greatly helped me in all the time of research and writing of this thesis. Their patience and hard work have inspired me to be a better researcher. I could not have imagined having better mentors for my doctorate.

To my colleagues Catharine, Indy, Kláudio, Matheus, Melquisedec, Mirna, Thaciana, Saulo, members (current and past) of the Software Practice Laboratory (SPLab), for their support, collaboration and good company during this research. To Dênnis, Igor, Clenimar, Bianca, and José Manoel by the technical support during the development of the current work.

To the funding agencies Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) and Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (INES) for their financial support throughout this work.

To professors Adalberto Cajueiro, Adenilso Simão, Ricardo Massa, and Wilkerson Andrade, for their acceptance for reviewing the present work, for their useful suggestions and contributions for this thesis.

Last but not least, I thank the Graduate Program in Computer Science from the Federal University of Campina Grande and its staff for the administrative support.

Contents

List of Symbols

DBC - *Design by Contract*

JML - *Java Modeling Language*

D - *Depth*

B - *Breadth*

LOC - *Lines of code*

#CC - *Number of contract clauses*

#Pre - *Number of precondition clauses*

#Post - *Number of postcondition clauses*

#Inv - *Number of invariant clauses*

#Const - *Number of constraint clauses*

CCo - *Contract complexity*

ARTOO - *Adaptive Random Testing for Object-oriented Software*

FPR - *False Positive Rate*

List of Figures

List of Tables

List of Source Codes

Chapter 1

Introduction

Design by Contract [73] (DBC) is a programming methodology inspired by formal methods research [48] that aims the construction of quality software. DBC is a direct descendant from Hoare's triples [54] – **P {Q} R** which means there is a required connection between a precondition (P), a program (Q) and a description of its execution result (R). According to Hoare [54], "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion."

DBC is based on the establishment of contracts between software modules: clients (those modules using or depending on functionality) and suppliers (those providing some functionality) [73; 74]. In this context, clients must satisfy *preconditions* before calling a supplier; suppliers in their turn, have to provide some guarantees over their results (*postconditions*) [74]. Both clients and suppliers may have contracts with respect to their fields – e.g. establishing the range of valid values – (*invariants*) [74].

With DBC, design errors may be detected during the software development phase, since the design decisions are written in the form of contracts – pre-, postconditions, and invariants – into the source code, thus tools can be used for checking whether the design decisions are being fulfilled. In addition, inconsistencies related to the misunderstanding of the requirements can be detected earlier than in scenarios without the use of contracts since the contracts representing those requirements can be checked at runtime.

The most popular languages and supporting tools for DBC are: Eiffel [75] and the AutoTest [77] tool, Java Modeling Language (JML) [67; 68] for Java with its respective set of tools [12], and Code Contracts [35] for .NET languages with the static checker Clousot [36] and with the dynamic approach IntelliTest.[1] Meyer [75] created Eiffel as an object-oriented programming language with a focus on quality software development. Meyer used the language for illustrating the concepts of Design by Contract methodology [74], such as *preconditions*, *postconditions*, and *invariants*. JML [67] enables behavioral specification for Java by means of qualified comments (comments delimited by @). Code Contracts [35] enables DBC by performing calls to static methods from a .NET library, allowing the use of DBC for languages such as C#.

With DBC the contracts become assertions that can be checked at runtime, fostering reliability for developers. In addition, those contracts enable the checking for semantic issues in a straightforward way since the requirements are closer to the developer in a language more accurate than natural language; so, violations to the requirements can be discovered at runtime. Furthermore, internal problems of a system module are simple to find out due to the use of preconditions and invariants for representing the expected behavior of each part of the module. In this way, unexpected behaviors are caught at runtime.

Design by contract is also a way of providing rich documentation that can be useful for maintaining the systems by enabling developers to understand the systems' behavior by reading the contracts. The contracts are also useful for testing the systems under development by providing an oracle for distinguishing passing from failing tests [83].

The remainder of this chapter has the following structure: first, we present the problem that motivates this work with its relevance (Section 1.1); then, we discuss the proposed solution (Section 1.2); next, we show the evaluation performed (Section 1.3); after that, Section 1.4 summarizes the main contributions of this work; and finally, Section 1.5 outlines the structure of this thesis.

1.1 Problem

In this thesis, we handle the problem of minimizing the programmers' resistance to using Design by Contract methodology. Contracts (such as those in JML [68] notation) bring about preciseness for expressing the code behavior; however, developers are resistant to their use. There are several likely reasons for this, such as the trouble to conceive good, useful con-

[1]IntelliTest is the evolution of Pex [122].

tracts, or the burden of maintaining contracts consistent with the evolving code [109]. According to Schiller et al. [109], there is a large gap between the contracts developers write and those they could write. The authors provide evidence that annotation burden, tooling, and training are primary factors affecting the extent to which developers use contracts as specifications as opposed to argument validation/assertions. On the other hand, Javadoc [65] (and code comments in general) is a common way of documenting Java programs. Even though the approach is quite simple since the developer can use natural language for describing the code behavior; this approach is imprecise, mainly due to ambiguity issues inherent to natural languages. Furthermore, commentary, in general, cannot be automatically checked. Approaches such as @TCOMMENT [118] enable the testing of Javadoc comments, specifically method properties about null values and related exceptions, but this approach is limited to those properties.

1.1.1 Motivating Example

In this section, we provide an example for illustrating the task of translating imprecise comments in natural language (written in Javadoc) to contracts in a contract-based language (JML), and the problem of conformance checking contract-based programs.

In Source Code 1.1, Counter represents a simple counter — visibility issues are omitted, for simplicity. This class has a constructor and two methods: one for updating values and one for resetting values. Comments in natural language (by means of Javadoc notation) guide the developer on the expected behavior of each method. For example, the method `updateCount` has a comment declaring that the parameter `b` must be true in order to enable an increase in the counter (lines 11 and 12).

Aiming the benefits provided by the use of contracts, such as the preciseness for expressing the code behavior and the possibility of runtime checking the methods' behavior (by using the contracts as oracles); one can manually translate the comments to JML syntax (Source Code 1.2). For each method comment, there is a contract for precisely expressing the behavior described. For example, for the `updateCount` method, one may translate the comment *"b equals true implies an increase into count."* into the JML contract `ensures (b == true) ==> (count == \old(count+1))`, in order to express the condition for increasing the value of `count` when the parameter `b` is `true`. JML 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 `Counter` defines the range of valid values for `count` field: [0, `MAX`] (lines 5 – 6). The `\old` clause used in the postcondition refers to the pre-state value of `count` (line 14).

Source Code 1.1: Counter class with Javadoc comments.

```
 1  class Counter {
 2    final int MAX = 3;
 3    /* count must be
 4    * in range [0, MAX]. */
 5    int count;
 6
 7
 8    /* initializes the counter to one. */
 9    Counter() {
10      count = 1;
11    }
12
13    /* b equals true implies
14    * an increase into count.*/
15    void updateCount(boolean b){
16      if(b){
17        count++;
18      }
19    }
20
21    /* count will be reset. */
22    void resetCount(){
23      count = 0;
24    }
25  }
```

Source Code 1.2: Counter class with JML contracts.

```
 1  class Counter {
 2    final int MAX = 3;
 3    int count;
 4
 5    /*@ invariant 0 <= count
 6      && count <= MAX; @*/
 7
 8    //@ ensures count == 1;
 9    Counter() {
10      count = 1;
11    }
12
13    /*@ ensures (b == true)
14    @ ==> (count == \old(count+1)); @*/
15    void updateCount(boolean b){
16      if(b){
17        count++;
18      }
19    }
20
21    //@ ensures count == 0;
22    void resetCount(){
23      count = 0;
24    }
25  }
```

Contracts in JML syntax are precise, however, the developer needs to understand a new language for achieving the benefits from contracts use. In this context, an approach closer to conventional Javadoc-notation could be wider used still providing precision for describing methods behavior by reducing annotation burden [109].

In addition, even though the contracts in this example are for expressing a simple counter, the program presents a nonconformance with its contracts; in the third consecutive call to `updateCount`, as illustrated by the test case in Source Code 1.3, the assertioninstrumented class raises an exception. This is a small example, but finding nonconformances in large systems, by manual analysis (having to deal with inheritance and polymorphism, for example) is certainly harder. In addition, traditional testing is unlikely to uncover this kind of nonconformance due to the generality of the oracles that may miss some contracts. Previously, Milanez et al. [84] proposed an approach implemented by the JMLOK2 tool, which generates test cases with sequences of method calls – like those seen in Source Code 1.3, in order to detect nonconformances. Besides detecting the nonconformance, one must identify the defect causing the failure — a contract, the program itself, or both [94].

Source Code 1.3: A test case for Counter class generated by JMLOK2.

```
void testCounter(){
  Counter c = new Counter();
  c.updateCount(true);
  c.updateCount(true);
  c.updateCount(true);
}
```

However, the developer is left with the task of analyzing the program for fixing the nonconformance. One could consider a few scenarios: the invariant may be too strong, defining an undesirable threshold for `count`; similarly, a precondition for `updateCount` could be added, prohibiting updates when the counter is about to exceed the defined limit. On the other hand, `updateCount`'s body could have enforced the condition, establishing an additional condition to avoid increments that make `count` exceed the limit.

The proper fix is dependent on a number of factors, and it may be related to the code, contract, or both. Fixing the invariant implies there was a defect in defining a limit. Creating a precondition relies on how DBC is applied — whether *defensive programming* [131] is the ruling guideline for system development. The choice gets down to determining where the additional check is placed, either in the contract or in the method body. Since `updateCount` does not have an explicit precondition, it is reasonable to determine the cause as *Weak Precondition*. This problem may be solved by adding a precondition to `updateCount`, relating the method parameter with the current value of `count`.

From now on, we define as *nonconformance type* the contract on which the failure was produced; in the example, the nonconformance is classified as *invariant*. Similarly, regarding likely causes (defect) of nonconformances, we establish the following terminology: weak or strong precondition, weak or strong postcondition, strong invariant, strong constraint, and code error.

1.1.2 Relevance

Contracts bring about preciseness for expressing routine's behavior, however, programmers are resistant to their use. There are several likely reasons for this, such as the trouble to conceive good, useful contracts, or the burden of maintaining contracts consistent with the evolving code. Therefore, an approach for allowing the integration of contracts into the comments notation by offering a few language shortcuts for specifying usual routine behaviors, such as pre- and postconditions and invariants, may help to foster the contracts adoption.

In addition, the possibility of having contracts automatically generated by analyzing natural language comments is also likely to foster Design by Contract because this approach does not require the developer to understand a new language or a new notation; so, the use of DBC becomes transparent.

1.2 Solution

For fostering the use of contracts in systems development, we exploit the relationship between code commentary and contracts. In this context, we propose and implement an extension to the Javadoc tagging system for allowing the integration of contracts into the comments notation: CONTRACTJDOC (Chapter 2). In addition, we propose and implement an approach for automatically suggesting contracts by analyzing tagged-comments written in natural language (Chapter 4).

Between having Java developers specifying JML or writing non-analyzable commentary, there is a potential compromise solution: a tag-based extension to Javadoc. The extension needs to provide a few language shortcuts for specifying usual routine behaviors, such as pre- and postconditions and invariants. And a compiler could translate the notation to assertions that, at runtime, check the conformance of the program behavior to the contracts. In consequence, contracts can be employed in a more natural way, following commonplace standards for code documentation, while adding precision to interface declaration and flags for additional verification support.

Another way of dealing with this problem is to automatically suggesting contracts based on natural language tagged-comments, reducing the annotation burden. For this purpose, machine learning techniques [111] can be applied for classifying the tagged-comments. After classification, the contracts are then generated. After generation, the contracts are checked at runtime in order to discover mismatches (nonconformances [15]) between source code and documentation. As a consequence, the benefits from contracts will be achieved by developers that do not need to learn a contract-based language.

We propose and implement a Javadoc-like language extension (with its respective compiler) called CONTRACTJDOC. CONTRACTJDOC provides a new way for documenting source code in which a developer can specify contracts by adding boolean expressions into brackets and using specific tags into the Javadoc. The approach tries to fulfill the gap between informal documentation (such as JAVADOC) and formal specification (such as JML [68]) by enabling the developer to write contracts by using default tags from JAVADOC (such as `@param`) and some new tags (such as `@inv`) in a controlled way. CONTRACTJDOC supports preconditions, postconditions, and invariants. For creating a contract in CONTRACTJDOC, a developer needs to write a boolean-valued expression into brackets attached to specific tags (such as `@param`). The expression can be composed of method calls and can have quantified expressions.

We propose and implement a machine learning-based approach for automatically generating contracts (pre- and postconditions) by analyzing tagged-code commentary, that we call CONTRACTSUGGESTOR. For enabling the use of machine learning, we needed to create a manually labeled dataset for each property we desired to generate contracts for. Then, we apply CONTRACTSUGGESTOR for generating contracts for two properties of Java programs: non-null and relational (greater than, greater than or equal, less than, less than or equal). Moreover, we use JMLOK2 for conformance checking the generated contracts.

1.3 Evaluation

We evaluate our approaches in several empirical studies [129]. First, we present the studies for evaluating CONTRACTJDOC (Section 1.3.1); then, we show the studies for evaluating CONTRACTSUGGESTOR (Section 1.3.2).

1.3.1 Evaluating ContractJDoc

We evaluate our proposed language - ContractJDoc - by means of three studies: (a) a case study aiming to serve as a proof of concept (Section 3.1); (b) an empirical study with Java developers (Section 3.2); (c) we investigate the comprehensibility on three alternatives for specifying behavior in a Java interface — Javadoc, JML, and ContractJDoc, by means of a survey with developers (Section 3.3). With the case study, we are able to write contracts to six open source Java projects previously annotated with Javadoc comments in natural language. The systems totalize 190,655 lines of code (LOC, henceforth) and we write a total of 3,994 contract clauses (#CC, henceforth). This study also allowed us to detect inconsistencies between Javadoc comments and source code, highlighting the importance of being able to runtime checking comments: without checking, those inconsistencies will remain undiscovered.

Twenty-four Java developers,[2] being 10 professionals (working in industry) and 14 students (only with academic experience) participated in the empirical study. There are 12 trials for the study — each trial is composed by a triple: (task, interface, documenting approach) —. So, there are two developers performing each trial. As results, 83% of the developers were able to perform the required task without perceiving difficulties. When grouping the results by documenting approach, they considered Javadoc as the less complicated for performing the trial, followed by ContractJDoc.

The comprehensibility survey confirmed the results from the empirical study: developers tend to find Javadoc comments more understandable than JML or ContractJDoc. One hundred forty-two developers answered the survey; from those, 36% are professionals and 64% are students (see Section 3.2.2 for details). Thirty-eight percent of the survey respondents chose Javadoc as the most understandable approach regarding interface's behavior, others 32% chose ContractJDoc and 18% chose the same understanding level for all documenting approaches. When asked about the simplest approach to be understood in a general context, 51% answered Javadoc and 29% ContractJDoc; for 13% the understanding is the same for all of them.

[2]We needed to discard three participants due to balancing issues.

1.3.2 Evaluating Contracts

We evaluated the quality of the contracts generated by CONTRACTSUGGESTOR by means of the rate of generated false positives. As results, the approach for non-null property produces more correct contracts than the approach for relational properties. This is explained by the differences between the sizes of the sets for training the machine learning algorithms and by the kind of classification being performed: binary for non-null and multiclass for relational (see Section 4.3 for details).

Finally, we performed a case study concerning nonconformance detection. We checked four systems for which CONTRACTSUGGESTOR automatically generated contracts by using JMLOK2 and detected six nonconformances. Concerning types, all six are precondition problems: five related to non-null property and one related to greater than or equal (relational) property (see Section 5.1).

To increase confidence on the results by JMLOK2 in checking the generated contracts, we performed another case study with JMLOK2 over 12 Java/JML systems (an extended sample of the programs used by Milanez et al. [84]). JMLOK2 detects 119 nonconformances. From those, 51 are invariant errors and 47 are postcondition; with respect to likely causes manually established, Weak precondition (51) and Code error (38) are the most commons. We also collected metrics as a proxy for characterizing the difficult for detecting nonconformances. The results indicated the necessity of at least two changes in average for detecting a nonconformance and the inspection of at least two methods for establishing a likely cause (see Section 5.3).

As an additional contribution, we leverage the RGT-based (randomly-generated tests) approach of JMLOK2 [84] for the C#/Code Contracts context by means of CONTRACTOK. CONTRACTOK is available online,[3] for Windows platforms under the GNU (GNU General Public License) GPL v3. We present also, in this thesis, additional studies on conformance checking of C#/Code Contracts systems. We applied CONTRACTOK over 12 systems. The tool detected 63 nonconformances, being 28 precondition errors and 25 postcondition errors. With respect to likely causes, as for the JML nonconformances, Weak precondition and Code error are the most commons: 40 and 18, respectively. For Code Contracts, the metrics indicated the necessity of at least six changes in average for detecting a nonconformance

[3] `https://github.com/alyssonfm/contractok`

and the inspection of at least four methods for establishing a likely cause. According to these metrics, is harder detecting a nonconformance in Code Contracts systems than in JML systems; in addition, establishing a likely cause is also harder in Code Contracts systems.

1.4 Summary of Contributions

In summary, the main contributions of this thesis are:

- We propose and evaluate a new approach for writing contracts (CONTRACTJDOC);
  - We performed a case study in which we applied CONTRACTJDOC to six open-source Java projects and generated 3,994 contract clauses;
  - We carried out an experimental study and a survey with Java developers for evaluating the readability of CONTRACTJDOC. As results, CONTRACTJDOC was considered intermediate between plain Javadoc and JML;
- We propose and evaluate a contract generation approach (CONTRACTSUGGESTOR);
  - We manually labeled 134,246 comment instances. Then, we created two datasets: one with respect to the non-null property containing all instances, and one with respect to the relational properties containing 1,808 instances;
  - We evaluated the generated contracts with JMLOK2: we applied the tool for conformance checking four Java systems with contracts automatically generated, and it detected six nonconformances;
- We leverage JMLOK2 for C#/Code Contracts context by means of CONTRACTOK [82] and we performed a case study with both tools over 24 systems (12 for each language) in which the tools detected 182 nonconformances. We also manually established likely causes for those nonconformances [80; 81].
- By means of CONTRACTJDOC, CONTRACTSUGGESTOR, JMLOK2, and CONTRACTOK we are able to perform conformance checking and detect inconsistencies in three levels of formality: informal (by generating contracts from Javadoc comments), semi-formal (CONTRACTJDOC), and formal by analyzing JML and Code Contracts.

1.5 Thesis Outline

The remaining parts of this thesis are structured as follows:

Chapter 2: Mixing Contracts with commentary in ContractJDoc. This chapter presents in details ContractJDoc — our new documenting approach.

Chapter 3: Evaluating ContractJDoc. In this chapter, we present the studies performed with the new documenting approach proposed.

Chapter 4: ContractSuggestor. This chapter presents in details our approach for automatically generating contracts based on Javadoc-tagged comments.

Chapter 5: Evaluating Contracts. In this chapter, we present the evaluation of the contracts generated by ContractSuggestor in terms of nonconformance detection. Furthermore, we present the studies performed to increase the confidence on the results by JmlOk2 in checking the generated contracts.

Chapter 6: Related Work. In this chapter, we discuss the main works related to the present thesis. We group the discussion by research area related to our context.

Chapter 7: Concluding Remarks. This final chapter presents conclusions and prospects for future work.

Chapter 2

Mixing Contracts with Commentary in ContractJDoc

Code commentary count as best practice in programming, especially when they enhance public interfaces designed as reusable components for client programs. Adoption is helped by standardized comment notation, such as Javadoc [47] for Java programs, which offers predefined tags (`@param` or `@return`, for instance) that mix nicely with natural language. Despite its value, such a notation does not compel the programmer to provide a precise or complete account of a routine's behavior. Moreover, as pointed by Subramanian et al. [116], the understanding of how to use third-party libraries can be difficult when using only comments in natural language as the source of documentation.

On the other hand, contracts [73] bring about preciseness for expressing the code behavior, however, programmers are resistant to their use. There are several likely reasons for this, such as the trouble to conceive good, useful contracts, or the burden of maintaining contracts consistent with the evolving code [109]. According to Schiller et al. [109], there is a large gap between the contracts developers write and the contracts they could write. They provide evidence that annotation burden, tooling, and training are primary factors affecting the extent to which developers use contracts as specifications as opposed to argument validation/assertions.

In this chapter, we present ContractJDoc, our extension to the Javadoc tagging system for allowing the integration of contracts into the Javadoc notation. ContractJDoc offers a few language shortcuts for specifying usual routine behaviors, such as pre- and postconditions and invariants. The AJMLC-CONTRACTJDOC compiler [101] translates the notation to aspect-oriented assertions that, at runtime, check the conformance of the program behavior to the contracts. In consequence, contracts can be employed in a more natural way, following commonplace standards for code documentation, while adding precision to interface declaration and flags for additional support verification.

We now describe CONTRACTJDOC and explore its features. First, we provide the theoretical background needed for understanding this chapter (Section 2.1). Then, we show the language and describe its infrastructure (Section 2.3).

2.1 Design by Contract

In this section, we present some details on Design by Contract methodology and discuss this methodology for Java. Furthermore, we present the conformance notion considered in this thesis.

Contract-based programs [51] incorporate a language-based solution integrating contracts and code into a single artifact. In this scenario, the Design by Contract [73] (DBC) is a methodology inspired by Formal methods research [48] that aims the construction of quality software. In addition, the methodology is a direct descendant from Hoare's triples [54].

DBC is based on the establishment of contracts between software modules: clients (those modules using or depending on some functionality) and suppliers (those providing some functionality) [73; 74]. Clients need to satisfy some requirements (*preconditions*) before calling a supplier, and suppliers have to provide some guarantees over their results (*postconditions*) [74]. Both clients and suppliers may have contracts with respect to their internal properties (*invariants*) [74]. By applying DBC, contracts become abstractions for methods behavior and runtime checkable; so, they can be used for conformance checking: detecting whether the contract violation occurred in client or supplier side.

2.1.1 JML

In the context of Java development, the Java Modeling Language (JML) [68] is a DBC-enabling notation (and corresponding toolset), with contracts as comments within Java code. JML has a 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 [75] with the specification model-based approach from Larch family of programming languages [52], and some elements from the 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. An `\old` clause refers to pre-state of some value. In addition, the keyword `pure` declares that a method is side-effect free.

In Source Code 2.1 we present a Java/JML program for specifying the method `headElement` (lines 4 – 11) from class `AccountQueueImpl`, a simple implementation for a queue of accounts. In JML the postcondition is represented with the clause `ensures` (line 4); the `\result` (line 4) refers to the return of `headElement` method. 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 [69].

Source Code 2.1: Example of DBC in JML.

```
 1  public class AccountQueueImpl{
 2    Queue<Account> accQueue = new LinkedList<Account>();
 3
 4    //@ ensures (!isEmpty()) ==> (\result instanceof Account);
 5    public /*@ pure @*/ Account headElement(){
 6      if(isEmpty()){
 7        return null;
 8      } else{
 9        return accQueue.element();
10      }
11    }
12
13    public /*@ pure @*/ boolean isEmpty(){
14      return accQueue.isEmpty();
15    }
16  }
```

Concerning tool support for JML, there are three kinds of tools: (i) runtime assertion checkers (RAC) or JML compilers — like *jmlc* [16], *jml4c* [108], *OpenJML* [22], or *ajmlc* [101]; (ii) dynamic and (iii) static conformance checking tools — like JMLUnit [17], JMLUnitNG [133], JET [15], JmlOk [127], and JMLOK2 [84], for dynamic checking; and ESC/Java [37], ESC/Java2 [23], LOOP [9], and JACK [6] for static checking. With respect to JML compilers, *jmlc*[1] is like a Java compiler but aware of JML contracts, adding them into bytecode. *jml4c*[2] is a JML compiler built by extending the Eclipse Java compiler; this compiler supports Java 5 features such as generics. *OpenJML*[3] is a new compiler for JML, yet in development and intends to support new features from Java 8. And *ajmlc*[4] is a seamless aspect-oriented extension to the JML language, compatible with AspectJ. *ajmlc* cleans modularization/specification of crosscutting contracts, such as preconditions and postconditions, while preserving documentation and modular reasoning. AJMLC-CONTRACTJDOC is backed by the *ajmlc* compiler.

2.1.2 Conformance Notion

Contracts help developers during implementation by providing a specification of the expected behavior of the code. Thus, it is important to check the conformance [15; 17] between the source code and the specification. Although other notions of conformance are possible [125], for this thesis we are considering the following: conformance is when the code fulfills the requirements from its contract clauses, in other words, the code satisfies its contracts. When the conformance is broken there is a nonconformance [15; 83; 84]. Since DBC contracts — invariants, pre- and postconditions — can be verified by a compiler, any contract violation between the system modules can be detected immediately, allowing the construction of more reliable systems.

A nonconformance can occur in two cases:

Case 1 When the client satisfies the precondition(s) and the supplier does not satisfy its postcondition(s) — a nonconformance in supplier side;

Case 2 When the client does not satisfy the supplier's precondition — a nonconformance in

[1]`http://www.dc.fi.udc.es/ai/tp/practica/jml/JML/docs/man/jmlc.html`
[2]`http://www.cs.utep.edu/cheon/download/jml4c/index.php`
[3]`http://www.openjml.org/`
[4]`http://www.cin.ufpe.br/~hemr/aspectjml/`

client side.

For example, the code presented in Source Code 2.2 is not in conformance with its contract (Case 1 of nonconformance — the supplier does not return the expected result to its client). As an example of nonconformance from Case 2, consider the Source Code 2.3. 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 an instantiation and a 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.

Source Code 2.2: Code that presents a nonconformance in the supplier side — Case 1.

```java
  //@requires true;
  //@ensures \result >= 0;
  public int getQueueSize(){
          return -1;    // Contract violation here.
  }
```

Source Code 2.3: Code that presents a nonconformance in the client side — Case 2.

```java
1  // supplier side
2  public class MathOperations{
3    //@ requires y > 0.0;
4    public double div(double x, double y){
5      return x/y;
6    }
7  }
8
9  // client side
10 MathOperations mo = new MathOperations();
11 mo.div(3.5,0.0); // Contract violation here.
```

In this work, we are considering only nonconformances from Case 1. Problems related to Case 2 are treated as meaningless [15] due to limitations related to the testing approach considered in this work.

2.2 Software Testing

As stated by Dijkstra [26], software testing just enables us to reveal system defects; however, the tests can be used to increase the confidence in the system behavior. In general, the testing process has two main goals: to demonstrate 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 testing: checks the expected behavior from the system; the second, leads to verification testing: checks if the software meets its stated functional and non-functional requirements. In this work, we use tests concerning both goals depending on the contracts available for conformance checking.

The testing process is an incremental process that starts when a requirement becomes available and continues through all steps of the software development. In this context, the concepts of *error*, *failure*, and *fault* are widely used. According to Binder [10] and IEEE [98], 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, we treat a nonconformance as an anomaly [13; 113]: code or contract that causes a failure in a system.

Testing can occur at three levels of granularity [114]:

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 contracts available, an approach for conformance checking can be used in any level of granularity: unit, component, or system testing. The remainder of this section presents some concepts on software testing that are needed to understand this thesis, like test case (Section 2.2.1), the oracle in DBC context (Section 2.2.2), test generation (Section 2.2.3), and conformance testing (Section 2.2.4).

2.2.1 Test Cases

A test case is composed of inputs, execution conditions, and expected results to test the behavior of a system under test (SUT) [10]. Each test case has the following information:

inputs – conditions that must be satisfied before test execution (preconditions), the data to test the system; the sequence of method calls to be performed; and outputs – postconditions and the output produced by the SUT.

In this work, a test case is a set of method calls in an object under test. The expected results are provided by the contracts. Source Code 2.3 presents an example of a test case, in line 10 there is an object instantiation and line 11 presents a method call.

2.2.2 Test Oracles in DBC Context

A test oracle determines whether the result of a program *p1* using a test case *tc1* is correct [115]. We can use several methods for creating an oracle, 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.

Contracts specify the expected behavior of the software. Since they are executable, they can be monitored at runtime in order to detect any contract violation in the program. However, the quality of the oracle depends on the quality and coverage of the contracts; so, if the program has few contracts, its oracle can assert few properties of the system.

For us, an oracle is achieved by means of the contracts available in the system. Those contracts are used for conformance checking between source code and specification. For example, to `div` method (Source Code 2.3) we show in Source Code 2.4 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 structures – assertion checkers in runtime. First there is a check for invariant violations (line 2 – due to the fact that an invariant must be held before and after every method call), then there is a precondition checking (line 3), if the precondition is respected, the method is executed (line 5) and after method execution there is a postcondition checking (line 6); if there are some contract violation, lines 10 to 12 try to catch, if no contract violation occurs, the invariant is checked again (line 13). Based on the oracle, if no contract violation occurs, we consider that the method (or system) under test is in conformance with its contracts. Otherwise, if the violation occurs at line 3 – checking for precondition violations, we may have a nonconformance if the problem occurs internally in a method call or a meaningless if the violation was produced by a direct call from a test case

Source Code 2.4: Oracle generated to div method.

```
 1  public double div(double arg0, double arg1){
 2    try{ //checks invariant before method execution
 3      try{ //checks precondition
 4        try{
 5            internal$div(double, double);
 6            //checks normal postcondition
 7          }
 8      }
 9    }
10    catch(JMLError){ //catches JML errors }
11    try{ //checks exceptional postcondition }
12    catch(JMLError){...}
13    finally{ //checks invariant }
14  }
```

(Case 2); any violation of invariant (lines 2 and 13) or postcondition (line 6 and 11) represent a nonconformance.

2.2.3 Tests Generation

With respect to test generation, two main approaches are available: white box and black box testing [61]. White box testing (such as Control Flow Testing, Branch Testing, and Loop Testing [63]) is a kind of test where the implementation details 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 [8]) 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 (an oracle). There is also a hybrid approach that mixes features from black and white box approaches; the Gray-box testing [1]. In this, work we consider a black box approach.

Furthermore, the test generation can be performed in two ways: manually – when the tests are written by a tester or developer; 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 [91; 90]. Randoop generates tests using feedback-directed random testing, a technique inspired by random testing that uses execution feedback gathered from executing test inputs as they are created, to avoid generating redundant and illegal inputs [91]. The tests generated are limited to public methods (or constructors) and classes. The tool uses a time limit for stopping the feedback process. The Java version generates tests in JUnit[5] format. Figure 2.1 illustrates the process that Randoop uses for generating tests: (1) sequence of calls for public methods and constructors are created; (2) the sequences are executed; (3) the execution is examined; (4) the feedback from the execution feeds back the process until the time limit be reached; (5) when the time limit is reached the test suite is returned.

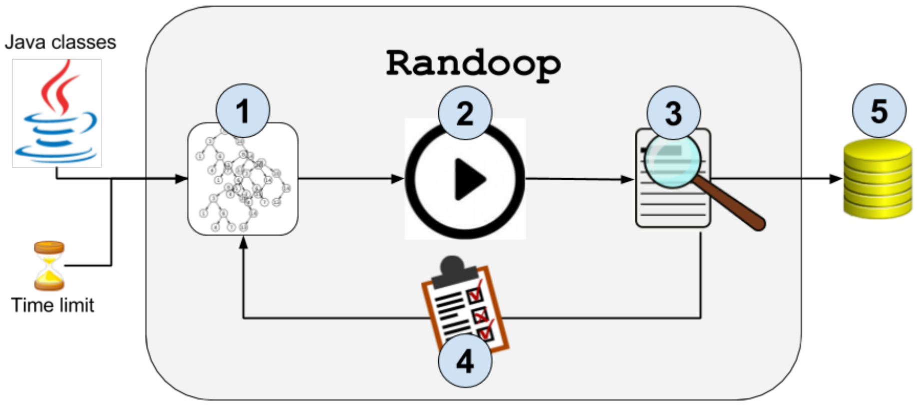


Figure 2.1: The Randoop test generation process. The process starts when are given the following inputs: a list of classes under test and a time limit. Then sequences of method calls are generated, executed and examined; the feedback from the execution feeds back the process until the time limit be reached. In the end, a test suite is returned.

2.2.4 Conformance Testing

Conformance testing is used for verifying whether the implementation of a system conforms to its specification (its contracts), in other words, whether the code satisfies its specification. This kind of test uses the specifications in the source code as oracle and determines the conformance by the adequacy of the test results with those contracts. There are two ways for automatically checking conformance: dynamically and statically. Static conformance checking is done, for example, by means of symbolic execution and abstract interpretation [36]. Dynamic conformance checking is done by running the code and testing for violations of assertions from the contracts, this is the way we use in the present work.

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

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 for s [126]. Following the conformance notion described in Section 2.1.2, a code satisfies its contracts if the code results are expected by its contracts (the test oracles). Therefore, the results of code execution are compared with oracles generated from the contracts.

2.3 CONTRACTJDOC

CONTRACTJDOC provides a way for documenting source code in which a developer can turn Javadoc comments into runtime checkable contracts by using a few new tags and following a specific pattern for writing comments (putting a boolean-valued expression into brackets). With the CONTRACTJDOC approach, we try to fulfill the gap between informal documentation (such as Javadoc) and formal specification (such as JML [68]) by enabling the developer to write contracts by means of default tags from Javadoc (such as `@param`) and some new tags (such as `@inv`) in a controlled way. CONTRACTJDOC supports preconditions, postconditions, and invariants.

For creating a contract in CONTRACTJDOC, the developer needs to write a boolean-valued expression into brackets attached to specific tags (such as `@param`). The expression can be composed by method calls (since the methods used have the tag `@pure`, indicating that those methods are side-effect free) and can have quantified expressions (either universal or existential expressions).

CONTRACTJDOC mixes natural language with contracts syntax combining features from Javadoc and JML. For this purpose, we increase the set of tags from Javadoc and establish a pattern for writing boolean-valued expressions into brackets. The tag being used will determine to which kind of contract — preconditions, postconditions, or invariants — the expression into brackets will be translated. The design of the CONTRACTJDOC language and its supporting compiler are discussed in Section 2.3.1. Next, Sections 2.3.2 to 2.3.6 present in details every kind of contract that CONTRACTJDOC supports.

2.3.1 CONTRACTJDOC Design

We define CONTRACTJDOC constructs as traditional Javadoc tags which are embedded within block comments. The main idea is to allow a mix between the traditional Javadoc syntax and JML.

Embedding contracts mean expressing specifications (e.g., preconditions) in the existing Javadoc comments and making them machine discoverable through the use of marker brackets within those comments. Advantages of embedding a contract language are that programmers do not need to learn a new specification language. This is especially true because the overwhelming majority of contracts that programmers write in practice are short and simple [34; 109]. For instance, 75% of Code Contracts [35] projects, the written contracts are basic checks for the presence of data (e.g., non-null checks) [109]. For scenarios like these, there is no additional effort in embedding such contracts in Javadoc comments using our CONTRACTJDOC language.

We divided our language into two levels: (1) one closer to Java (level 0) for programmers that are oblivious to JML language specification features, and (2) level 1, which is closer to JML specifications for those programmers have some experience/familiarity with JML-like features, such as pre- and postconditions, and invariants.

Level zero is composed of the conventional Javadoc tags, such as `@param` and `@return`, that are useful for specifying pre- and postconditions, respectively. The developer can also use `@throws` or `@exception` for defining the exceptional behavior (an exceptional postcondition) of a method. While using such tags, a programmer treats them like using plain Javadoc comments with no need to use any other tag to write just simple pre- and postconditions [34; 109].

For those already familiar with JML-like features, level one provides some additional tags in relation to the usual Javadoc ones. For instance, tags `@pre` or `@requires` can be used to embed preconditions. Similarly, tags `@post` or `@ensures` can be used to embed (normal) postconditions within Javadoc comments. Exceptional behavior is modeled by exceptional postconditions, which can be embedded with tags `@throws` or `@exception`. Object invariants can also be expressed with CONTRACTJDOC. Tag `@inv` is used to document properties that must hold in all visible states [68].

The grammar for CONTRACTJDOC language is presented in Table 2.1. For purposes of clarification, some constructions are simplified in the grammar. In the next sections, we present examples extracted from the units we used during the case study (Section 3.1) and from the empirical study with Java developers (see Section 3.2) for illustrating all tags supported by our compiler.

2.3.2 Documenting Preconditions

CONTRACTJDOC provides three ways for writing contracts for preconditions that mix traditional Javadoc and JML syntax. The closest to Javadoc is to write the contract clause in a `@param` tag along with a description in natural language; as developers traditionally do into the comments. Table 2.2 shows the grammar for constructing preconditions into our approach. The syntax for postconditions and invariants follow the same pattern (the only change is the kind of tag used `POST_TAG` for postconditions and `INV_TAG` for invariants).

We present in Source Code 2.5 a contract for `name` parameter. In this excerpt, we establish that the name received as the parameter by the constructor of `Authority` — a class from the *Dishevelled* project (see Section 3.1) — cannot be null.

Source Code 2.5: Application of CONTRACTJDOC in a class from Dishevelled project performed during our Case Study (Section 3.1).

```java
/**
* Create a new authority with the specified name.
* @param name authority name [name != null]
*/
public Authority(final String name){
  // ...
}
```

In addition, the tags `@pre` and `@requires` enable the developer to write preconditions for methods without parameters, establishing conditions over fields or other methods. Source Code 2.6 illustrates the tag `@requires` — the syntax for `@pre` is the same. This example, extracted from `AccountStack` interface from the empirical study with Java developers (see Section 3.2), presents the contracts for `push` method. The tag `@requires` is used for establishing conditions over the pure method `numberElements`, which must return a value less than the constant `SIZE`; otherwise, it will not be possible to push an element into the stack. This excerpt also illustrates the use of contracts into `@param` tags: the account to

Table 2.1: Grammar for CONTRACTJDOC language.

Tokens

```
COMMENT_START: /**
COMMENT_END: */
WORD: [A-Za-z0-9_]
PRE_TAG: @param | @requires | @pre
POST_TAG: @return | @ensures | @post
INV_TAG: @inv
DOC_TAG: @doc_open | @doc_public
EXCEPTIONAL_TAG: @throws | @exception | @nothrows
PURE_TAG: @pure
ALSO_TAG: also
NOT: !
BOOLEAN_OP: && | || | ==> | <==>
RELATIONAL_OP: == | != | < | <= | > | >=
QUANTIFIER: @forall | @exists
```

Production rules

```
CONTRACTJDOC ::=
      COMMENT_START
      (DESCRIPTION)*
      (TAG)+
      COMMENT_END

DESCRIPTION ::= * (WORD)*
TAG ::= * (CONTRACT_TAG (WORD)* [(EXP | QUANTIFIED_EXP)*]) | TOKEN
CONTRACT_TAG ::= PRE_TAG | POST_TAG | INV_TAG
TOKEN ::= DOC_TAG | EXCEPTIONAL_TAG | PURE_TAG | ALSO_TAG
EXP ::= BOOLEAN_EXP | METHOD_CALL
BOOLEAN_EXP ::= TERM OP TERM | NOT TERM | @result OP TERM
TERM ::= (WORD)+ | OLD
OLD: (@old( (WORD)+ )
OP ::= BOOLEAN_OP | RELATIONAL_OP
QUANTIFIED_EXP ::= (QUANTIFIER TERM: EXP)
```

Table 2.2: BNF for preconditions in CONTRACTJDOC language.

```
PRECONDITION ::=
      /**
      * (DESCRIPTION)*
      * PRE_TAG (WORD)* [(EXP | QUANTIFIED_EXP)]
      */
```

be added to the stack `acc` must not be null.

Source Code 2.6: Preconditions for the method `push` from `AccountStack` interface — see Section 3.2.

```java

/**
* Pushes an item into the top of this stack if the number of elements in the
* stack is less than SIZE.
* @param acc – the account to be added in a stack. [acc != null]
* @requires [numberElements() < SIZE]
* @return the account being added to the stack. [@return instanceof Account]
*/
public Account push(Account acc);
```

2.3.3 Documenting Postconditions

As for preconditions, CONTRACTJDOC also provides three tags for expressing postconditions. The traditional `@return` from Javadoc comments, and two closer to JML syntax: `@post` and `@ensures`. A postcondition allows us to express the obligations of a supplier (and respectively the rights of a client). By means of a postcondition, we can discover what is expected as result of a method's execution.

Source Code 2.7 exemplifies the use of postconditions in the `Polygon` interface (an interface from the `Aufgabe3` project — see Section 3.1) — we manually turn an expression in natural language into a contract for the method `edges`. The comment in natural language declares the method's return as being always greater than or equal to three, so we turn this statement into a runtime checkable expression by using the `@return` tag.

Source Code 2.7: Excerpt illustrating the use of postconditions in CONTRACTJDOC approach.

```java
/**
* Returns the number of edges of a polygon as integer.
* @return Integer the number of edges. Is always >= 3 [@return >= 3]
*/
int edges();
```

2.3.4 Documenting Invariants

In addition to the support of pre- and postconditions, CONTRACTJDOC enables the use of invariants by means of a new tag: @inv. The format of writing is the same as those for pre- and postconditions. The difference is related to the semantics: while pre- and postconditions apply to a specific method, an invariant must be held for all methods from a class. Being checked after the constructor and before and after every method [69].

In Source Code 2.8, we present an invariant for the field name from class Product (a class from the SimpleShop project — see Section 3.1). In this case, the name must be not null after the constructor and before and after every method call to this class.

Source Code 2.8: Excerpt illustrating the use of invariants in CONTRACTJDOC approach.

```
public class Product{
  /**
   * @inv [name != null]
  */
  /**
   * @doc_open
  */
  private String name;
  // ...
}
```

2.3.5 Quantifiers

CONTRACTJDOC supports the use of existential and universal quantifiers, by means of @exists and @forall tags, respectively. In Source Code 2.9 (extracted from vocabulary a project from the Dishevelled unit) the comments in natural language state the conditions for a universal quantifier over the accessions from the field concepts: the accession specified as the parameter must be unique within the domain. In the @forall clause, we declare that the accessions for each concept must be equals to the accession received as the parameter. With this clause, we ensure that the only way of calling this method (createConcept) is by passing an accession that is unique to the current domain.

Source Code 2.9: Excerpt illustrating the use of universal quantifiers in CONTRACTJDOC approach.

```
/**
```

```
 * Create and return a new concept in this domain with the specified name,
 *  accession , and definition . The specified accession must be unique within
 *  this domain and may not be null.
 *
 * @param name concept name
 * @param accession concept accession , must be unique within this domain
 *     and may not be null
 *
 * [accession == null || (accession != null &&
 *      (@forall int i; i >= 0 && i < concepts.size();
 *              !((Concept)concepts.toArray()[i]).getAccession().equals(accession)))]
 *
 * @param definition concept definition
 * @return a new concept in this domain with the specified name, accession ,
 *   and definition .
 */
public Concept createConcept(final String name, final String accession ,
      final String definition){
  //...
}
```

2.3.6 Other tags

As in JML, in CONTRACTJDOC is also possible to make references for the return value of a method and for the pre-state of the value received as the parameter. The tag for the return value is `@return`, this tag can be used to denote a postcondition clause and for referring to the value being returned by a method. In other words, a developer can use the traditional `@return` as in Javadoc and put this tag into brackets in order to establish a boolean expression that will be evaluated as a postcondition (see Source Code 2.7). For referring to the pre-state value the tag is `@old`. Similar to JML definitions [68], the tag `@old` cannot be used inside a precondition or an invariant clause.

The tags `@doc_open` and `@doc_public` are used for changing the visibility of a method/field in the context of a contract — as `spec_public` from JML language [68] (see Source Code 2.8, lines 5 - 7 for an example). Changing the visibility of a method/field may be useful for internal contracts of a given system. Furthermore, `@pure` declares that a method is side-effect free — a pure feature is one that has no side effects when executed [69]. This tag enables the use of a method into a CONTRACTJDOC contract. For example, for fields with accessor methods can be preferable to use those methods into the contracts instead of directly using the field.

2.3.7 CONTRACTJDOC's Supporting Infrastructure

We implemented the CONTRACTJDOC compiler in the top of the open source AspectJML/ajmlc compiler [100; 101; 102]. Unlike the original JML compiler (jmlc), ajmlc presents code optimizations and improved error reporting [101]. Differently from jmlc, AspectJML also enables the modularization of crosscutting contracts that can arise in standard JML specifications [100].

We adapted the front-end of the AspectJML/ajmlc compiler to convert/preprocess the CONTRACTJDOC tags into the corresponding JML features, like pre- and postconditions. After conversion, the compilation occurs as usual and generates aspects to check the contracts during runtime. See Figure 2.2 for an overview of the compilation strategy. First, a source code with CONTRACTJDOC contracts passes through a tag processor and type checker. Then, the assertions generated are runtime checked and AspectJ compiler produces a bytecode with assertions.

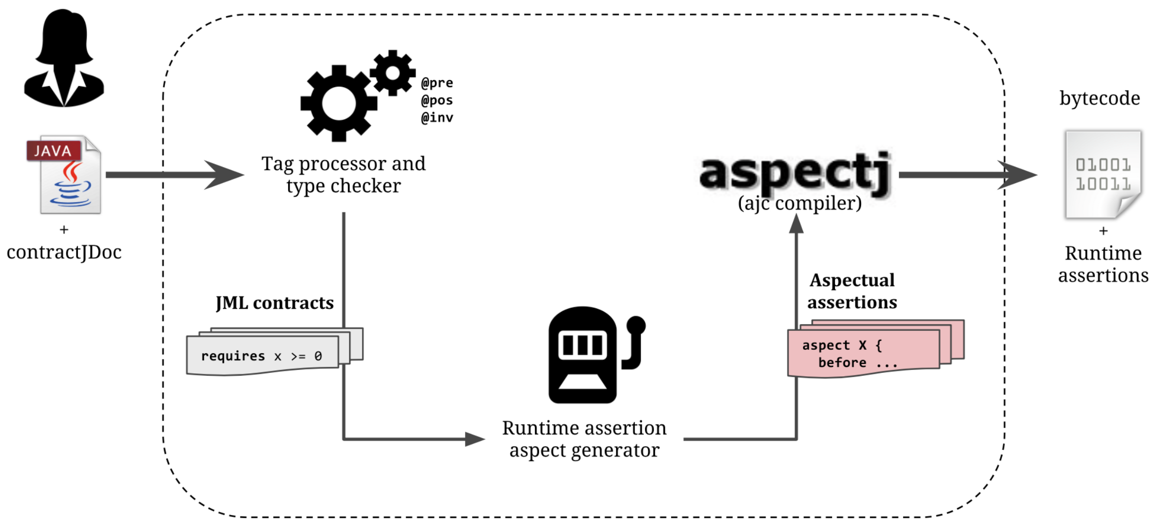


Figure 2.2: ContractJDoc Compilation. First, a source code with CONTRACTJDOC contracts passes through a tag processor and type checker. Then, the assertions generated are runtime checked and AspectJ compiler produces a bytecode with assertions.

Chapter 3

Evaluating ContractJDoc

In this chapter, we evaluate ContractJDoc approach by performing three studies [129]. First, we apply ContractJDoc to six Javadoc-annotated open source systems (see Section 3.1) aiming at a proof of concept of the compiler implementation and the language applicability. Then, we observed 24 developers programming for Java interfaces with behavior documented by the conventional Javadoc, conventional JML, and ContractJDoc, within a controlled environment (see Section 3.2). Finally, we investigate the comprehensibility on the three documenting approaches considered in this thesis for specifying the behavior of a Java interface — Javadoc, JML, and ContractJDoc, by means of a survey with Java developers (see Section 3.3).

3.1 Case Study

In this case study, we apply ContractJDoc to six Javadoc-annotated open source systems to evaluate compilation and generation infrastructure, assessing ContractJDoc's applicability and effectiveness.

3.1.1 Definition

This study aims at assessing ContractJDoc applicability, with respect to automation benefits, from the point of view of Java developers. We observe the results from applying ContractJDoc to six Javadoc-rich open source systems; all their method-level Javadoc annotations are manually translated to ContractJDoc, before running tests looking for mismatches between specifications and actual method behavior. In particular, our case study addresses the following research question:

Q1. Is the preciseness of ContractJDoc, when compared to Javadoc, useful for contract verification?

We collected a few open source systems based on their use of Javadoc, applied ContractJDoc to each system and evaluated results in terms of detected conformance errors.

3.1.2 Systems

The case study was performed on a convenience sample: six Javadoc-rich open source systems available at GitHub[1] repository. They were selected based on the presence of method-level Javadoc annotations. Projects are searched by the following set of key phrases: "must be", "must not be", "should be", "should not be", "greater than", "not be null", "less than" into Javadoc comments. After some visual filtering, we collected the five most important classes in each system, based on overall dependence, and check whether those classes contained method-level Javadoc comments for most of their methods. If so, the system is selected. Finally, we checked whether the system presented a suite of unit test, which are run during the case study to detect inconsistencies. We were able to find four systems meeting these criteria, although we performed the manual translation to six systems.

While `ABC-Music-Player` plays music from an ABC file (part of a project assignment from MIT class 6.005), `Dishevelled` hosts free and Open Source libraries for several user interface components and supporting code, with emphasis on views and editors for complex data structures, like collections, sets, lists, maps, graphs, and matrices; `Jenerics` is a general-purpose set of Java tools and templates library. On the other hand, `OOP Aufgabe3` aims to manipulate polygons. `SimpleShop` is an electronic shopping system. In addition, `Webprotégé` is a collaborative ontology development environment for the Web. Those systems amount to more than 190 KLOC. See Table 3.1 for details in terms of code lines (LOC) – only real source code lines, blank lines, closing brackets and comments are not considered, total contract clauses (#CC) – following [34] approach, in which the number of contract clauses is a proxy for contract complexity – as split into

[1]https://github.com/

preconditions (#Pre), postconditions (#Post), and invariants (#Inv).[2]

Table 3.1: Case study Systems. LOC shows the code lines (LOC), total contract clauses (#CC), as split into preconditions (#Pre), postconditions (#Post), and invariants (#Inv)).

System	LOC	#CC	#Pre	#Post	#Inv
ABC-Music-Player	1,973	115	41	74	0
Dishevelled	110,577	2,655	1,405	1,250	0
Jenerics	2,538	190	105	85	0
OOP Aufgabe3	353	54	28	26	0
SimpleShop	472	50	16	15	19
Webprotégé	74,742	930	351	579	0
Total	**190,655**	**3,994**	**1,952**	**2,029**	**19**

The manual translation abides by the following criteria: method-level comments were considered preconditions if the comments establish some restriction over the method parameters. For instance, "`@param notes - Should not be null and should be of length >= 2`" was replaced by the following ContractJDoc-based expression `[notes != null && notes.size() >= 2]`, and postconditions that establish details on the return value of the methods, e.g. "`@return Integer the number of edges. Is always >= 3`" was replaced by `[@return >= 3]`. Class-level comments make up for invariants when they describe properties over fields that must be maintained for all methods of the class.

3.1.3 Experimental Procedure and Research Method

Three researchers applied ContractJDoc in six existing open-source systems available at GitHub. They followed a bottom-up approach for writing the ContractJDoc contracts: the researchers started applying ContractJDoc in the simplest methods and classes (or interfaces), following up to the most complex. Contracts followed the Javadoc comments available in natural language (in English) and some of them were inferred from the experimental unit's context or from the methods' source code. As result, they wrote 3,994 contract clauses: 1,952 preconditions, 2,029 postconditions, and 19 invariants (see Table 3.1). The process of applying ContractJDoc is composed of four steps: 1) generation of the contracts based on the natural language comments available (as shown in Section 3.1.2); 2)

[2]The clauses correspond to the contracts we applied in each system.

contracts are compiled by means of AJMLC-CONTRACTJDOC compiler, in order to generate the bytecode enriched with assertions; 3) the test suite available in each system is run over the contract-aware bytecode; 4) results of the test suite execution are analyzed and conformance errors are investigated.

Concerning the kind of written contracts, we group the contracts according to the approach of Schiller et al. [109]: application-specific contracts (AppSpec.) – the kind of contracts that enforce richer semantic properties: valid argument values, how a state is modified, and the relation between expressions; common-case contracts (Com.Case) – the kind of contracts that enforce expected (common) program properties: that data is present, strings are not blank, collections are not empty; code-repetitive (Repet.) – the kind of contracts that repeat exact statements from the code.

All systems with the contracts added in this study are available in a replication package.[3] Concerning the verification performed after applying CONTRACTJDOC contracts into the systems, we used the test suites available for the purpose of identifying problems (four out of six projects have a test suite available). Every test case that failed was investigated in order to find out if it was a conformance error in the system.

As a secondary goal, the study allowed us to check the expressiveness of CONTRACTJDOC and to evaluate the effort related to adding contracts to existing systems. In addition, we enhanced the compiler and added features in order to make the process of applying CONTRACTJDOC in an existing project simpler. With respect to the overhead of contracts, it is dependent on the number of contracts we add to each system. For example, in the case of `Webprotégé`, *javac* compiles the whole project in 19 seconds whereas AJMLC-CONTRACTJDOC takes around 700 seconds for it.

3.1.4 Results

Table 3.2 presents the results of applying CONTRACTJDOC to each system. Column #Clauses displays the number of clauses manually added in each system. Column #Anomalies presents the number of anomalies detected by the systems test suite after compiling the source code enhanced with contracts in CONTRACTJDOC approach. Column Time reveals the time (in seconds) needed for compiling the whole project with its dependencies before

[3] `https://goo.gl/yO8or2`

applying ContractJDoc contracts. Column Time' reveals the time (in seconds) needed for compiling the whole project with its dependencies after applying ContractJDoc contracts. Columns #Com.Case to #Repet. show the contract clauses added in each system grouped by type (following the definitions from Schiller et al. [109]).

Table 3.2: Case study Results.

System	#Clauses	#Anomalies	Time (s)	Time' (s)	#Tests	#Com.Case	#AppSpec.	#Repet.
ABC Music Player	115	2	1	9	30	42	11	62
Dishevelled	2,655	381	59	434	2,643	1,536	151	968
Jenerics	190	7	1	20	44	156	0	34
OOP Aufgabe3	54	1	1	4	11	16	30	8
SimpleShop	50	0	1	5	0	30	11	9
Webprotégé	930	0	19	713	0	717	79	133
Total	**3,994**	**391**	**82**	**1,185**	**2,728**	**2,497**	**282**	**1,214**

Concerning the kind of contracts, the only unit in which we wrote more application-specific contracts was `OOP Aufgabe3` system (55% of the written contracts are application-specific). On the other hand, in `ABC-Music-Player`, more than 90% of the contracts remain between common-case and repetitive code: verifications that strings are not blank, collections are not empty, or that a method returns a field. For `Dishevelled`, the majority of the written contracts is classified as common-case (57.51%), other 36.92% are repetitive with code and only 5.57% are application-specific. In addition, all contracts written for `Jenerics` are related to verification of nullity from parameters or the return value, thus all contracts remain between common-case and repetitive code. In `SimpleShop`, the written contracts are distributed in the following manner: common-case 60%, repetitive code 19%, and application-specific 21%; again the number of common-case and repetitive code outperforms application-specific contracts. Finally, in `WebProtégé`, the distribution is: common-case 77.51%, repetitive code 14.38%, and application-specific 8.11%.

When applying ContractJDoc to `ABC-Music-Player`, we found inconsistencies between Javadoc comments and the source code. The problems occurred in the class `Utilities` (package `sound`) because there are comments concerning a parameter declaring that the value of this parameter must not be greater than or equal to zero; however in the body of the methods there is an if-clause that throws exceptions when the value received by the parameter is negative.

3.1.5 Discussion

For all systems (see Table 3.1), we wrote more pre- and postconditions than invariants. This result has the following explanation: as expected the amount of Javadoc comments over the classes' fields in the evaluated systems is low in comparison with the amount of Javadoc comments over method's parameters and return.

Concerning pre- and postconditions, for `ABC-Music-Player` and `WebProtégé` projects, we wrote almost twice as many postconditions as preconditions. In `ABC-Music-Player` this is related to the number of accessor methods available and for `WebProtégé`, the difference is related to the available comments.

We were able to detect potential inconsistencies in `ABC-Music-Player`; the exception will be always thrown, differently from what is expected from the commentary. We also found a problem with the `WebProtégé` project, in the class `OWLLiteralParser` there was one exception in the Javadoc tag `@throws` that was not declared in the throws of the method's signature. Those inconsistencies may be related to the lack of synchronization between source code and code commentary updates. Therefore, an approach such as CONTRACTJDOC may be useful for detecting those problems.

In addition, sometimes the tests available along with the systems do not respect the definitions from the Javadoc comments. For instance, when the comments in natural language from `ABC-Music-Player` system are turned into CONTRACTJDOC contracts, some tests from `MainTest`, `ParserTest`, and `SequencePlayerTest` violate the methods' preconditions from class `Utilities`, they try to call `Utilities`' methods by passing the value zero as the second parameter, even though the comment declares the second parameter must be greater than zero. This scenario also occurred in `Dishevelled` unit, the comments turned in CONTRACTJDOC contracts also enable us to detect some tests that do not respect the restrictions available in the Javadoc comments. These results open place to researches concerning the relationship between tests and code commentary.

As a proof of concept, CONTRACTJDOC and its compiler (AJMLC-CONTRACTJDOC) enabled us to write runtime checkable code for third-party systems based on the comments in natural language. As expected, the quality and variety of the contracts depended strongly on the available comments, however, we were able to detect and correct inconsistencies and missing expressions between source code and comments.

3.1.6 Threats to validity

Due to its size, results from this study cannot be generalized; its purpose is evaluating the applicability and relative usefulness. The sample is not representative since there is no available estimate of the Javadoc-rich project population in GitHub, then probability sample is impossible. Our approach is as systematic as feasible in selecting the evaluated project – manual translation does not scale, then the sample contains only six projects. Therefore, those systems may not be representative of the real use of Javadoc in real systems; however, we were able to detect inconsistencies between Javadoc comments and source code, as occurred in `Utilities` class (`ABC-Music-Player` experimental unit) in which the comment for a parameter of the methods is the right opposite of the expected behavior in the source code.

`Dishevelled` and `WebProtégé` sizes set them apart from the other systems. For instance, `Dishevelled` is more than 56 times more LOC than `ABC-Music-Player`, 43 times more LOC than `Jenerics`, 313 times more LOC than `OOP Aufgabe3`, and 234 times more LOC than `SimpleShop`. In order to reduce the threat on the manually-defined contracts, all systems were annotated and reviewed by three researchers, separately.

3.1.7 Answer to the Research Question

Q1. Is the preciseness of CONTRACTJDOC, when compared to Javadoc, useful for contract verification?

By adding comments in CONTRACTJDOC style, we were able to detect inconsistencies between source code and comments that would remain unnoticed until a client write a code based on the documentation of the class and not be able to get the expected value. Furthermore, we detected that the test suite made available along with the systems do not respect the definitions into the Javadoc comments.

3.2 Empirical Study

In this empirical study, we ask some Java developers to perform an implementation task based on a documented-interface, aiming to evaluate the readability and understandability of three approaches for documenting Java code.

3.2.1 Definition

The goal of this empirical study is to investigate CONTRACTJDOC, for the purpose of evaluation with respect to readability and understandability, from the point of view of developers in the context of Java programming language. We group our research questions concerning the factors evaluated. The study presents two factors: the task performed by the developer (task), and the documenting approach (approach). Those factors have the following treatments: client and supplier[4] – for the task; and Javadoc, CONTRACTJDOC, and JML – for the approach.

In particular, the empirical study addresses the following research questions:

Q2. With respect to the task performed by each developer, how difficult is the required task?

We ask the developers to perform a task based on a given documented interface. Then, we ask them to evaluate the difficulty by means of a Likert-type scale and perform statistical tests for checking differences according to the task performed.

Q3. Regarding the documenting approach used by the developer, how difficult it is for a developer to perform the required task by using the available documented interface?

By using a Likert-type scale, we measure the difficulty for each approach considered in this experiment and performed statistical tests for comparing the approaches.

Q4. Are there faults in the produced source code?

We check the source code produced by each developer by using some manually produced test cases in order to investigate the occurrence of faults with respect to the rules available in the comments.

3.2.2 Participants

The participants of our experiment (called Participants henceforth) are grouped in professionals and students. Professionals those who work or already have worked with program-

[4]By client, we mean a class calling the methods provided by an interface, and by supplier, we mean a class implementing the interface.

ming to the industry. Students those who have only academic experience. All participants have experience on programming in Java for at least one year. The majority works with Java for more than three years.

3.2.3 Study Design

In this study, we addressed two factors: approach for commenting source code, task to be performed; with the following treatments: Javadoc, CONTRACTJDOC, and JML – for approach, and client and supplier – for the task. Moreover, we use two Java equivalent interfaces in this experiment: Stack and Queue. We randomly assigned subjects to each combination of the treatments [129]. For the purpose of this experiment, each triple <approach, task, interface> is called a trial (a combination of treatments). Since there are three documenting approaches, two tasks, and two Java interfaces, the experiment counts with 12 trials. The assignment Participant — trial is performed by using a completely randomized design in order to not bias the results. The experiment uses a balanced design, which means there is the same number of people in each trial [129] – two in this experiment.

3.2.4 Experimental Procedure

The experiment was executed offline, i.e., participants received the experimental material via an online Survey platform[5] that we use to collect the results. Each participant received an experiment package, consisting of (i) a statement of consent, (ii) a pretest questionnaire, (iii) instructions and materials to perform the experiment, and (iv) a post-test questionnaire. Before the study, we carried out a 20-minute presentation showing features from each documentation approach and explaining, in detail, the task we asked the developer to perform.

Before starting the experiment, we asked each participant to fulfill a pre-study questionnaire reporting their programming experience (with respect to Java and contract-based programming experience). After filling in the questionnaire, we randomly selected a task for each of them.

The first part of the experiment consists of the following activities: (i) apply a question-

[5]An instance of the platform used is available online: `https://www.formpl.us/form/5671648952844288`

naire pre-experiment – in order to collect information on developers experience; (ii) give some kind of training on the documenting approach, such as JML and CONTRACTJDOC; (iii) ask them to execute the tasks – for each developer will be given one task for one approach with one Java interface; (iv) apply a post-experiment questionnaire – in order to collect qualitative information about the developers' view of each task. The post-experiment questionnaire presents a Likert-scale question with five levels: varying from *very difficult* 1 to *very easy* 5; and one opened question for the developer to provide details of the answer.

3.2.5 Instrumentation

In this empirical study, we performed a pilot with three Java developers in order to fit the structure of the questions, the way in which we make the data available for the developers. As a result, we changed the manner of making the working dataset available to the participants. Initially, we were making the documented interface available in a link and the working dataset in another. The answers to the pilot highlighted this fact and we decided to create a single package containing all Java classes (all classes needed to the compilation of the code) related to the experiment in a single URL.

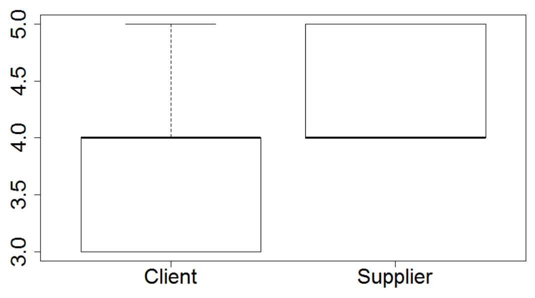
We manually created a set of tests for checking each property documented in the documenting interfaces and used those tests for checking the code produced by the participants of this study. The tests we created are available on our companion website [85].

3.2.6 Results

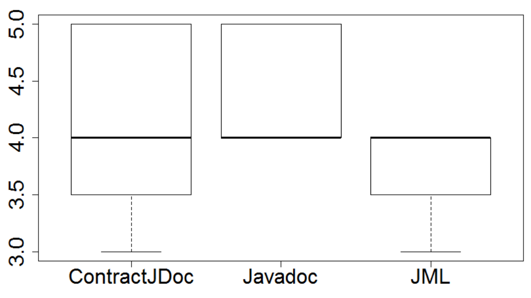
27 Java developers participated in our experiment. From those, we randomly discarded three in order to maintain a balanced number of participants in each trial. Thus, we maintained 24 developers, from those, 10 are working in the industry (41.6%) and 14 are students (58.4%).

Concerning the task performed by the developers, we present in Figure 3.1a answers on difficulty grouped by the task performed. The implementation of the documented interface (Supplier task) seemed to be easier than the task of creating a Client class, however, this difference is not statistically significant as presented by a Wilcoxon rank sum test [62] (p-value = 0.07, with a confidence level of 95%).

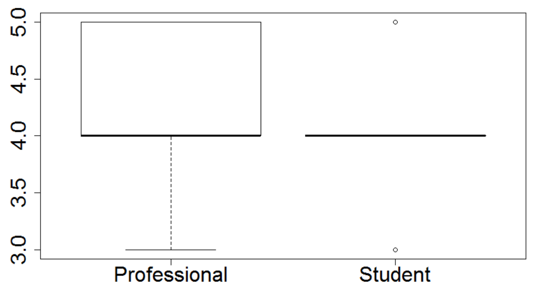
When grouped by documenting approach (see Figure 3.1b), the Kruskal-Wallis rank sum



(a)



(b)



(c)

Figure 3.1: Results of our empirical study with Java developers, on an implementation task based on a documented-interface, aiming to evaluate the readability and understandability of three approaches for documenting Java code.

test showed no difference between the approaches (p-value = 0.15).

When grouped by experience (Figure 3.1c), the Wilcoxon rank sum test (p-value = 0.45) also does not show differences statistically significant between professionals and students.

Javadoc and CONTRACTJDOC were the only documenting approaches in which all participants were able to produce a code satisfying the oracle (respecting the restrictions available in the comments). On the other hand, there was one case developed by following the JML documenting approach in which the contract is not satisfied by the implementation.

3.2.7 Discussion

We proceed with the discussion of the research questions.

Q2. We ask each developer to perform one task: either implement a given interface or implement a client code for using the methods provided by an interface – such the use of an API (Client). Although developers have assigned more *Very Easy* and *Easy* for the supplier task than to the client (see Figure 3.1(a)), the statistical test did not provide evidence for a significant difference between the difficulty perceived by developers when performing the required task.

Q3. Although not supported by statistical tests since Kruskal-Wallis rank sum test showed no difference between the approaches (p-value = 0.15), Javadoc and CONTRACTJDOC were perceived as being more understandable than JML (see Figure 3.1b). This visual result – available as a boxplot – indicates CONTRACTJDOC as an approach in an intermediate level between Javadoc and JML, with both providing runtime conformance checking. Therefore, the proposed approach is promising: CONTRACTJDOC is easy to understand (create a code based on the comments) – 75% of the developers' answers for difficulty remains between *Easy* and *Very easy* – and enables the runtime checking of the comments by means of AJMLC-CONTRACTJDOC compiler.

Q4. Concerning code correctness, all Participants using interfaces documented with Javadoc and CONTRACTJDOC produced code in accordance with the contracts available in the interfaces. Only one developer using an interface with JML contracts was not able to satisfy all the contracts: in one method the source code produced is not in conformance with the contracts.

Even though the developers have perceived the supplier task as less difficult than the client task (according to the visual information from the boxplot), they produced code respecting the restrictions available on the comments more times to the client task. All developers were able to write a client code in accordance with the restrictions. Maybe the difficulty reported by the developers is related with the attention required for using methods provided by an interface: one needs to read the documentation available in order to know how to use the methods; whereas implementing the interface is more simple; mainly for the interfaces used in this experiment: they are traditional and well-known data structures. The results of this experiment suggest that when writing a client code, developers tend to pay more attention to the documentation available than when they are writing a supplier code (implementing a given interface).

3.2.8 Threats to validity

Concerning internal validity, all material for the empirical study is available only in English, therefore, the experience of the participants with English may have affected their performance. In addition, the interfaces were documented by the researchers and this may have affected the results. With respect to construct validity, the answers from developers may not be representative of their real opinion on difficulty perception; to overcome this threat we made a space for comments available along with the Likert-scale questions, which are taken into account when collecting the answers.

Regarding external validity, only 24 developers participated in this study and this sample is not representative of the real population of Java developers. In addition, we used only two similar data structure interfaces (queue and stack). In other domains with more complex structures, the results may vary.

3.2.9 Answers to the Research Questions

- **Q2.** With respect to the task performed by each developer, how difficult is the required task?

  Analyzing the answers grouped by task (see Figure 3.1b), developers did not find difficulty in performing the required task. They provided all answers between neutral (3) and very easy (5).

- **Q3.** Regarding the documenting approach used by the developer, how difficult it is for a developer to perform the required task by using the available documented interface?

  Although not supported by statistical tests, Javadoc and CONTRACTJDOC were perceived as being more understandable than JML (see Figure 3.1b). This result indicates CONTRACTJDOC as an approach in an intermediate level between Javadoc and JML.

- **Q4.** Are there faults in the produced source code?

  Concerning code correctness, all Participants using interfaces documented with Javadoc and CONTRACTJDOC produced code in accordance with the contracts available in the interfaces. Only one developer using an interface with JML contracts was not able to satisfy all the contracts: in one method the source code produced is not in conformance with the contracts.

3.3 Comprehensibility Survey

We conducted an exploratory study that involved data collection through a survey with Java development practitioners. The goal of this survey is to compare three documentation approaches (Javadoc, CONTRACTJDOC, and JML) with respect to comprehensibility, from the point of view of developers. In particular, the survey addresses the following research question:

Q5. In the developer's opinion, which approach, among Javadoc, CONTRACTJDOC, and JML, is the most effective in communicating a routine's behavior?

3.3.1 Survey Design

For this study, we followed a quantitative method based on a web-based survey instrument, suited to measure opinions and behaviors in response to specific questions [30], in a non threatening way.

The survey instrument[6] begins with a purpose of clarification along with a consent term. Then, a characterization of the respondent is conducted by some questions related to Java experience and experience with contract-based programming. Next, the survey is presented:

[6] `https://goo.gl/forms/8W9jUMGCavkkzDj12`

links for three Java interfaces with each one documented in a different approach is showed, then some questions related to the understanding of the behavior of a class implementing the interfaces based on the comments available is asked.

We used Likert-scale questions. In two questions we ask the developers to choose the most simple/understandable documentation approach: one question related to the provided interface; and one question concerning the use of the approach in a general context. The questions have five levels: varying from *very difficult* 1 to *very easy* 5, with 3 as the neutral answer. The levels are: *very difficult* – 1, *difficult* – 2, *neutral* – 3, *easy* – 4, and *very easy* – 5.

3.3.2 Participants

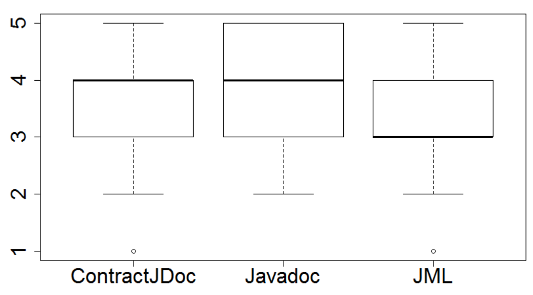
The survey participants are extracted by means of a non-probability sampling technique – convenience sampling [129]: the nearest and most convenient persons are selected as subjects. We send the survey link to academic and professional mailing lists. In addition, our contacts made a snowball approach, sending the survey to their respective contact lists, increasing the sample and the number of participants in our study. The survey was open for three weeks (from June to July 2016) and received 142 answers (from an estimated total of 700 who received the link, 20% response rate).

3.3.3 Results

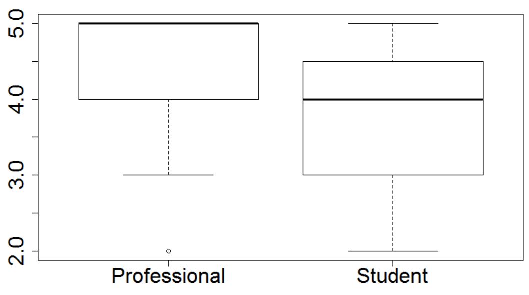
142 Java developers answered the survey. From those, 51 are professionals (36%) and 91 are students (64%). The survey participants did not participate in the experimental study (Section 3.2). All participants of the survey have experience on programming in Java for at least one year. The majority works with Java for more than two years.

With respect to the survey answers, 50.7% (72) of the Subjects chose Javadoc as the simplest approach to understand when using it in a general context. In addition, for 38% (54) of the subjects, Javadoc is also the most understandable approach with regard to the provided interface.

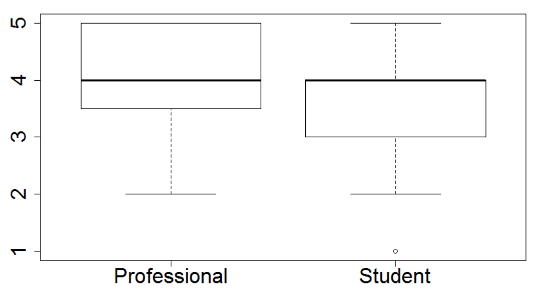
The survey results provided us statistical difference when comparing the comprehensibility of the documentation approaches evaluated (see Figure 3.2a). By performing a Oneway



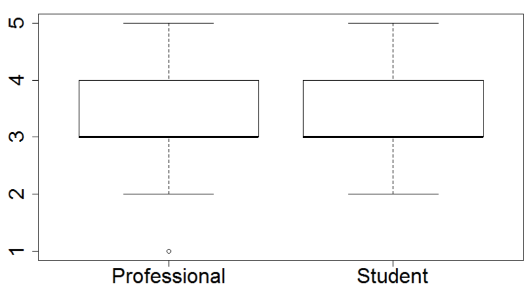
(a) All approaches



(b) Javadoc



(c) CONTRACTJDOC



(d) JML

Figure 3.2: Subjects' answers to the individual evaluation of comprehensibility for each documentation approach. And answers grouped by experience for each approach.

ANOVA test [62] and a corresponding post hoc analysis we were able to distinguish the three approaches ($p\text{-value} < 0.05$). The Tukey HSD [62] and pairwise comparisons using t-tests with Bonferroni correction [62] produced the following p-values: Javadoc-ContractJDoc = 0.012, JML-ContractJDoc = 0.000, and JML-Javadoc = 0.000. These results indicate that the comprehension perceived for the three approaches are different, however, when grouped in pairs, the comprehension for Javadoc and for CONTRACTJDOC are closer (p-value closer to 0.05) than for the other comparisons. This result is also supported by the *diff* value returned by Tukey HSD test: Javadoc-ContractJDoc = 0.296, JML-ContractJDoc = -0.458, and JML-Javadoc = -0.754.

When analyzing data grouped by experience (Figures 3.2b to 3.2d) by means of Wilcoxon rank sum test with continuity correction tests, only for JML we found no statistical difference between Professionals and Students (p-value = 0.17). For both Javadoc and CONTRACTJDOC, Professionals had perceived the approaches as being more comprehensible than Students (p-value = 0.012 and p-value = 0.004, respectively).

3.3.4 Discussion

According to the statistical tests performed, Javadoc is the most understandable documentation approach, and CONTRACTJDOC is intermediate between JML and Javadoc, being closer to Javadoc. This can also be seen in Figure 3.2a.

An interesting result came from the analysis of the difficulty grouped by experience (Figures 3.2b to 3.2d): students and professionals have perceived the same level of difficulty for JML, which is promising as contract-based languages are usually considered harder to be understood by people with less experience (students, in our survey). This result suggests that JML can have its use fostered when people get in touch with the language syntax, maybe by means of classes for presenting JML in undergraduate courses.

Overall, this survey corroborates with the results of our experiment: CONTRACTJDOC is intermediate between Javadoc and JML, being closer to Javadoc with respect to comprehensibility. Furthermore, the results highlight Javadoc as the most understandable approach concerning the behavior of a documented interface.

3.3.5 Threats to validity

Concerning internal validity, all material for the survey study is available only in English, therefore, the experience of the Subject with English can have affected their comprehensibility of the behavior of the provided interface.

The order in which we display the documented interfaces on the survey form, the questions used for evaluating comprehensibility, the kind of questions used, and the absence of opened questions can threat the construct validity. For dealing with these threats we perform a pilot before applying the survey and used the results from the pilot to improve the survey structure.

Regarding external validity, even receiving a satisfactory number of answers – 142 – to our survey, the results are not representative of the community of Java developers. In addition, we used only one data structure interface: `Stack`, for asking about the comprehensibility of the interface behavior. In other domains with more complex structures, the results can vary considerably.

3.3.6 Answer to the research question

From our results, we made the following observations:

- **Q5.** In the developer's opinion, which approach, among Javadoc, CONTRACTJDOC, and JML, is the most effective in communicating a routine's behavior?

  According to the statistical tests performed, Javadoc is the most understandable documentation approach, and CONTRACTJDOC is intermediate between JML and Javadoc, being closer to Javadoc.

Chapter 4

ContractSuggestor

Javadoc comments are semi-structured natural-language text blocks that specify a code element (a Java method, field, class, or package) [49]. Javadoc comments use tags to structure the documentation. For instance, the *@param* tag marks the specification of a formal parameter and *@return* marks the specification of the value returned by a method. However, those comments are not runtime checkable as DBC contracts are.

In this context, we present ContractSuggestor, our approach for automatically suggesting contracts from natural language tagged comments. Currently, ContractSuggestor suggests pre- and postconditions for methods (methods is a short for methods and constructors) by applying machine learning techniques [111] for classifying those comments.

We now describe ContractSuggestor and explore its features. First, we provide the theoretical background needed for understanding this chapter (Section 4.1). Then, we show the approach and describe its infrastructure (Section 4.2). Next, Section 4.3 provides an evaluation of the machine learning algorithms we used and some tips for guiding extensions to the present work. Finally, we highlight the main limitations of our contract suggestor (Section 4.4).

4.1 Machine Learning

The term machine learning refers to the automated detection of meaningful patterns in data. In the past couple of decades, it has become a common tool in almost any task that requires information extraction from large data sets [111]. Learning is a very wide domain. Consequently, the field of machine learning has branched into several subfields dealing with different types of learning tasks [111; 88]. The term "learning" is closely related to generalization. The goal of statistical learning is to find patterns in data that will generalize well to new, unobserved data [88].

In addition, when performing a learning task, we can have a binary or multiclass classification (see Section 4.1.2).

4.1.1 Supervised Learning

For this work, we consider Supervised Learning [88] because text categorization is a supervised learning problem where the task is to assign a given text document one or more predefined categories [107].

Supervised machine learning algorithms may apply what has been learned in the past to new data using labeled examples to predict future events [111]. Starting from the analysis of a known training dataset, the learning algorithm produces an inferred function to make predictions about the output values. The system is able to provide targets for any new input after sufficient training. The learning algorithm can also compare its output with the correct, intended output and find errors in order to modify the model accordingly.

With respect to algorithms for supervised learning, we explain three from those available at Scikit-learn Python [93]: AdaBoost, Multi-layer Perceptron, and Passive-Aggressive.

4.1.1.1 AdaBoost

Boosting is an ensemble technique that attempts to create a strong classifier from a number of weak classifiers [88]. This is done by building a model from the training data, then creating a second model that attempts to correct the errors from the first model. Models are added until the training set is predicted perfectly or a maximum number of models are added.

AdaBoost is a shorthand for Adaptive Boosting. AdaBoost [88; 111] is used with short decision trees. After the first tree is created, the performance of the tree on each training instance is used to weight how much attention the next tree that is created should pay attention to each training instance. Training data that is hard to predict is given more weight, whereas easy to predict instances are given less weight. Models are created sequentially one after the other, each updating the weights on the training instances that affect the learning performed by the next tree in the sequence. After all the trees are built, predictions are made for new data, and the performance of each tree is weighted by how accurate it was on training data. In Figure 4.1 we illustrate how AdaBoost works for classifying two-labeled comments. A first model is created and a prediction occurs, then, new models are created in order to correct the errors from the first model. In the end, all weak classifiers are combined in a single model.

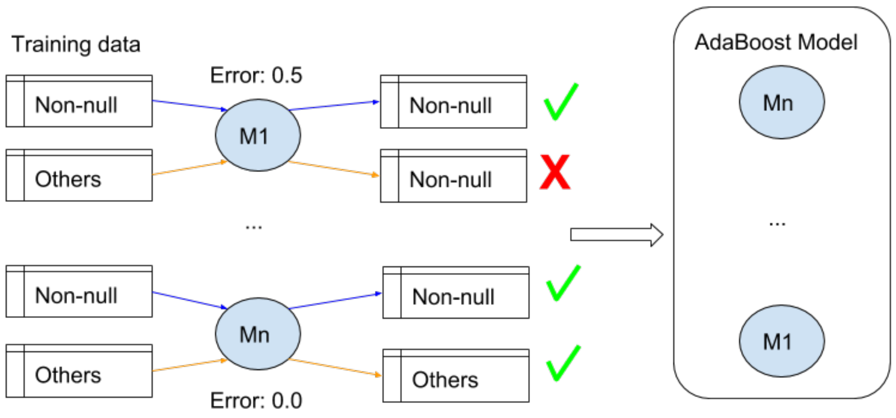


Figure 4.1: AdaBoost for text classification. The first model correctly classifies the document as non-null, but incorrectly classify the others document. Then, new models are created until reach a correct classification for all documents.

4.1.1.2 Multi-layer Perceptron

Instead of using weak learners that are improved in each run of the learning algorithm, an artificial neural network is a model of computation inspired by the structure of neural networks in the brain. Artificial neural networks are formal computation constructs that are modeled after this computation paradigm [111].

The idea behind neural networks is that many neurons can be joined together by communication links to carry out complex computations. It is common to describe the structure of a neural network as a graph whose nodes are the neurons and each (directed) edge in the graph links the output of some neuron to the input of another neuron. We will restrict our attention to feedforward network structures in which the underlying graph does not contain cycles [111].

A multilayer perceptron (MLP) is a class of feedforward artificial neural network. An

MLP consists of at least three layers of nodes. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. It can distinguish data that is not linearly separable [93].

A MLP typically consists of an input layer (i.e., spectral data or principal components), one or more hidden layers, where the real processing is performed via a system of weighted 'connections', and an output layer (prediction), where the answer is output [87]. They function by linking the input neurons to output neurons, through the connections (weights). For the purpose of text classification, the output layer presents the predicted category for each text document.

4.1.1.3 Passive-Aggressive

Online learning takes place in a sequence of consecutive rounds. On each online round, the learner first receives an instance. Then, the learner is required to predict a label. At the end of the round, the learner obtains the correct label. Finally, the learner uses this information to improve his future predictions [111]. The Passive-Aggressive procedure for training is similar to AdaBoost (Section 4.1.1.1): the algorithm is passive when a correct classification occurs and aggressive when it needs to update the rule used for classification.

The passive-aggressive algorithms are a family of algorithms for large-scale learning [93],[25]. Crammer et al. [25] presented the online Passive-Aggressive (PA) algorithm, which is as fast as but more accurate than Perceptron [111]. It updates the model to have a low loss on the new sample, as well as to ensure that the new model is close to the current one.

4.1.2 Type of classification

Concerning types of classification, there is binary and multiclass classification [111].

Binary or binomial classification is the task of classifying the elements of a given set into two groups (predicting which group each one belongs to) on the basis of a classification rule [111]. For example, we may want to distinguish nullable comments from non-nullable.

Multiclass categorization is the problem of classifying instances into one of several possible target classes [111]. For instance, whether we consider more than one property of interest for classifying the comments, we will need to use a multiclass algorithm for that.

Text Classification (TC) is a supervised learning problem where the task is to assign a given text document to one or more predefined categories [107]. In this context, feature selection is the task of reducing the dimensionality of feature space by identifying informative features and its primary goals are improving classification effectiveness, computational efficiency, or both [107]. This is needed because the performance of a classifier is affected by the employed feature selection mechanism. One of the mechanisms commonly used nowadays for features selection during text classification is TF-IDF approach (stands for Term Frequency-Inverse Document Frequency) [93].

TF-IDF is a numerical statistic that is intended to reflect how important a word is to a document in a collection [99]. It is often used as a weighting factor in searches of information retrieval, text mining, and user modeling. For an example, consider we have a document base composed of two files:

File 1: @param bag the bag to decorate, must not be null.

File 2: @throws IllegalArgumentException if bag is null.

The TF-IDF for the word *not* is calculated as follows:

$$\begin{aligned} &TF-IDF('not') = \\ &\quad = TF('not') * IDF('not', Base) \\ &\quad = 1/10 * log(2/1) \\ &\quad = 0.1 * 0.301 \\ &TF-IDF('not') = 0.0301 \end{aligned} \tag{4.1}$$

Moreover, lemmatization [70] refers to do things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the lemma. For instance, the words: [work, working, worked, workers] represent a single lemma: *work*.

4.2 CONTRACTSUGGESTOR: an Approach for Suggesting Contracts

Motivated by the case study performed with CONTRACTJDOC (see Section 3.1) and works such as @tComment [118], we developed an approach for processing texts in natural language and for suggesting contracts for code properties based on those texts. CONTRACTSUGGESTOR provides an approach for automatically suggesting contracts by analyzing natural language tagged-comments.

This effort tries to reduce the annotation burden [34] – one of the points developers highlight as being a blocking for using the DBC approach – by enabling developers to use contracts without asking them to learn a new syntax or even to use an approach such as CONTRACTJDOC.

When performing the CONTRACTJDOC case study, we noticed that some comments are commonly used for describing properties in each evaluated system. Therefore, we decided to create an approach for automatically suggesting contracts for those properties by applying a supervised machine learning algorithm. In the systems evaluated in CONTRACTJDOC's case study (Section 3.1), comments such as "noteBase - represents the basic note letter, must not be null", "noteLength - must not be null", or "@return A non-null value" are used for declaring some parameter or return value as non-null; and comments such as "timeLimit time limit for this time limit exit strategy, must be >= 1" declares relational (greater than/equal, less than/equal) properties; these kinds of comments are common on the evaluated systems.

Furthermore, as highlighted by Meyer, Kogtenkov, and Stapf [78], we want to apply operations to objects through the associated non-void references and in a language that does not guarantee void safety – void safety is the avoidance of void calls –, the run-time effect of a void call is either to crash the program or, if the language supports exception handling, to trigger an exception. So, to provide some way of avoiding a void reference in a language such as Java (that does not guarantee void safety) may improve software quality and reduce program crashes. Thus, CONTRACTSUGGESTOR is an approach for generating contracts for code properties, currently applied to void safety and relational properties.

Since we chose a supervised approach, we needed to create a dataset for training the machine learning algorithms for classification. After training, the algorithms are able to classify new comments based on their knowledge base and we can evaluate each algorithm based on machine learning metrics [111]: precision, recall, and F1-score.

With the best algorithm in hands, CONTRACTSUGGESTOR generates contracts for each property (currently non-null and relational properties) we consider and those contracts can be automatically checked by a tool such as JMLOK2 [84]. Figure 4.2 presents an overview of our strategy. First, the project received as parameter pass through the tagged-comments extractor. Then, the trained machine learning algorithm is applied for classifying each comment instance (after the training process with the dataset manually produced). Finally, a contract generator will generate contracts for the property desired (in order to not modify the sources of the projects, we are using AspectJ [64] aspects as contracts).

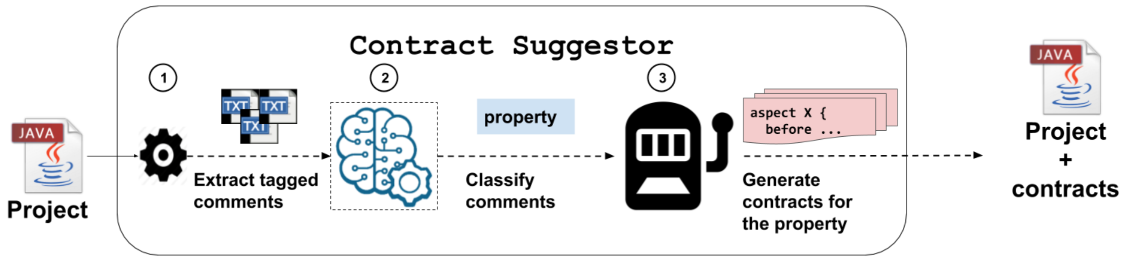


Figure 4.2: ContractSuggestor infrastructure. First, the project received as parameter pass through the tagged-comments extractor. Then, the trained machine learning algorithm is applied for classifying each comment instance (after the training process with the dataset manually produced). Finally, a contract generator will generate contracts for the property desired (in order to not modify the sources of the projects, we are using AspectJ aspects as contracts).

In the next sections, we detail each part of CONTRACTSUGGESTOR approach. First, we describe the process of extracting tagged-comments (Section 4.2.1). Then, we detail the construction of the dataset for each property considered in our approach (Section 4.2.2). Next, Section 4.2.3 explains the process of training the machine learning algorithms considered. After that, we discuss the contracts generated by CONTRACTSUGGESTOR (Sections 4.2.4). Finally, Sections 4.3 and 4.4 present how we evaluated each machine learning algorithm and the main limitations of the current approaches, respectively.

4.2.1 Extracting Tagged-comments

As the first step in our approach, we need to extract all tagged-comments from a given project. For this purpose, we created a Python [31] script – *extract_comments* – for reading the project files, collecting tagged comments and generating one file for each tagged-comment. We provide all scripts and data from this work in our companion website [85].

For illustrating how the script works, consider the following source code:

Source Code 4.1: Example of tagged-comments extraction.

```java
/**
* Constructor that wraps (not copies).
*
* @param bag  the bag to decorate, must not be null
* @throws IllegalArgumentException if bag is null
*/
protected AbstractBagDecorator(Bag bag) {
    super(bag);
}
```

Supposing this is the only file with tagged-comments in our Java project, *extract_comments* will create two files: one file for the tagged-comment – “@param bag the bag to decorate, must not be null” and one for the comment related to the “@throws” tag. Figure 4.3 shows the result of applying the script to Source Code 4.1.

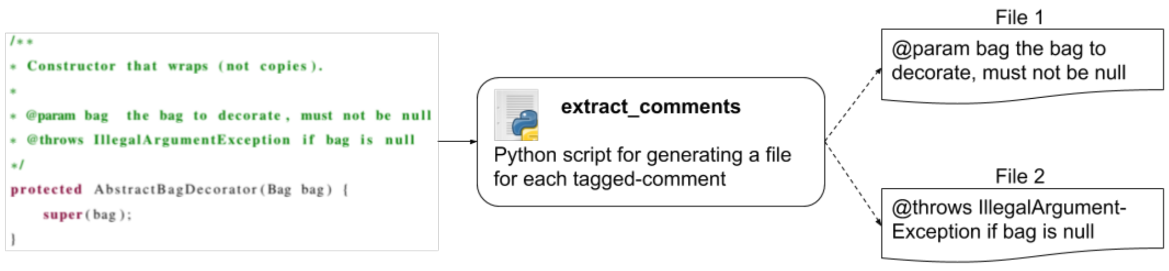

```java
/**
* Constructor that wraps (not copies).
*
* @param bag  the bag to decorate, must not be null
* @throws IllegalArgumentException if bag is null
*/
protected AbstractBagDecorator(Bag bag) {
    super(bag);
}
```
extract_comments
Python script for generating a file for each tagged-comment
File 1
@param bag the bag to decorate, must not be null
File 2
@throws IllegalArgument-Exception if bag is null

Figure 4.3: Extract comments process. After receiving a project as input, the extractor will create a file for each tagged comment.

The result of running *extract_comments* can be used as an input for a machine learning algorithm (as occurs in the final version of ContractSuggestor) or as an input to a manual analysis of each tagged-comment for creating a dataset for classification purposes (Section 4.2.2).

4.2.2 Constructing a Dataset

As we are using supervised machine learning in our contract suggestor, we needed to create a dataset for each property considered. The process for creating the dataset is the following: first, we apply the extractor of comments in order to collect all tagged-comments from a given project; then, we manually open and read the content of each file and decide in which folder the file must be put: in a binary classification - with property (for those tagged-comments that presents the desired property) or without property (for those comments not related to the desired property). Figure 4.4 illustrates this process. The result of this process is a dataset of tagged instances that serves as input to the training and testing processes of a machine learning algorithm being considered (see Section 3.1).

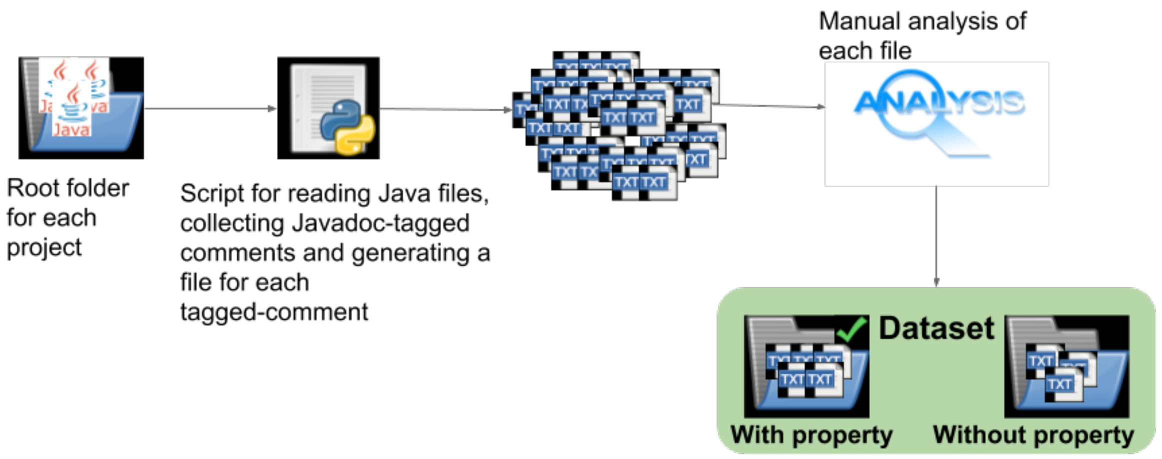

Root folder for each project
Script for reading Java files, collecting Javadoc-tagged comments and generating a file for each tagged-comment
Manual analysis of each file
ANALYSIS
Dataset
With property
Without property

Figure 4.4: Dataset construction. After applying the comments extractor, we manually read the content of each file and decide the folder that the file belongs to.

For the Source Code 4.1, two files are created: file 1 for the @*param* content and file 2 for the @*throws* content. By manually analyzing each file, the following classification is performed: file 1 goes to the non-null folder (the folder for those comments related to the desired property) and file 2 goes to the others folder (the folder for comments not related to the desired property).

As we applied our approach for two properties: non-null and relational, we proceed by presenting the features of each dataset created. First, we present the non-null datasets (Section 4.2.2.1); then, we present the relational dataset (Section 4.2.2.2). In each section we list information about the systems we used for creating the dataset and the final size of the dataset.

4.2.2.1 Non-null dataset

As null dereferencing is considered a critical problem, the inventor of null reference calls it a billion-dollar mistake [55], the first property we tried to automatically identify and generate contracts for is non-null – an approach for avoiding null references based on natural language comments.

For non-null, the Non-null dataset is composed of 21 open-source Java projects. This dataset contains all comment instances from the systems used in CONTRACTJDOC case study (Section 3.1), from the systems considered into @tComment approach [118], and the comments from eight new open-source Java systems not considered in those studies. Table 4.1 summarizes information about these projects.

Table 4.1: Experimental units' summary. Column Source presents a reference to the source code location for each system. Column Description shows a brief description of each system. And column LOC shows the code size of each experimental unit.

Experimental Unit	Source	Description	LOC
ABC-Music-Player	[86]	A music player and parser	1,973
Apache Collections	[39]	Collection library and utilities	26,323
Dishvelled	[53]	Libraries for user interface components and supporting code	110,577
GlazedLists	[56]	List transformations in Java	5,134
Jenerics	[104]	Library templates and tools for Java	2,538
JFreeChart	[29]	Chart creator	72,490
JodaTime	[60]	Date and time library	25,916
Log4J	[41]	Logging service	20,987
Lucene	[42]	Text search engine	48,201
OOP Aufgabe3	[97]	Polygons manipulation	353
SimpleShop	[92]	A simple shop	472
Webprotégé	[71]	Ontology development environment	74,742
Xalan	[43]	XML transformations	178,549
Betwixt	[58]	Mechanism for mapping beans to XML	23,754
Cobertura	[119]	Java code coverage tool	10,854
IText	[57]	PDF generation and manipulation for Java and .NET	137,507
JodaMoney	[121]	Java library to represent monetary amounts	8,467
Maths	[40]	Library of mathematics and statistics	209,653
OpenJML	[28]	Program verification tool for Java	782,657
SmartHome	[120]	Framework to design a Smart Home solution	118,005
SWT	[38]	Widget toolkit for Java	122,074

The manual classification of all tagged-comments from those 21 projects (See Table 4.1) results in a dataset of 134,246 comment instances. From those, 7,241 are instances related to the non-null property (folder non-null) and 127,005 are not related to the property (folder others).

After creating the Non-null dataset, we use it for training and testing all machine learning algorithms for supervised learning from Scikit-learn [93] (details in Section 4.2.3.1). With the machine learning algorithms trained and tested, we applied the top-three (in terms of precision, recall, and F1-score) to new open-source Java projects available in GitHub in order to check their performance to new data (details in Section 4.2.3.1).

The dataset is available online in our companion website [85] as well as Python scripts for downloading all considered systems for purposes of replication of the current datasets.

4.2.2.2 Relational dataset

For relational properties (greater than, greater than or equal, less than, and less than or equal), we selected a subset of the systems considered into the Non-null dataset (Section 4.2.2.1) composed of six systems (see Table 4.2). After manually classifying all comments available, the sizes for the dataset of relational properties are: greater than (20), greater than or equal (31), less than (10), less than or equal (13), and others (1,734) – others represent those comments that are not related to any relational property neither non-null property.

Table 4.2: Experimental units' summary. Column Source presents a reference to the source code location for each system. Column Description shows a brief description of each system. Column LOC shows the code size of each experimental unit.

Experimental Unit	Source	Description	LOC
ABC-Music-Player	[86]	A music player and parser	1,973
Cobertura	[119]	Java code coverage tool	10,854
JodaMoney	[121]	Java library to represent monetary amounts	8,467
OOP Aufgabe3	[97]	Polygons manipulation	353
SimpleShop	[92]	A simple shop	472
Webprotégé	[71]	Ontology development environment	74,742

The dataset and Python scripts for downloading all considered systems are also available online in our companion website [85] for purposes of replication of the current dataset.

4.2.3 Training Machine Learning algorithms

The next step in CONTRACTSUGGESTOR approach is to set up the best supervised machine learning algorithm for the purpose of classifying natural language tagged-comments either in a binary or multiclass fashion.

The input to a learning algorithm is training data, representing experience, and the output is some expertise, which usually takes the form of another computer program that can perform some task [111]. In addition, we need a test set for verifying how much the machine learning algorithm is able to generalize based on the training process. Thus, we need to split our dataset into at least two sets: training and testing.

There are some rules of thumbs for this splitting [103]. One of them, suggests the 80/20 split, 80% for training and 20% for testing purposes. We started splitting our dataset with these proportions and evaluated the machine learning metrics (precision, recall, and F1-score) in order to establish the best proportional size for each set.

After performing variations on the sizes for the sets, we found 3/4 and 1/4, respectively for training and testing, as being the suitable sizes for the sets for the problem of text classification – we achieve these sizes by analyzing the machine learning metrics (precision, recall, and F1-score) of the algorithms. Therefore, for all datasets considered in this work, we split the data as follows: 75% for training and 25% for testing.

Every classification problem needs to deal with features for performing the desired classification. We are performing a classification that involves text. We use TF-IDF for extracting features from our dataset in order to enable the text classification.

In addition, for grammatical and personal reasons, natural language comments use different forms of a word, such as *organize*, *organizes*, and *organizing*. Additionally, there are families of derivationally related words with similar meanings. In many situations, it seems as if it would be useful for a search for one of these words to return comments that contain another word in the set [70]. For dealing with these problems, our script for processing the document context applies lemmatization to the extracted comments before running the machine learning algorithms for classification.

After defining the sizes for training and testing sets and the approach for extracting features and dealing with natural language variations for writing words with the same semantic, we need to explain how we choose the machine learning algorithms for classifying the comments. In the next sections, we explain how to choose the algorithm we used for classifying comments for each property.

4.2.3.1 Non-null property

We performed a comparison of the algorithms available for supervised learning in Scikit-learn package [93] (see Appendix A for details) and chose the top-three algorithms in terms of accuracy, precision, recall, and F1-score with respect to the Non-null dataset (Section 4.2.2.1). These algorithms represent the following classes of supervised learning algorithms: while AdaBoost is an Ensemble Method, Multi-layer Perceptron is an artificial Neural network, Passive-Aggressive is a generalized linear model.

In order to compare the algorithms' results, we investigated the relationship between the parameters of each algorithm. All of them have a parameter that allows us to control the number of iterations over the training data: n_estimators for AdaBoost; and max_iter for Multi-layer Perceptron and Passive-Aggressive. After performing Grid Search parameter tuning [111], we identified the following best values: n_estimators = 100 (AdaBoost), max_iter = 200 (Multi-layer Perceptron), and max_iter = 1,000 (Passive Aggressive). We summarize in Table 4.3 the machine learning metrics achieved for the algorithms after parameter tuning [111].

Table 4.3: Machine Learning Algorithms. For each algorithm and property classified, we present precision, recall, F1-score, and Accuracy.

ML Algorithm	**Property**	**precision**	**recall**	**F1-score**	**Accuracy**
AdaBoost	non-null	1.00	0.99	1.00	0.9995
	others	1.00	1.00	1.00	
Multi-layer Perceptron	non-null	1.00	1.00	1.00	0.9994
	others	1.00	1.00	1.00	
Passive-Aggressive	non-null	0.99	1.00	1.00	0.9992
	others	1.00	1.00	1.00	

Considering precision, recall, F1-score and accuracy, AdaBoost is the best algorithm for classifying natural language comments between non-null and others (see Table 4.3). According to this result, AdaBoost is more likely to correct classify a comment with respect to void safety property (distinguish a comment between non-null and not related to non-null) – the approach is likely to produce more true positives. Based on this, we reach our first guideline:

we may use AdaBoost classifier for binary text classification problems whenever you have a properly created dataset – manually validated.

4.2.3.2 Relational property

For relational properties, Table 4.4 summarizes the metrics for evaluating each one of the chosen algorithms. Different of what happened to the binary classification, AdaBoost is not the best algorithm; Passive-Aggressive is the best one for all considered metrics. Despite using the multiclass classification version for each algorithm [111; 134], AdaBoost was outperformed by the Passive-Aggressive classifier. Thus, we achieve our second guideline: When performing multiclass classification (for the context of text classification), the strategy from Passive-Aggressive can outperform boosting: be passive for a right classification and aggressive with the errors.

Table 4.4: Machine Learning Algorithms. For each algorithm and property classified, we present precision, recall, F1-score, and Accuracy.

ML Algorithm	Property	precision	recall	F1-score	Accuracy
AdaBoost	greater than	1.00	0.44	0.62	0.9801
	greater than or equal	0.71	0.62	0.67	
	less than	0.50	1.00	0.67	
	less than equal	1.00	1.00	1.00	
	others	0.99	0.99	0.99	
Multi-layer Perceptron	greater than	1.00	0.33	0.50	0.9801
	greater than or equal	0.50	1.00	0.70	
	less than	0.50	1.00	0.67	
	less than equal	0.67	1.00	0.80	
	others	1.00	0.99	1.00	
Passive-Aggressive	greater than	1.00	0.56	0.71	0.9889
	greater than or equal	0.78	0.88	0.82	
	less than	0.50	1.00	0.67	
	less than equal	1.00	1.00	1.00	
	others	1.00	1.00	1.00	

4.2.4 Generating Contracts

The last step from CONTRACTSUGGESTOR is the contract generation. For generating contracts – pre- and postconditions for each property considered – we use AspectJ [64]. The decision for AspectJ occurred after the lack of success in using some JML-related approach. Then, we decided to use AspectJ for the following reasons: first, we do not want to modify the source code of the projects being evaluated (and in some cases, the source code may not be available); second, we need an updated compiler for supporting new Java constructs (JML default compilers are not updated with Java features); finally, the advice mechanism from AspectJ seems suitable for generating pre- and postconditions in a way similar to JML contracts [100].

AspectJ supports three kinds of contract-like constructs. AspectJ divides its constructs into that which runs before a specific point – similar to a precondition; that which runs after a specific point – similar to a postcondition; and that which runs in place of (or "around") a point. For our current purposes, only two kinds are useful: the construct which runs before a specific point (construct *before*) – for generating preconditions, and the construct which runs after a point – for postconditions (construct *after*).

For illustrating the contract generation step, we use the example in Source Code 4.1. After passing through the comments extraction and machine learning application, the @param comment is classified as having the non-null property. Then, starts the contract generation phase: first, the script identifies the kind of contract that needs to be generated (pre- or postcondition) – for this example, a precondition (*before* advice); next, the script identifies the method for which to apply the aspect; finally, the aspect is created to check the desired property. Source Code 4.2 shows the aspect generated for the Source Code 4.1.

Source Code 4.2: Aspect representing a precondition generated to the Source Code 4.1.

```
public aspect AbstractBagDecoratorAJ {
 before(Bag bag):
   execution (* AbstractBagDecorator.AbstractBagDecorator(..)) && args(bag) {
         assert bag != null;
 }
}
```

This aspect, as well as a precondition, will be checked before the execution of `AbstractBagDecorator`'s constructor for checking whether the `bag` parameter is different of *null*; the assert into the aspect will pass when the `bag` is not null and will fail otherwise.

For a postcondition, consider the Source Code 4.3. This code excerpt has a tagged-comment declaring the return will be always greater than or equal to three. By applying CONTRACTSUGGESTOR approach, one file is generated for the @return tagged-comment. Then, the machine learning algorithm identifies this as being a greater than or equal case. Finally, the aspect presented in Source Code 4.4 is generated.

Source Code 4.3: Example for generating a postcondition.

```
public interface Polygon{
  /**
   * Returns the number of edges of a polygon as integer.
   *
   * @return Integer the number of edges. Is always >= 3
   */
  // edges >= 3
  int edges();
}
```

Source Code 4.4: Example of postcondition generated.

```
public aspect PolygonAJ{
 after() returning(int o): call(* Polygon.edge()){
   assert o >= 3;
 }
}
```

The after returning aspect will check if the value returned by `edge` method is greater than or equal to three. As a postcondition, the check is performed after the execution of a given method.

4.3 Evaluating Contract Generation

In this section, we evaluate the contracts generated by CONTRACTSUGGESTOR approach. The aim of our evaluation is to determine the number of true positives (a comment that is manually classified as non-null and for which the machine learning algorithm also classify it as non-null) generated by the approach when facing systems not available in the datasets.

Although machine learning metrics such as precision, recall, and f1-score are largely used for evaluating the algorithms for classification, in our context, we also need to evaluate the contracts generated (aspects for representing pre- and postconditions) since CONTRACTSUGGESTOR aims to automatically suggest contracts for Java programs with tagged-comments related to the properties discussed in previous sections (Section 4.2.4).

In the context of contract-based programming, a contract wrongly generated (called False positive in machine learning literature [111]) is critical: a client that should be able to get the needed information from a supplier does not receive that information, or a supplier should be working properly (returning the expected value) but having a wrong contract could be considered as not working, not returning the expected value. So, we are interested in checking all generated contracts for avoiding this problem. For each system, we used for evaluating CONTRACTSUGGESTOR, we present in Table 4.5 the total amount of contracts generated, and from those, the number of correct (true positives – manually classified as non-null and classified as non-null by the machine learning algorithm), the number of false positives, and the false positive rate – as a measure of quality for the approach in a given system.

$$FPR = \frac{\#FalsePositives}{\#Contracts} \tag{4.2}$$

Table 4.5: Evaluating contracts generated by CONTRACTSUGGESTOR. For each system, we present, grouped by dataset, the number of contracts generated, the number of correct contracts, the number of false positives, and the false positive rate (as defined by Equation 4.2).

Non-null dataset				
System	**#Contracts**	**#Correct**	**#False Positive**	**FPR**
Apache Commons-lang	558	545	13	0.023
Apache Dubbo	9	9	0	0.000
Apache PDFBox	25	25	0	0.000
Curity	1	0	1	1.000
Cyclops Group	1	1	0	0.000
Databus	5	5	0	0.000
Design Patters in Java	3	3	0	0.000
Java lang	5	4	1	0.200
Java util	20	15	5	0.250
Mozilla Zest	1	1	0	0.000
Riak	1	1	0	0.000

Relational dataset				
System	**#Contracts**	**#Correct**	**#False Positive**	**FPR**
Apache Collections	24	4	20	0.833
Betwixt	7	0	7	1.000
PdfBox	15	0	15	1.000

Regarding Non-null dataset for only one system – `Curity` – the FPR is high 1.0; for all other systems the FPR is less than 0.3, indicating the majority of the generated contracts are correct and can be maintained in the systems. Those results for the non-null property are explained by the amount of non-null instances in the dataset (more than 7K instances) that represent several ways of declaring (in natural language constructs) that a parameter or a method return value must be non-null.

On the other hand, the FPR for the dataset Relational is high for all systems (greater than 0.8). This can be explained the size of each class (relational property being considered – greater than, greater than or equal, less than, and less than or equal): the class with more comment instances – greater than or equal – has only 31 instances. As natural language provides several ways of declaring the same property, we need to increase the sets for achieving a low FPR. Explaining what happened in each system, for Apache Collections, CONTRACTSUGGESTOR approach generates 24 contracts, from those, four are correct, and two are misclassified – the contract should be a greater than but it was classified as greater than or equal. With respect to Betwixt, all generated contracts are incorrect. For PdfBox, the approach generates 15 contracts; from those, 14 are incorrect and one was misclassified – the contract should be a greater than but it was classified as greater than or equal.

4.4 Limitations

The current approach only considers tagged-comments, this limits our contracts to be pre- and postconditions. In addition, we miss all untagged information provided in method comments. A way of reducing this limitation is to improve *extract_comments* for collecting all comments in the Java source code that are delimited by the /** ... */ delimiters and update the datasets for considering those comments. Then, the approach for generating contracts will need to use the information from comments for generating contract invariants.

Although the datasets, mainly the dataset for the non-null property, are relatively large, they do not represent all possible ways of writing about the considered properties; therefore, even the best algorithm we used produces both false positives and false negatives. For contract generation context, a false positive is critical: correct clients can have some functionality negated because an incorrect contract that is automatically generated. On the other hand, a high number of false negatives may turn CONTRACTSUGGESTOR approach useless.

Chapter 5

Evaluating Contracts

In this chapter, we first present our evaluation of CONTRACTSUGGESTOR (Section 5.1); then we present our leverage of JMLOK2 [84] for the C#/Code Contracts context (Section 5.2.2); finally, we present the case study we perform with both tools – CONTRACTOK and JMLOK2 – concerning automatic detection and manual classification of nonconformances in contract-based programs (Section 5.3).

5.1 Checking Contracts by CONTRACTSUGGESTOR

We run JMLOK2 [84] over the contracts automatically generated in order to check if the systems are in conformance with the contracts inferred from natural language tagged-comments. First, we present the study definition (Section 5.1.1) and planning (Section 5.1.2). Then, Sections 5.1.3 and 5.1.4 present and discuss the results, respectively. Finally, Section 5.1.5 describes the threats to validity. The evaluation follow three main steps: (1) we run the contract generator over each experimental unit, (2) manually check each generated contract, and (3) we run JMLOK2 over the system in order to detect inconsistencies between source code and contracts – potential nonconformances, because the inconsistencies may not be real bugs but outdated code commentary.

5.1.1 Definition

The goal of this case study is to analyze the contracts generated by CONTRACTSUGGESTOR for the purpose of evaluation with respect to detection of nonconformances from the point of view of developers in the context of programming. In particular, we address the following research question:

Q1. How many nonconformances is JMLOK2 able to detect in systems with contracts for non-null and in systems with contracts for relational generated by CONTRACTSUGGESTOR approach?

We measure the number of detected nonconformances (#NCs, henceforth) in each system within a given test generation time limit.

5.1.2 Planning

In this section, we present the design of this case study. First, we show the units used. Then, we present the experimental procedure.

Experimental Units

The case study is performed on systems not used for validating the machine learning algorithms for non-null and relational properties: `Apache Collections`,[1] `Apache Commons-lang`,[2] `Apache PDFBox`,[3] and `JodaMoney`.[4] These projects sum up to 240 KLOC and 978 contracts. These projects were the only projects that could be checked by JMLOK2.

The companion website [85] groups complete descriptions of all considered systems, which are listed in Table 5.1, all metrics considered in this study, and a replication package of this study.

Study procedure

The study consists of three main steps (see Figure 5.1): (1) run the tools over each contract-based program in a given time limit (the time limit is the time used for tests gen-

[1]`https://archive.apache.org/dist/commons/collections/source/commons-collections-3.2.1-src.tar.gz`
[2]`https://github.com/apache/commons-lang`
[3]`https://pdfbox.apache.org/`
[4]`https://github.com/JodaOrg/joda-money`

Table 5.1: Experimental units' summary. Column LOC shows the code size of each experimental unit. Column #CC presents the total of contracts generated for each experimental unit. Column Property lists the property that is checked with the contracts.

Experimental Unit	LOC	#CC	Property
Apache Collections	26,321	6	Relational
Apache Commons-lang	76,860	545	Non-null
Apache PDFBox	128,399	25	Non-null
JodaMoney	8,467	402	Non-null
Total	**240,047**	**978**	

eration [91]), and (2) manually classify each detected nonconformance in terms of its likely cause.

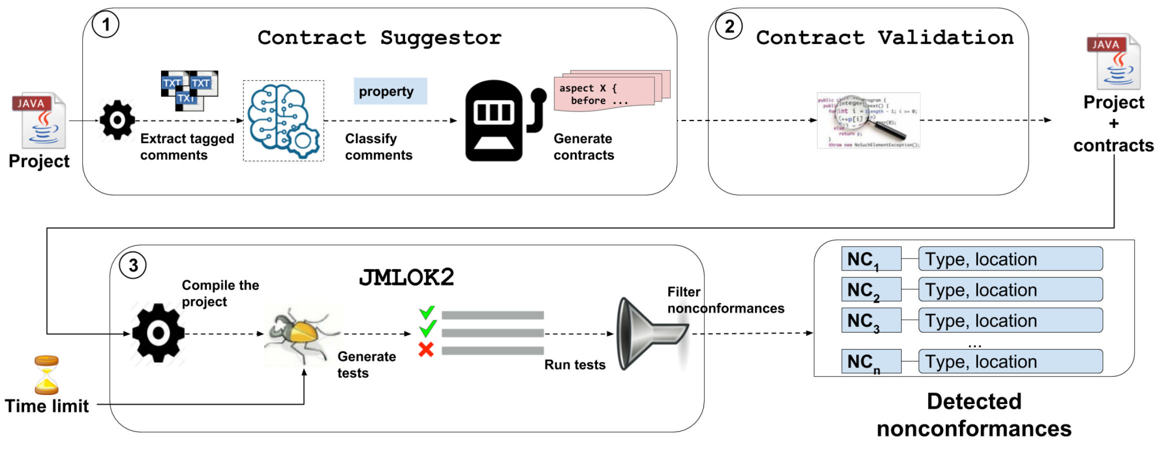


Figure 5.1: Steps performed in this study. First, we use our approach for suggesting contracts based on natural language comments. Then, we perform the manual validation of each contract. Finally, we run JMLOK2 over each system in order to detect potential nonconformances.

We performed the case study on a 3.6 GHz core i7 with 16 GB RAM running Windows 8.1 Enterprise, backed by Java 1.8 update 162, AspectJ compiler 1.9.0.RC4. Since Randoop [91] requires a time limit for test generation – the time after which the generation process stops –, we used 600s, running the tool 10 times.

5.1.3 Results

JMLOK2 detects 6 nonconformances in the evaluated projects (see Table 5.1): `Apache Collections` (1), `Apache Commons-lang` (3), and `JodaMoney` (2). In `Apache PDFBox`, JMLOK2 was not able to detect nonconformances with the configuration used (Section 5.3.2). Concerning nonconformance types, all nonconformances detected are *precondition* problems (see Table 5.2).

Table 5.2: For each system, we present the class and method in which the problem has occurred. Column Type shows the nonconformance types.

System	Class	Method	Type	Total
Apache Collections	BoundedBuffer	decorate	precondition	1
Apache Commons-lang	FieldUtils	getDeclaredField	precondition	3
		getField	precondition	
	JavaVersion	atLeast	precondition	
JodaMoney	BigMoney	checkCurrencyEqual	precondition	2
	Money	checkNotNull	precondition	
Total				**6**

5.1.4 Discussion

We proceed with the discussion of our research question. JMLOK2 detected a total of 6 nonconformances: 5 related to non-null and one related to the relational property. The only type of nonconformance detected was the *precondition*. These numbers indicate that most violations occur at the entry of operations; internally the code of the systems do not consider the restrictions that are into the comments. We submitted all nonconformances for the systems' developers but they did not answer our contact.

In `Apache Commons-lang`, even though the called methods have the natural language comments such as: "the {@link Class} to reflect, must not be {@code null}", the callers do not respect this and call them by passing a null value as a parameter.

With respect to the nonconformance detected into `Apache Collections`, method `decorate` from `BoundedBuffer` is called with 0 for the parameter `maximumSize` by the method `boundedBuffer` from `BufferUtils`. Method `decorate` has the following contract: *maximumSize* >= *1;* generated from the natural language tagged-comment:

"the maximum size, must be size one or greater".

For `JodaMoney` unit, however, we believe the contracts automatically generated for classes `BigMoney` and `Money` are not sufficient for satisfying the requirement declared in the Javadoc comment: "the monetary values to total, not empty, no null elements, not null". Our approach is only able to generate a partial contract for this comment; this is a limitation of our approach. A way of solving the problem would be to improve the contract by adding the following statements:

```
for(BigMoneyProvider b: monies) {
        assert b != null;
}
```

Then, the excerpt "no null elements, not null" would be checked and the test cases generated into JMLOK2 would be classified as *meaningless* test cases because they would violate the preconditions of the method directly.

5.1.5 Threats to validity

This study has some limitations; next, we describe some threats to the validity of our evaluation.

Internal validity

The randomness promoted by the use of an automatic test generator (Randoop [91]) by JMLOK2 is an internal threat to the validity of our study; so, we ran each system 10 times for confidence. In each execution, tests are generated independently of the previous run, which may show different contexts revealing the same nonconformances.

External validity

A Randoop-based approach lacks repeatability, in terms of machine setting or operating system. Also, even though we diversified our choice of systems, varying in code size and contracts, generalizing the obtained results is almost impossible.

5.2 Verifying Nonconformances in General

In this section, we discuss nonconformances in a general perspective. We first provide the theoretical foundation for understanding Code Contracts; then, we present our approach for conformance checking of C#/Code Contracts systems. Finally, Section 5.3 presents our case study with JMLOK2 and CONTRACTOK over 24 open-source systems.

5.2.1 Code Contracts

Code Contracts [35] provides a language-agnostic way to express coding assumptions in .NET programs. Contracts are calls to static methods from the `Contract` class `Requires` and `Ensures`, specifying pre- and postconditions, respectively. In addition, we can specify class invariants. Invariants are placed in methods identified by the attribute `[ContractInvariantMethod]`. Those methods are composed only by a sequence of calls to the static method `Invariant` from `Contract` class.

In Source Code 5.1 we present the implementation of `headElement` with contracts written in Code Contracts. The postcondition – the static call to `Ensures` method declares that the return value `Result` will be not null when the queue is not empty – when the queue has at least one element.

Source Code 5.1: Example of DBC in Code Contracts.

```
public class AccountQueueImpl{
  public Queue<Account> accQueue = new Queue<Account>();

  public Account headElement(){
    Contract.Ensures(isEmpty() || Contract.Result<Account>() != null);
    if (isEmpty()){
      return null;
    } else{
      return accQueue.First();
    }
  }

  public bool isEmpty(){
    return accQueue.Count() == 0;
  }
}
```

Concerning tool support, Clousot [36] is the static verifier for Code Contracts, and Pex [122] is the dynamic conformance checking tool; the tools are integrated into the Microsoft Visual Studio development environment for the .NET platform.

5.2.2 ContractOk

As an additional contribution of this thesis, we leverage the RGT-based (randomly-generated tests) approach [84] to detect nonconformances in the C#/Code Contracts context by means of ContractOk (Figure 5.2). ContractOk is available online,[5] for Windows platforms under the GNU (GNU General Public License) GPL v3. We first discuss the process for detecting nonconformances, then, we discuss the manual classification performed after running the tool.

Detection

The following steps make up the approach: In Step 1, the system classes are compiled with Code Contracts activated (the binary re-writer). Then, in Step 2, tests are automatically generated and executed; randomly generated tests are composed of sequences of calls to methods and constructors under test in a given time limit [90]. Next, test results are compared against oracles established from the contracts (Step 3). In Step 4, two filters are applied: first, *meaningless* test cases are discarded [84] – tests violating a precondition in the first call to a method under test. The remaining *failures* consist of relevant contract violations, which are candidate nonconformances. The second filter categorizes failures into distinct faults – those faults make up the nonconformances subject to manual classification process (the establishment of likely causes).

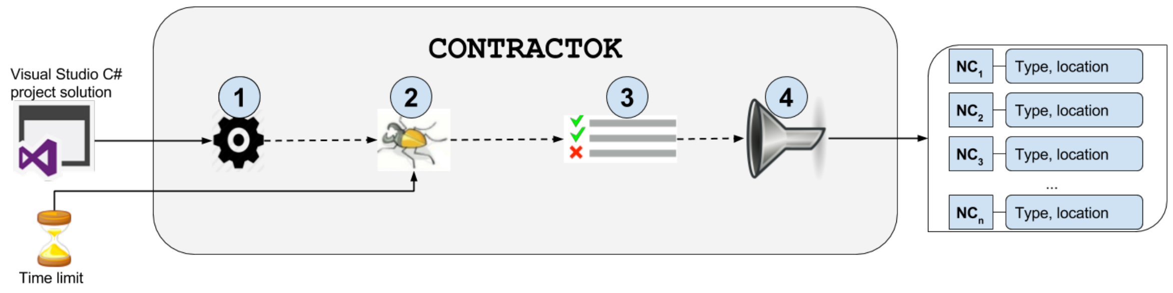


Figure 5.2: ContractOk approach for detecting nonconformances in C#/Code Contracts systems. The tool inputs are a C# project solution and a time limit for test generation.

Classification

Regarding nonconformance classification, we follow the heuristics from Milanez [79] for suggesting likely causes for nonconformances. For three types, namely *precondition*,

[5] https://github.com/alyssonfm/contractok

postcondition, and *invariant*, we updated the proposed model (the heuristics for *constraint* and *evaluation* types remain unchanged).

The study is developed in a heuristic-based approach so that the researcher may follow the analysis flow until identifying a likely cause for a nonconformance. The heuristics represent a way to establish a pattern for the analysis. For each type of nonconformance the first analysis is made in the source code: if the method's body in which the nonconformance was detected (or some intermediate method) modifies a value breaking the *precondition*, the *postcondition* or the class *invariant*, the likely cause for the nonconformance can be pointed as *Code error*.

Specifically for *precondition*, if there is no check for the value received as a parameter (or sent to the method in which the nonconformance occurred), the likely cause suggested is *Weak precondition*; otherwise, *Weak postcondition* can be suggested when an intermediate called method does not restrict the values passed as parameters to the method where the nonconformance occurred. Furthermore, the *Strong precondition* may also be suggested when a method has too many clauses in the precondition that makes it difficult to be satisfied.

Concerning *postcondition*, if the precondition is too permissive the likely cause suggested is the *Weak precondition*. It is also possible to suggest *Weak postcondition* as a likely cause, it happens if a postcondition of a called method allows the returns of a value that breaks the postcondition of the caller method. The last likely cause that can be assigned to *postcondition* is *Strong postcondition*, occurring if the method has many postcondition clauses making it difficult to satisfy all of them.

For *invariant*, the non-existence of a precondition in the method where the nonconformance occurred or in some intermediate called method is expressed by the likely cause *Weak precondition*. Otherwise, if a postcondition allows a value that breaks the class invariant after the method's execution the likely cause suggested will be *Weak postcondition*. Similarly to the Strong pre- and postcondition likely causes, an *invariant* nonconformance can be caused by a *Strong invariant*, when there are many invariant clauses that cannot be satisfied by the present or some called method.

5.3 A Study on Java/JML and C#/Code Contracts Open Source Systems

We evaluate each tool – JMLOK2 and CONTRACTOK – in 12 real projects available on the literature with respect to automatic detection and manual classification of nonconformances. Moreover, we investigate how DBC practitioners write contracts by analyzing all systems evaluated with the tools. First, we present the study definition (Section 5.3.1) and planning (Section 5.3.2). Then, Sections 5.3.3 and 5.3.4 present and discuss the results, respectively. Section 5.3.5 describes the threats to validity. Finally, Section 5.3.6 summarizes the main findings. The evaluation follows two main steps: (1) we run each tool in 12 real contract-based systems to detect nonconformances, and (2) we manually classify all detected nonconformances in order to establish likely causes.

5.3.1 Definition

The goal of this case study is to analyze one approach (implemented for both tools) for the purpose of evaluation with respect to automatic detection and manual classification of nonconformances from the point of view of researchers in the context of contract-based programming. In particular, we address the following research questions:

- **Q1.** How many nonconformances is the approach able to detect in real contract-based programs for a given time limit, and which are the most common types and likely causes of those nonconformances?

  We measure the number of detected nonconformances (#NCs, henceforth) in each system within a given test generation time limit, and analyze the frequency of nonconformances by types and likely causes.

- **Q2.** What is the testing cost for detecting a nonconformance?

  We measure and summarize metrics that define the complexity of the first failing test case and the complexity for identifying a likely cause for the nonconformance, in terms of metrics B and D (see Section 5.3.2), respectively.

In order to discover how practitioners write contracts in JML and Code Contracts, some additional questions are defined:

- **Q3.** Which are the most common contract types written by developers?

  We measure the frequency of contract clauses (#CC) for each type of contract in each experimental unit and discuss if there are significant differences between the use of each contract type.

- **Q4.** Is there any relationship between contracts complexity and the number of detected nonconformances? And what is the relationship between contract type and nonconformance type?

  We measure contract complexity (*CCo*) and relate this metric with #NC for each system, by using a correlation test. We also relate the most common contract type with the most common nonconformance type.

- **Q5.** Which are the most common nonconformance types for the units with the most complex contracts?

  We rank the top five systems for *CCo* and group the detected nonconformances in the respective unit according to the nonconformance type.

- **Q6.** What is the relationship between contract type and the number of nonconformances per contract clause (nonconformance ratio)?

  We discuss the relationship between contract type and the number of nonconformances.

5.3.2 Planning

In this section, we present the design of this case study. First, we show the units used, then, the experimental procedure. Next, we describe the tools and detail the manual classification procedure.

Experimental Units

The case study is performed on 24 open source projects, summing up to 148 KLOC and 12.2 K Contract Clauses [34] (KCC, henceforth). From those, the 12 JML sum up to

26 KLOC and more than 4.2 KCC and the 12 Code Contracts sum up to 122 KLOC and 8 KCC. These projects were the only open-source projects from GitHub,[6] CodePlex,[7] and BitBucket,[8] that could be compiled by our tools.

Regarding JML projects, they include sample programs available in the JML web site[9], composed by 11 example programs for training purposes, written by specialists in the JML language.[10] Also, the study includes programs collected from 11 open-source JML projects. While `Bank`, `PayCard`, and `TheSchorrWaiteAlgorithm` are presented in the KeY approach book [7], `Bomber` [101] is a mobile game, and `Dnivra-Jive` is the set of programs used by the Jive JML Dynamic Program Verifier[11]; `HealthCard` [105] is an application that manages medical appointments into smart cards. `JAccounting` is a case study from the *ajml* compiler project [101], implementing an accounting system. Likewise, `Javacard` is a technology used to program smart cards and it comprises a subset of the desktop Java; a subset of the programming language itself and a cut down version of the API (The current API implementation has been developed for the KeY interactive verification system[12]). `Mondex` [110] is a system whose translation from original Z specification was developed in the Verified Software Repository[13] context. In its turn, `PokerTop` is a poker system.[14] Finally, `TransactedMemory` [96] is a specific feature of the Javacard API.

With respect to Code Contracts, while `AutoDiff` [112] is a library for automatic differentiation of mathematical functions and `Boogie` [4] is an Intermediate Verification Language (IVL) for proof obligations solved by reasoning engines, `Contractor` builds contract specifications with type-state information, and `DotNetComponentOrientedProgramming` exemplifies component-oriented programming in .NET. In addition, `DotNetExtensions` contains some numerical exten-

[6] `https://github.com/`

[7] `https://www.codeplex.com/`

[8] `https://bitbucket.org/`

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

[10] `dbc`, `digraph`, `dirobserver`, `jmlkluwer`, `jmltutorial`, `list`, `misc`, `reader`, `sets`, `stacks`, `table`, and an adaptation of the subpackage `stacks – stacks2`. Two other programs (`prelimdesign` and `jmlrefman`) could not be compiled.

[11] `https://bitbucket.org/dnivra/jive-jml-dynamic-program-verifier`

[12] `http://www.key-project.org/case_studies/`

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

[14] `https://github.com/topless/PokerTop`

sions for .NET, `DRail` parses routes and tracks, and `EuroManager` is a football manager online with graphical simulation. `Frost` is a hardware accelerated drawing and composition library written in C#. Finally, `MonoMobileFacebook` is a Facebook C# SDK for MonoMobile, the `NeuroFlow Overhaul` is a Machine Learning Algorithm Library; `ProgrammingWithCC` is a set of exercises using C# and Code Contracts, and `Yandex` is a .NET interface to the JSON-based Yandex.Direct API.

The companion website [85] groups complete descriptions of all considered systems, which are listed in Table 5.3, all metrics considered in this study, and a replication package of this study.

Study procedure

The study consists of two main steps (see Figure 5.3): (1) run the tools over each contract-based program in a given time limit (the time limit is the time used for tests generation [91]), and (2) manually classify each detected nonconformance in terms of its likely cause.

JmlOk2 and ContractOk

JmlOk2 and ContractOk detect nonconformances by automatically generating and executing tests in a given time limit (by using Randoop [91]), comparing the test results with oracles (generated from the contracts) (Steps 1.A to 1.C). After test execution, two filters are applied (Step 1.D): first, *meaningless* test cases are discarded [84] – tests violating a precondition in the first call to a method under test, which would be false positives, as they convey test data inadequacy (side effect from test generation). The remaining *failures* consist of relevant contract violations, which are candidate nonconformances. The second filter categorizes failures into distinct faults – those faults make up the nonconformances subject to the manual classification process.

We performed the case study on a 3.6 GHz core i7 with 16 GB RAM running Windows 8.1 Enterprise, backed by Java 1.7 update 80, JML compiler 5.6_rc4, Visual Studio Community 2015, and Code Contracts 1.9.10714.2. Since Randoop [91] requires a time limit for test generation – the time after which the generation process stops – and we want to extend the evaluation of JmlOk2 performed in previous works [83; 84], we varied time limits from the 60s to 600s, each interval executed 10 times; aiming to stabilize the number of detected nonconformances through time, we ran JmlOk2 with the time limit varying until 1,100s for the unit `Samples` and ContractOk for the unit

Table 5.3: Experimental units' summary. Column LOC shows the code size of each experimental unit. Column #CC presents the total of contract clauses of each experimental unit. And columns #Pre to #Const display the clauses grouped by their types.

		Contract clauses				
Experimental Unit	**LOC**	**#CC**	**#Pre**	**#Post**	**#Inv**	**#Const**
Bank	792	126	66	43	17	0
Bomber	6,258	121	43	78	0	0
Dnivra-Jive	232	143	30	108	5	0
HealthCard	2,156	1,019	646	235	104	34
JAccounting	6,648	194	98	94	2	0
Javacard API	3,294	33	14	4	15	0
Mondex	655	174	28	104	39	3
PayCard	110	78	21	40	17	0
PokerTop	309	190	80	95	15	0
Samples	3,855	1,834	683	912	237	2
TheSchorrWaiteAlgorithm	110	23	11	9	3	0
TransactedMemory	1,779	295	149	52	93	1
JML	**26,198**	**4,230**	**1,869**	**1,774**	**547**	**40**
Experimental Unit	**LOC**	**#CC**	**#Pre**	**#Post**	**#Inv**	**#Const**
AutoDiff	1,291	157	101	56	0	-
Boogie	66,971	5,214	3,071	1,554	589	-
Contractor	10,420	507	292	150	65	-
DotNetComponentOrientedProgramming	7,180	147	71	73	3	-
DotNetExtensionsImproved	1,384	39	38	1	0	-
DRail	628	60	31	29	0	-
EuroManager	4,853	190	190	0	0	-
Frost	12,014	1,156	892	231	33	-
MonoMobileFacebook	5,332	192	160	32	0	-
NeuroFlow Framework Overhaul	10,626	377	326	39	12	-
ProgrammingWithCodeContracts	119	14	5	7	2	-
Yandex	1,972	18	18	0	0	-
Code Contracts	**122,790**	**8,071**	**5,195**	**2,172**	**704**	**-**

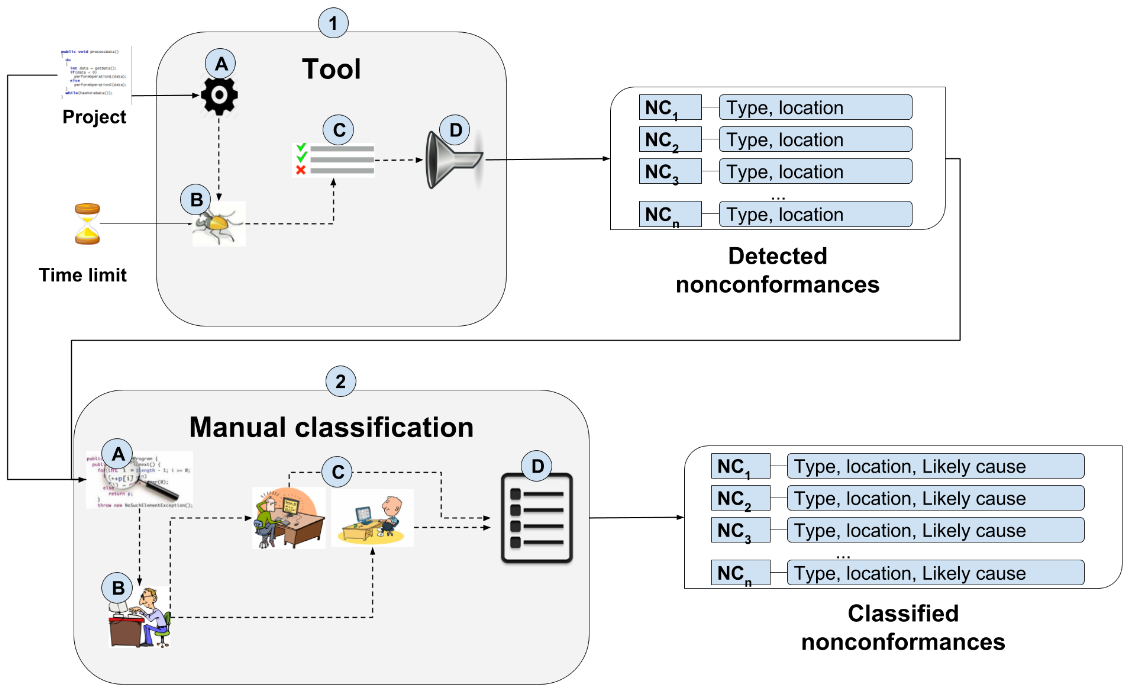

1
Tool
A
B
C
D
Project
Time limit
NC1 Type, location
NC2 Type, location
NC3 Type, location
...
NCn Type, location
Detected nonconformances
2
Manual classification
A
B
C
D
NC1 Type, location, Likely cause
NC2 Type, location, Likely cause
NC3 Type, location, Likely cause
...
NCn Type, location, Likely cause
Classified nonconformances

Figure 5.3: Steps performed in this study. First, we use the tools for detecting nonconformances in each system. Then, we perform the manual classification of likely causes for each detected nonconformance. First, each researcher tries to establish a likely cause (Steps 2.A and 2.B) by inspecting the source code and contracts; then, each researcher reviews the classification performed by the others (2.C); finally, in a discussion session (2.D) the most suitable likely cause is established.

`Boogie`. These limits were chosen based on the fact we test the systems as a whole, not only a class by time as similar works do [77]. Furthermore, based on the tool results for each experimental unit, we can suggest the best cost-benefit for choosing a time limit. As a basis, we suggest starting with a low time limit, such as 60s and increases this value based on the budget for the test generation step. All systems source code, generated test cases, and JMLOK2 binaries are available as a replication package on the companion website [85].

Manual classification

Each detected nonconformance results in the following manual steps: first, the researcher examines the source code and its respective contracts. Then, he examines the test case that shows the nonconformance, so he will be able aware of the state changes performed until the program fails. After that, the error *category* can be established: code, contract, or both. Finally, the fault must be diagnosed, before it can be fixed (Steps 2.A and 2.B).

After the first classification, each researcher reviews the classification performed by the others (in this study, three researchers individually classified all nonconformances - Step 2.C). Then, in a discussion session, they establish the most suitable likely cause for each nonconformance (Step 2.D). We tried to contact the developers of all systems used in this study, but we did not receive answers from all of them until the writing of the present thesis; however, those who answered our contact (developers of four of the evaluated systems) confirmed the nonconformances and agreed to our classification.

We perform a kappa test [21] in order to investigate the agreement between the researchers' classification and we get an unweighted kappa of 0.74 indicating a substantial strength of agreement [66] between the classification performed by the researchers with a confidence level of 95% for the nonconformances into Java systems; and we get an unweighted kappa of 0.21 indicating a fair strength of agreement [66] between the classification performed by the researchers with a confidence level of 95% for the nonconformances in Code Contracts projects.

Researchers considered as causes: weak or strong precondition, strong postcondition, strong invariant, and code error – weak invariant and weak postcondition are ruled out as researchers cannot judge lack of general system restrictions without deep knowledge about the requirements.

For each nonconformance (nc) we manually collect *breadth* (B) and *depth* (D) metrics. The first measures the number of top calls within the test method until the failure occurs. This metric is defined in Equation 5.1; the $calls(tm)$ returns the sequence of method calls into the test method (tm). The second is the call *depth* (Equation 5.2) 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.

$$B(nc, tm) = position(nc, calls(tm)) \tag{5.1}$$

$$\begin{aligned}
&D(nc, tm) = \\
&\quad let\ p = position(nc, calls(tm)) \\
&\quad let\ m = method(calls(tm)[p]) \\
&\qquad if\ (nc \in body(m))\ then\ result =\ 1 \\
&\qquad else\ result =\ 1 + D(nc, m)
\end{aligned} \tag{5.2}$$

For quantifying contract clauses, we follow Estler et al. [34] approach, in which the number of contract clauses is a proxy for contract complexity. The *CCo* metric (Equation 5.3) is the ratio between #CC and LOC. The implementation for counting contract clauses and lines of code is available online.[15]

$$CCo(x) = \frac{\#CC(x)}{LOC(x)} \tag{5.3}$$

For the system presented in our motivating example (Chapter 1.1.1), B is equal to 4 because the test case (Source Code 1.3) has the creation of one `Counter` object and three calls to `updateCount` method; and D is equal to 1 because we just need to look into `updateCount`'s method in order to figure out the problem. Regarding CCo metric (Equation 5.3), the following values are computed: LOC = 10; #CC = 5; #Pre = 0; #Post = 3; #Inv = 2. Therefore, CCo = 4/10, CCo is equal to 0.4.

5.3.3 Results

JMLOK2 detects 119 nonconformances in the evaluated projects (see Table 5.4); from those, 24 nonconformances are new in experimental units that have been used in previous works with JMLOK2 [83],[84]: `Bank` (3), `Bomber` (5), `Dnivra-Jive` (6), `HealthCard` (41), `JAccounting` (26), `Javacard` (7), `Mondex` (2), `PokerTop` (1), `Samples` (18), `TheSchorrWaiteAlgorithm` (2), and `TransactedMemory` (8). In `PayCard`, JMLOK2 was not able to detect nonconformances with the configuration used (Section 5.3.2). Concerning the types, those nonconformances were distributed in the following manner: 51 *invariant*, 47 *postcondition*, 8 *constraint*, 7 *evaluation*, and 6 *precondition*.

[15] `https://github.com/igornatanael/util/tree/master/ContractCounter`

Most of the 119 detected nonconformances manually received *Weak precondition* (51) as likely cause, followed by *Code error* (38). Four units (`HealthCard`, `JAccounting`, `Samples`, and `TransactedMemory`) presented *Weak precondition* as the most frequent likely cause (see Table 5.4).

Table 5.4: For each system, #NC, grouped by type and likely cause. The columns represent nonconformance types, from left to right: precondition, postcondition, invariant, constraint, and evaluation. Also, likely causes: weak precondition, strong precondition, weak postcondition, strong postcondition, strong invariant, strong constraint, and code error.

	Types					**Likely causes**							
System	**Pre**	**Pos**	**Inv**	**Con**	**Eva**	**WPre**	**SPre**	**WPos**	**SPos**	**SInv**	**SCon**	**Code**	**Total**
Bank	0	1	1	0	1	0	0	0	2	1	0	0	3
Bomber	1	2	2	0	0	1	1	0	0	0	0	3	5
Dnivra-Jive	0	4	0	0	2	2	0	0	4	0	0	0	6
Health-Card	2	15	16	8	0	23	0	2	11	0	1	4	41
JAccoun-ting	0	11	12	0	3	9	0	0	1	0	0	16	26
Javacard	0	0	7	0	0	0	0	0	0	0	0	7	7
Mondex	0	0	2	0	0	0	0	0	0	0	0	2	2
PayCard	0	0	0	0	0	0	0	0	0	0	0	0	0
PokerTop	0	0	1	0	0	0	0	0	0	0	0	1	1
Samples	2	13	2	0	1	10	0	2	4	0	0	2	18
TheSchorr-WaiteAlg.	0	1	1	0	0	0	0	0	1	0	0	1	2
Transacted-Memory	1	0	7	0	0	6	0	0	0	0	0	2	8
JML	**6**	**47**	**51**	**8**	**7**	**51**	**1**	**4**	**23**	**1**	**1**	**38**	**119**

ContractOK detects 63 nonconformances in the 12 Code Contracts projects: `AutoDiff` (1), `Boogie` (17), `Contractor` (3), `DotNetComponentOrientedProgramming` (14), `DotNetExtensions` (1), `DRail` (2), `EuroManager` (3), `Frost` (15), `MonoMobileFacebook` (1), `NeuroFlowOverhaul` (3), `ProgrammingWithCC` (2), and `Yandex` (1). From those, 28 were *precondition*, 25 *postcondition*, and 10 *invariant* errors. From the manual classification, most nonconformances were assigned *weak precondition* as likely cause (40), followed by *code error* (18). Table 5.5 presents all nonconformances, grouped by type and likely cause. In addition, we determine which likely causes are assigned to each nonconformance type, as presented in Table 5.6.

Table 5.5: For each system, #NC, grouped by type and likely cause. Columns represent nonconformance types, from left to right: precondition, postcondition and invariant. Also, likely causes: weak precondition, strong precondition, strong postcondition, strong invariant, and code error.

	Types			Likely Causes					
System	Pre	Post	Inv	WPre	SPre	SPost	SInv	Code	Total
AutoDiff	1	0	0	0	1	0	0	0	1
Boogie	8	1	8	8	0	0	0	9	17
Contractor	1	0	2	1	0	0	0	2	3
DotNetCOP	0	14	0	12	0	0	0	2	14
DotNetExtensions	1	0	0	1	0	0	0	0	1
DRail	2	0	0	0	0	0	0	2	2
EuroManager	3	0	0	3	0	0	0	0	3
Frost	7	8	0	13	0	1	0	1	15
MonoMobileFacebook	1	0	0	0	0	0	0	1	1
NeuroFlowOverhaul	3	0	0	2	1	0	0	0	3
ProgrammingWithCC	0	2	0	0	0	2	0	0	2
Yandex	1	0	0	0	0	0	0	1	1
Total	**28**	**25**	**10**	**40**	**2**	**3**	**0**	**18**	**63**

Table 5.6: Likely causes for each nonconformance type.

	Likely Cause						
Type	**WPre**	**SPre**	**WPost**	**SPost**	**SInv**	**SConst**	**Code**
precondition	21	3	4	0	0	0	6
postcondition	41	0	0	23	0	0	8
invariant	20	0	0	0	1	0	40
constraint	7	0	0	0	0	1	0
evaluation	2	0	0	3	0	0	2

For JML, average *breadth* ranges from 1.00 (in `PokerTop` experimental unit) to 4.17 (in `Dnivra-Jive`); average *depth* varies from 1.00 (in `TheSchorrWaiteAlgorithm`) to 4.22 (in `Samples`) – see Table 5.7. Concerning how JML developers write contracts, we found a small difference between the use of preconditions (0.442) and postconditions (0.419) over the evaluated projects – see Table 5.7; on the other hand, the use of history constraint is limited to few projects (4 out of 12), correspondingly the number of contract clauses of this type is also small: just 40 from 4,230 contract clauses. For Code Contracts, average *breadth* ranges from 2.00 (`DRail`) to 14.00 (`AutoDiff`); average *depth* varies from 1.50 (`ProgrammingWithCC`) to 11.00 (`MonoMobileFacebook`) – see Table 5.7. Moreover, for 10 out of 12 systems, *precondition* clauses are the most common contract clause. On the other hand, *invariant* clauses are less common, they are present in only six systems (also see Table 5.7).

For JML, in three of the top-five units with respect to CCo metric, *Invariant* is the most common type of nonconformance. The contract type most common for the projects with the highest number of nonconformances detected is *Precondition* (three out of five) – see Table 5.7. Regarding the relationship between contract type and nonconformance type, contract clauses of *precondition* are more related to nonconformances of *invariant* type; whereas contract clauses of *postcondition* are equally related to *postcondition* and *invariant* types – see Table 5.7. Regarding contract complexity *CCo* for Code Contracts, the values vary from 0.01 (`Yandex`) to 0.12 (`AutoDiff` and `ProgrammingWithCC`). Regarding nonconformance type, *precondition* occurred more frequently in 7 out of 12 systems (see Table 5.7). Furthermore, *Precondition* clauses are the most common contract type in three out of the top five systems in terms of nonconformance ratio (see Table 5.7).

5.3.4 Discussion

We proceed with the discussion of the research questions. Regarding **Q1**, JMLOK2 detected a total of 119 nonconformances. The approach detected nonconformances to the set of example programs, written by JML specialists. 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 four nonconformances in methods that invoke, in their contract, `JMLDouble.approximatelyEqualTo`; `JMLDouble` is imported from the standard JML API, only being visible on the contract level; its method `approximatelyEqualTo` performs a 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.

From the 119 nonconformances detected, the most frequent type was the *invariant* – 51,

Table 5.7: For each system, we present the results of average *breadth* (B), *depth* (D), the ratio between each contract type and contract clauses, the value of *CCo*, the number of nonconformances detected (#NCs), the nonconformance ratio (#NCs/CC), the most common contract and nonconformance type.

System	**B**	**D**	$\frac{Pre}{CC}$	$\frac{Post}{CC}$	$\frac{Inv}{CC}$	$\frac{Cons}{CC}$	**CCo**	**#NCs**	$\frac{\#NCs}{CC}$	**Contract type**	**NC type**
Bank	2.00	1.33	0.524	0.341	0.135	0.000	0.16	3	0.024	Pre	Post and Inv
Bomber	1.40	2.00	0.355	0.645	0.000	0.000	0.02	5	0.041	Post	Post and Inv
Dnivra-Jive	4.17	3.67	0.210	0.755	0.035	0.000	0.62	6	0.042	Post	Post
HealthCard	3.46	2.61	0.634	0.231	0.102	0.033	0.47	41	0.040	**Pre**	Inv
JAccounting	1.69	1.38	0.505	0.485	0.010	0.000	0.03	26	0.134	**Pre**	Inv
Javacard	1.29	2.43	0.424	0.121	0.455	0.000	0.01	7	0.212	**Inv**	Inv
Mondex	1.50	1.50	0.161	0.598	0.224	0.017	0.27	2	0.011	Post	Inv
PayCard	–	–	0.269	0.513	0.218	0.000	0.71	0	0.000	Post	–
PokerTop	1.00	2.00	0.421	0.500	0.079	0.000	0.61	1	0.005	Post	Inv
Samples	3.67	4.22	0.372	0.497	0.129	0.001	0.48	18	0.010	**Post**	Post
TheSchorrWaite-Algorithm	1.50	1.00	0.478	0.391	0.130	0.000	0.21	2	0.087	Pre	Post and Inv
TransactedMemory	1.63	1.50	0.505	0.176	0.315	0.003	0.17	8	0.027	**Pre**	Inv
JML	**2.68**	**2.45**	**0.442**	**0.419**	**0.129**	**0.009**	**0.31**	**9.92**	**0.05**	**–**	**–**
AutoDiff	14.00	3.00	0.643	0.357	0.000	–	0.12	1	0.006	Pre	Pre
Boogie	6.71	2.12	0.589	0.298	0.113	–	0.08	17	0.003	Pre	Pre and Inv
Contractor	5.33	3.00	0.576	0.296	0.128	–	0.05	3	0.006	Pre	Inv
DotNetCOP	6.71	5.14	0.483	0.497	0.020	–	0.02	14	0.095	Post	Post
DotNetExtensions	3.00	2.00	0.974	0.026	0.000	–	0.03	1	0.026	Pre	Pre
DRail	2.00	2.00	0.517	0.483	0.000	–	0.10	2	0.033	Pre	Pre
EuroManager	4.67	3.33	1.000	0.000	0.000	–	0.04	3	0.016	Pre	Pre
Frost	6.93	5.87	0.772	0.200	0.029	–	0.10	15	0.013	Pre	Post
MonoMobile-Facebook	10.00	11.00	0.833	0.167	0.000	–	0.04	1	0.005	Pre	Pre
NeuroFlowOH	5.33	4.67	0.865	0.103	0.032	–	0.04	3	0.008	Pre	Pre
Programming-WithCC	4.00	1.50	0.357	0.500	0.143	–	0.12	2	0.143	Post	Post
Yandex	5.00	5.00	1.000	0.000	0.000	–	0.01	1	0.056	Pre	Pre
Code Contracts	**6.14**	**4.05**	**0.644**	**0.269**	**0.087**	**–**	**0.07**	**5.25**	**0.008**	**–**	**–**

followed by *postcondition* violations – 47. 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 seven were evaluation errors). A few nonconformances were related to history constraints (eight) 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, the manual classification established *Weak precondition* as the main cause (with 51 instances – as presented in Table 5.4). In this case, a method allows values that should be denied for the correct method's execution – 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` – from all 98 precondition clauses, 44 are the default precondition. Maybe *Weak precondition* has been the most common case of likely cause due to the complexity involved in the establishment of preconditions; since the specifier does not previously know the system clients, an overly strong precondition deny access to several clients; on the other hand, if the precondition is too weak, many clients will not be able to get the expected result. Therefore, this represents an important trade-off for contract-based programs. However, *Code error* is also very recurrent, with 38 instances. This problem is probably related to different levels of abstraction between programming and contract languages, in which the developer may not understand a contract written by someone else, or even the contract may be too complex to be implemented, or even 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.

ContractOk detects 63 nonconformances in the systems evaluated; at least one conformance was detected in every system. Still, for **Q1**, most nonconformances are caused by *weak preconditions*, as suggested by our manual classification. Within a diversity of systems, with varying complexity and features, evidence shows preconditions are critical to the correctness of C# contract-based programs. Assigning inputs to program operations in the permitted range is indeed one of the most impactful design tasks, either for method callers or implementers. There is an important trade-off in every contract-based program when designing preconditions: if the developer of a method does not know its clients beforehand, he may construct an overly strong precondition, rejecting many clients; on the other hand, given the developer does not establish a precondition for a method, it may not be able to satisfy its contracts with the allowed client inputs.

Q2. *breadth* and *depth* are, on average, higher than 1, for all systems, which suggests that a typical nonconformance-revealing test presents sequences of at least two calls (horizontal complexity) and more than one indirect call to a problematic method (vertical complexity). By manually analyzing test cases revealing nonconformances, and observing the minimal test case able to detect them, the highest average values are for `AutoDiff` (14.00) and `MonoMobileFacebook` (11.00) for *breadth* and *depth*, respectively. Conversely, the smallest mean for *breadth* (1.00) was obtained in `PokerTop` and for *depth* (1.00) in `TheSchorrWaiteAlgorithm`. These results suggest that finding nonconformances is challenging without the support of well-designed test cases.

Q3. Developers apparently prefer to write pre- and postcondition clauses in comparison with invariants (see Table 5.7). For further evidence, we performed a paired t-test [62] with the null hypothesis that the average of preconditions and postconditions is the same for the evaluated systems. The test resulted in a p-value of 0.021, allowing us to reject the null hypothesis samples do not differ, on average, with a confidence level of 95%. As a conclusion, the use of precondition clauses in the studied systems outperform the use of postcondition clauses, which corroborates with Estler et al. [34]: there is no preference for certain contract type, however, preconditions, tend to have more clauses than postconditions.

The number of nonconformances is relatively proportional to system size; `DotNetComponentOrientedProgramming` is exceptional, with a high occurrence of nonconformances in a low number of contract clauses (14 over 147) – still, the system presents high LOC, which surely indicate nonconformances are due to wrong method behavior, not only contracts themselves. Since this system was developed for educative purposes, perhaps developers did not enforce conformance as should be done in more critical systems. `Boogie`, if compared to the rest of the sample, is much larger (approximately five times as large as the second, both in LOC and #CC), but shows only a few more nonconformances than other large systems (for instance, `Frost`), which may be an indication of quality, which is certainly required for a rather complex verification system such as `Boogie`. In addition, we found evidence that when contract complexity grows, the number of nonconformances detected also grows (see Table 5.7). This result corroborates with Polikarpova et al. [95]: more contract clauses allow the detection of more nonconformances.

Regarding the relationship between contract and nonconformance type (Table 5.7), systems with mostly postconditions tend to present, more often, postcondition and invariant nonconformances – indicating the postconditions may be harder to satisfy, however, they are often insufficient to avoid invariant errors. Considering systems in which preconditions predominate, more preconditions imply more precondition nonconformances – showing that those preconditions might be too strong to be satisfied by method callers. This result may be related to the trade-off in establishing preconditions: if the precondition is permissive, the clients may be not able to get the expected value; however, a restrictive precondition can rule out valid clients. Moreover, since precondition nonconformances occur when a method internally calls other methods, the involved preconditions might be best placed in other methods within the call trace, or the body of other methods in the chain should deal with the values received as parameters in order to maintain the preconditions of methods further called.

Q5. Precondition nonconformances are preponderant in three of the top five complex-contract systems (Table 5.7). This result is expected because the precondition clauses are the most common and in accordance with the results of Polikarpova et al. [95], more contract clauses allow detecting more nonconformances. Moreover, as the preconditions act as filters to input values, they are able to avoid calls to methods in situations in which a postcondition or an invariant clause could be violated.

Q6. Postcondition clauses are the most common in the systems with highest nonconformance ratios (`ProgrammingWithCC` and `DotNetCOP`). This is expected, since preconditions avoid method executions with forbidden values, reducing the nonconformance ratio. This result suggests more postcondition clauses imply more nonconformances per contract clause.

5.3.5 Threats to validity

This study has some limitations; next, we describe some threats to the validity of our evaluation.

Internal validity

The randomness promoted by the use of an automatic test generator (Randoop [91],[90]) by the tools is an internal threat to the validity of our study; so, we ran each system 10 times for each time limit (varying from 60 to 600 seconds) for confidence. In each execution, tests are generated independently of the previous run, which may show different contexts revealing the same nonconformances.

In addition, Randoop is not the only randomness to consider; we observe that the `Boogie` system behaves randomly during the detection phase, showing instability on the number of nonconformances detected and on the number of test cases generated. Considering the 10 executions carried out for the time of 60 seconds and the respective number of nonconformances for the top three systems in size (LOC and #CC) and number of nonconformances detected (#NC), systems `DotNetComponentOrientedProgramming` and `Frost` present a small variation on the number of nonconformances, with an average number of 11.8 (between 10 and 12) with 12 for the median and 8.6 (between 7 and 9) with 9 the for median, respectively. These numbers are explained by the randomness promoted by Randoop.NET and by the machine environment itself. On the other hand, `Boogie` presents an average number of 8.1 nonconformances (between a minimum of 2 nonconformances and maximum of 12) with 8.5 for the median, being possible to see a considerable number of standard deviation, 3.41. In our study, for the `Boogie` system, we ran 10 times each time limit for test generation, and report the highest number of detected nonconformances, thus minimizing (but not eliminating) chances of false negatives.

Conclusion validity

Randoop is based on randomization for generating distinct tests; to mitigate this threat, we ran each tool 10 times for each system and time limit. Another conclusion threat is related to *CCo*, based on the ratio between the number of contract clauses and LOC, which may not be representative of contract complexity. We followed this approach for its simplicity, as it has been used by related research [34].

Construct validity

Perhaps metrics B and D are not representative of the real trouble in detecting nonconformances. We defined these metrics as a surrogate for the complexity of test cases detecting the nonconformance. They are imperfect in measuring the complexity to visually find a nonconformance, for instance. Also, to reduce the threat on the manually-defined likely cause, all nonconformances were classified and reviewed by three researchers, separately. Finally, the detection approach is incomplete, since it is test based. Some nonconformances may be as well due to updates in the language's compiler and runtime.

External validity

A Randoop-based approach lacks repeatability, in terms of machine setting or operating system. Also, even though we diversified our choice of contract-based systems, varying in code size and contract clause count, generalizing the obtained results is almost impossible.

5.3.6 Answers to the Research Questions

From our results, we made the following observations:

- **Q1.** How many nonconformances is the approach able to detect in real contract-based programs for a given time limit, and which are the most common types and likely causes of those nonconformances?

  The tools detect 182 nonconformances in all 24 systems evaluated (JMLOK2 detects 119 nonconformances and CONTRACTOK detects 63 nonconformances). Most nonconformances are caused by *weak preconditions*, as suggested by our manual classification. Within a diversity of systems, with varying complexity and features, evidence shows preconditions are critical to the correctness of contract-based programs. Assigning inputs to program operations in the permitted range is indeed one of the most impactful design tasks, either for method callers or implementers.

- **Q2.** What is the testing cost for detecting a nonconformance?

  Breadth and *depth* are, on average, higher than 1, for all systems, which suggests that a typical nonconformance-revealing test presents sequences of at least two calls (horizontal complexity) and more than one indirect call to a problematic method (vertical complexity).

- **Q3.** Which are the most common contract types written by developers?

  Developers apparently prefer to write pre- and postcondition clauses in comparison with invariants; however, there is no preference for certain contract type, however, preconditions, tend to have more clauses than postconditions.

- **Q4.** Is there any relationship between contracts complexity and the number of detected nonconformances? And what is the relationship between contract type and nonconformance type?

  The number of nonconformances is relatively proportional to system size; `DotNetComponentOrientedProgramming` is exceptional, with a high occurrence of nonconformances in a low number of contract clauses (14 over 147) – still, the system presents high LOC, which surely indicate nonconformances are due to wrong method behavior, not only contracts themselves. Regarding the relationship between contract and nonconformance type (Table 5.7), systems with mostly postconditions tend to present, more often, postcondition and invariant nonconformances – indicating the postconditions may be harder to satisfy, however, they are often insufficient to avoid invariant errors.

- **Q5.** Which are the most common nonconformance types for the units with the most complex contracts?

  Precondition nonconformances are preponderant in three of the top five complex-contract systems. This result is expected since the precondition clauses are the most common; and, in accordance to the results of Polikarpova et al. [95], more contract clauses allows the detection of more nonconformances. Furthermore, as preconditions act as filters to input values, they are able to avoid method calls in situations in which a postcondition or an invariant clause could be violated.

- **Q6.** What is the relationship between contract type and the number of nonconformances per contract clause (nonconformance ratio)?

  Postcondition clauses are the most common in the systems with highest nonconformance ratios (`ProgrammingWithCC` and `DotNetCOP`). This is expected, since preconditions avoid method executions with forbidden values, reducing the nonconformance ratio. This result suggests more postcondition clauses imply more nonconformances per contract clause.

Chapter 6

Related Work

Our work is related to the following research areas: source code documentation; processing of Javadoc-tagged comments; conformance checking in Design by Contract; nonconformance classification; and automatic test generation.

Section 6.1 presents the main approaches for documenting source code we identify as related to our work. Then, Section 6.3 summarizes research on conformance checking of contract-based programs. Next, Section 6.4 describes approaches for classifying contract violations as a way of helping the developer when correcting those violations. Finally, Section 6.5 shows automatic test generation approaches.

6.1 Source Code Documentation

According to Meyer [76], the use of Design by Contract is essential in the production of reusable software elements and, more generally, in organizing the module interfaces in large software systems. He claims that preconditions, postconditions, and class invariants provide potential clients of a module with basic information about the services offered by the module, expressed in a concise and precise form. Therefore, proposals such as JML [68], Spec# [5], Code Contracts [35], and Eiffel [75] are related to CONTRACTJDOC. Whereas those approaches propose new notation, CONTRACTJDOC extends the tagging system from Javadoc and has mechanisms for turning the tags into runtime checkable contracts. CONTRACTJDOC approach fulfills the gap between informal documentation (as with Javadoc) and formal specification (such as JML [68] or Code Contracts [35]). Being closer to the way developers already write comments in Javadoc notation, it is likely that CONTRACTJDOC is easier to use than JML (see Section 3.3).

On the other hand, Torchiano [124] describes a structured approach to document pattern use in Java, instead of using a DBC language such as Meyer [76] work. His solution is based on the standard Javadoc tool and generates HTML documentation. He proposes four tags for documenting the use of patterns in Java programs: `@pat.name` - for the name of a pattern; `@pat.role` - for the pattern role; `@pat.task` - for the pattern task; and `@pat.use` - for the pattern use. The first three pattern specific tags can be used to describe whole classes, methods, and fields. The last one is typically used to describe methods that use pattern instances. Whereas tags in Torchiano's approach are used for structural/syntactical reasoning, in CONTRACTJDOC we use tags for semantic purposes: improve documentation and enable conformance checking of Java programs. The approaches could be integrated in order to perform conformance checking of design patterns.

Still, on applying knowledge from Javadoc, Zhai et al. [132] present a technique that builds models for Java API functions by analyzing the documentation. Their models are simpler implementations in Java compared to the original ones and hence easier to analyze. More importantly, they provide the same functionalities as the original functions. They argue that API documentation, like Javadoc and .NET documentation, usually contains wealthy information about the library functions, such as the behavior and exceptions they may throw. Thus it is feasible to generate models for library functions from such API documentation. In this context, the comments in CONTRACTJDOC approach can be used as input for the technique in order to improve model generation.

Some approaches establish the generation of contracts. In this context, Daikon [32; 33] is an implementation of dynamic detection of likely invariants. An invariant is a property that holds at a certain point or points in a program. Program invariants can help programmers by identifying program properties that must be preserved when modifying the code. Thus, invariants are useful for asserting statements, documentation, and formal specifications. Daikon can detect invariants in C, C++, Java, and Perl programs, and in record-structured data sources, making the process of writing contracts easier. CONTRACTSUGGESTOR does not support automatic generation of invariants yet, but we also improve the contracts use by providing an approach for automatically generating contracts for pre- and postconditions.

Table 6.1 summarizes those approaches with respect to the scope of the documentation supported.

Table 6.1: Related Work on documentation of source code.

	Approach scope
JML [68]	It extends Java language for supporting Design by Contract.
Spec# [5]	It extends C# language for supporting Design by Contract.
Eiffel [75]	It is a complete language for supporting Design by Contract.
Torchiano [124]	It provides constructs for documenting design patterns.
Zhai et al. [132]	It generates models for Java API functions.
Daikon [32; 33]	It detects invariants for C, C++, Java, and Perl.
CONTRACTJDOC	It provides tags for documenting the behavior of Java methods and classes.
CONTRACTSUGGESTOR	It suggests contracts based on tagged-comments in natural language.

6.2 Javadoc comment processing

Code commentary embeds natural language annotations in source code [24]. Comments should be significant for developers and maintainers: developers should use commentary to make software properly documented and to improve source code comprehensibility and maintainability [2]. However, the flexibility provided by natural language may result in useless information for those developers who did not originally authored source code. In this context, some techniques have been proposed to automatically process Javadoc-tagged comments [118; 49; 117] in order to extract useful information from those natural language texts.

Tan et al. [118] present @TCOMMENT approach. @TCOMMENT is an approach for testing Javadoc comments, specifically method properties about null values and related exceptions. The approach is composed of two components. The first component takes as input source files for a Java project and automatically analyzes the English text in Javadoc comments to infer a set of likely properties for a method in the files. The second generates random tests for these methods, checks the inferred properties, and reports inconsistencies. @TCOMMENT analyzes comments written in a natural language to detect comment-code inconsistencies and to improve automated testing. This approach is closely related to CON-

TRACTSUGGESTOR, however, in CONTRACTSUGGESTOR we generate contracts (pre- and postconditions) for non-null and relational values (both parameters and return values).

Instead of checking for null values and exceptional, Goffi et al. [49], propose a technique that automatically creates test oracles for exceptional behaviors from Javadoc comments. The technique uses a combination of natural language processing and runtime instrumentation. Their implementation, Toradocu, can be combined with a test input generation tool. The experimental evaluation shows that Toradocu improves the fault-finding effectiveness of EvoSuite and Randoop test suites by 8% and 16% respectively, and reduces EvoSuite's false positives by 33%. Toradocu is concerned with exceptional behavior whereas CONTRACTSUGGESTOR concerns contracts for program properties (both parameters and return values). Mixing both approaches could produce good results.

On the other hand, Tan et al. [117] propose converting programmers' intentions inferred from the comments and code they write into formal annotations and use these annotations to detect interrupt related OS concurrency bugs. Since the significant amount of effort involved in annotating programs can greatly limit the impact of annotation languages. They propose aComment tool. aComment automatically propagates annotations starting from a few IRQ annotations, i.e., annotations for functions that directly disable, enable or restore interrupts. For the Linux kernel, they manually identified 4 interrupt disabling functions, 2 interrupt enabling functions, and 2 interrupt restoring functions. aComment takes the 8 IRQ annotations as input, and automatically propagates them to all other functions, a total of 96,821 annotations. CONTRACTSUGGESTOR analyses only Javadoc-tagged method comments and automatically suggest pre- and postconditions for some properties: non-null and relational.

Table 6.2 summarizes those approaches with respect to (1) the technique used for processing the Javadoc comment; (2) the aim of the processing.

6.3 Conformance Checking

In the recent years, several efforts on verified software [12; 27; 77] have been carried out in the context of source code specification with contract-based languages [75; 68; 5; 35], and the Design by Contract (DBC) methodology [76]. Dynamic checking of contracts, despite its essential incompleteness, provides immediate feedback for developers, even when they

Table 6.2: Related Work on Javadoc-comments processing.

	Technique	Aim
@tComment [118]	Simple heuristics to analyze the free-form text	Infer method properties for null values and related exceptions in Java libraries/frameworks.
Toradocu [49]	Natural Language Processing and Run-time instrumentation	Create test oracles for exceptional behaviors.
aComment [117]	Simple program analysis with effective heuristics	Generate annotations to detect interrupt related OS concurrency bugs.
ContractSuggestor	Machine Learning Supervised Learning	Generate contracts (pre- and postconditions) for program properties.

write only partial contracts. The detection of nonconformances is, in this case, dependent on the quality of the test cases that exercise the runtime assertions produced by the contracts.

For DBC, a related approach proposes auto tests [77], where contracts are used as oracles, and the test generation is performed automatically. The AutoTest tool is an implementation of conformance checking to the Eiffel language [75]. This tool is similar to JmlOk2 and ContractOk: both tools aim at conformance checking, and use randomly-guided tests generation (ARTOO [20] for AutoTest and Randoop [91; 90] for JmlOk2 and ContractOk). However, AutoTest supports mixing manual and automated test, while ContractOk supports only tests automatically generated. In scenarios where tests automatically generated are not able to explore system's behavior, it is interesting to support manual testing since the developer will be able to improve the suite generated by adding manual tests. JmlOk2 is directed to JML, which is relatively simple to apply to existent Java programs; in addition, the tool is concerned with automatic nonconformance classification. And ContractOk is tailored for C#/Code Contracts context.

Concerning Spec# language, Boogie [4] is the Spec# static program verifier. This tool generates logical verification conditions for a Spec# program, which are analyzed by an automatic theorem prover in order to find errors in the program. Boogie provides feedback about syntax, typing, and semantic errors. The tool has a separation between proof obligation generation and verification phases. In addition, the tool performs loop-invariant inference using abstract interpretation and generates verification conditions to be passed to an automatic theorem prover. As ContractOk, Boogie also performs conformance checking; the difference is that Boogie concerns static checking whereas ContractOk concerns dynamic checking. Those approaches could be integrated in order to dynamically verify the conditions that Boogie is able to generate during program verification.

For the Code Contracts context, there are two main approaches for conformance checking: statically, by means of Clousot [36]; and dynamically, by means of Pex [122].[1] Clousot, like Boogie, is an abstract interpretation-based static checker. The tool analyzes every method in isolation, using the usual assume/guarantee reasoning. The precondition of the method is turned into an assumption and the postcondition into an assertion. For public methods, the object invariant is assumed at the method entry and asserted at the exit point. The approach has heuristics for sorting the warning messages, trying to report the more relevant first. In addition, the tool analyzes annotated programs to infer facts (including loop invariants), and uses this information to discharge proof obligations, helping to reduce the annotation burden by inferring some postconditions. However, Clousot can report false warnings in two main cases: (i) it does not know some external fact (for instance some third-party library methods return a non-null value); (ii) it is incomplete (as all the static analyses). Even though CONTRACTOK be incomplete (as all dynamic approaches), the tool has a filter for not reporting false positives when detecting nonconformances.

Pex [122] is a systematic approach based on a test-generation technique, called Dynamic Symbolic Execution (DSE). DSE is a technique that explores a method under test (MUT) and generates test inputs that can achieve high structural coverage of the MUT. The tool explores a MUT with default inputs. During exploration, Pex collects constraints on inputs from the predicates in branch statements. The tool negates collected constraints and uses a constraint solver to generate new inputs that guide future program explorations along different paths. To generate method sequences, Pex uses a simple heuristic-based approach that generates fixed sequences based on static information of constructors and other methods (of classes under test) that set values to member fields, helping to produce desired object states. CONTRACTOK, in its turn, uses a feedback random-based test generation for dynamic conformance checking.

There are some tools that apply dynamic checking for JML programs. JMLUnit [17] is a semi-automatic tool to check conformance, generating test case skeletons by combining calls to the methods under test, lacking test. In order to overcome some JMLUnit limitations, JMLUnitNG [133] automatically generates test data for non-primitive types; however,

[1]Pex is currently named as IntelliTest.

it does not exempt users from manually providing test data in some situations. On the other hand, JMLOK2 is completely automatic and provides an automatic classification 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 generated by Korat only consist of one object construction and one method invocation over 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. JMLOK2 and CONTRACTOK, on the other hand, do not require implementation of functions and generates more than one call for the methods under test; in addition, JMLOK2 provides an automatic classification for the detected nonconformances.

Also semi-automatic, Jartege [89] is a tool 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, JMLOK2 is completely automatic and provides automatic nonconformance classification.

On the other hand, JET [15] is completely automatic. The tool applies dynamic testing for conformance checking in JML, by randomly generating test cases using contracts as oracles. The tool applies genetic algorithms 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, JET is closely related to JMLOK2, since 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, JMLOK2 presents automatic nonconformance classification, a feature not available at JET.

From the point of view of static approaches, ESC/Java2 [23] performs static verification in JML programs, applying a logic-based technique that statically verifies the occurrence of runtime violations of JML contracts. Nevertheless, ESC/Java2 is neither sound nor complete, this tool presents a high rate of false positives. Whereas JMLOK2 is sound because all nonconformances found are correct, but it is also incomplete because it is not ensured that the tool will found all nonconformances.

For Javadoc context, Tan et al. [118] present @TCOMMENT approach. @TCOMMENT is an approach for testing Javadoc comments, specifically method properties about null values and related exceptions. The approach is composed of two components. The first component takes as input source files for a Java project and automatically analyzes the English text in Javadoc comments to infer a set of likely properties for a method in the files. The second generates random tests for these methods, checks the inferred properties, and reports inconsistencies. They evaluated @TCOMMENT on seven open-source projects and found 29 inconsistencies between Javadoc comments and method bodies. Their work focuses on method properties for null values and related exceptions in Java libraries/frameworks; whereas JMLOK2 (with support to CONTRACTJDOC) and CONTRACTOK focus on pre- and postconditions and invariants.

Table 6.3 summarizes those approaches with respect to (1) the kind of conformance checking performed; (2) whether there are some classification of the nonconformances; (3) whether the approach is automatic; (4) the specification language used in the approach.

Table 6.3: Related Work on conformance checking.

	Conformance checking	**Classification**	**Automation level**	**Specification Language**
AutoTest [77]	dynamic	–	automatic	Eiffel
Boogie [4]	static	–	automatic	Spec#
Clousot [36]	static	–	automatic	Code Contracts
Pex [122]	dynamic	–	automatic	Code Contracts
JMLUnit [17]	dynamic	–	semi-automatictomatic	JML
JMLUnitNG [133]	dynamic	–	semi-automatic	JML
Korat [11]	dynamic	–	semi-automatic	JML
Jartege [89]	dynamic	–	semi-automatic	JML
JET [15]	dynamic	–	automatic	JML
ESC/Java2 [23]	static	–	automatic	JML
@TCOMMENT [118]	dynamic	–	automatic	Java/Javadoc
JMLOK2 [84]	dynamic	automatic	automatic	JML
CONTRACTOK	dynamic	–	heuristic-based	Code Contracts

6.4 Nonconformance Classification

Regarding classification, Rosenblum [106] 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, similar to *jmlc* compiler. His work presents two levels in which a problem (a contract violation) may happen: *Specification of Function Interfaces*, and *Specification of Function Bodies*, the first one is related to 119 – 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 the types we consider in JMLOK2: I1, I3, and I7 are related to preconditions; I2 and I6 are related to postconditions; and I4, I5, and I8 are related to invariants.

More recently, Polikarpova et al. [95] present three categories to classify nonconformances: *specification faults*, *inconsistency faults*, and *real faults*. In this work, we use 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.

Christakis et al. [18] present three cases for classifying invariant violations: (1) the object invariant is stronger than intended. In this case, one should weaken the invariant. (2) the invariant expresses the intended properties, but the program does not maintain it. This case constitutes a bug that should be fixed. (3) the invariant expresses the intended properties and can, in principle, be violated by clients of the class, but the entire program does not exhibit such violations. Our classification model is more general: our likely causes are more abstract than Christakis et al.'s classification. Their work can be integrated into our classification model in order to provide more details on the invariant problems that are detected.

Table 6.4 summarizes those approaches with respect to (1) the classification 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 classification proposed.

Table 6.4: Related Work on nonconformance classification.

	Classification scope	Automation level	Language
Rosenblum [106]	external and internal behaviors	manual	C
Polikarpova et al. [95]	external and internal behaviors	manual	Eiffel
Christakis, Müller, and Wüstholz [18]	invariant problems	manual	C#/Code Contracts
JMLOK2 [84]	only external behavior	automatic	JML

6.5 Automatic Test Generation

Software testing, although not guaranteeing that the software is error-free, is a widely-used approach to check software behavior. In contract-based programs, tests automatically generated can be used for conformance checking, 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 need more than one modification into the object under test. In our tools (JMLOK2 and CONTRACTOK), we use a random-directed test generation approach, by means of Randoop [91]. In Randoop, the feedback from the execution of sequence being constructed is used as pruning function – only valid constructions are considered in the next sequence of generations. This approach is similar to Adaptive Random Testing (ART) [14]. In ART, the test case generation is based on the idea that tests more distant are more probable to detect problems than tests separated by smaller distances; ART uses the Euclidean distance to calculate the distance between test cases. An extension of ART ideas is 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 related work is presented on JET [15], where genetic algorithms are used in test generation process. Genetic algorithms are based on feedback to the creation of new generations, similar to the feedback-directed approach from Randoop. Another approach to automatic test generation is EvoSuite [45]. The tool uses an evolutionary search approach that evolves whole test suites with respect to an entire coverage criterion at the same time. The EvoSuite approach, similar to CONTRACTOK, uses a guided approach to tests generation, in that case, a search-based approach.

Another approach for test generation is Parameterized Unit Tests (PUTs), proposed by

Tillman and Schulte [123]. PUT is a new methodology extending the current industry practice of closed unit tests. Test methods are generalized by allowing parameters. Parameterized test methods are specifications of the behavior of the methods under test: they do not only provide exemplary arguments to the methods under test but ranges of such arguments. In addition, PUTs describe a set of traditional unit tests which can be obtained by instantiating the parameterized test methods with given argument sets. Instantiations should be chosen so that they exercise different code paths of the methods under test. This kind of test is used in Pex [122] approach.

Xie et al. [130] present Symstra: a framework that achieves both test generation tasks – generating method sequences that build relevant receiver object states – and generating relevant method arguments – using the symbolic execution of method sequences with symbolic arguments. Symstra uses symbolic execution to exhaustively explore bounded method sequences of the class under test and to generate tests that achieve high branch and intra-method path coverage for complex data structures such as container implementations. Symstra exports concrete test sequences into a JUnit test class. It also exports a constraint associated with the test as a comment for the test in the JUnit test class. The user can configure Symstra to select only those generated tests that increase branch coverage or throw new uncaught exceptions.

Table 6.5 summarizes those approaches with respect to test generation approach.

Table 6.5: Related Work on automatic test generation.

	Test generation approach
ART [14]	Based on the distance between test cases
ARTOO [20]	Based on distance – considering properties from object-oriented programs
JET [15]	Based on the feedback – from genetic algorithms
EvoSuite [45]	Based on evolutionary search
PUT [123]	Parameterized tests
Symstra [130]	Symbolic execution
Randoop [91]	Based on feedback from the execution of sequence being constructed

Chapter 7

Concluding Remarks

In this thesis, we address the problem of fostering Design by Contract [74] by exploiting the relationship between code commentary and contracts. For this purpose, we propose and implement a new way of writing comments in the context of Java programs (CONTRACTJDOC); we also propose and implement an approach for automatically suggesting contracts based on natural language tagged-comments (CONTRACTSUGGESTOR).

CONTRACTJDOC allows the use of Design by Contract in a format closed to traditional Javadoc comments (see Chapters 2 and 3). In addition, we developed and evaluated an approach for automatically suggesting contracts based on natural language tagged-comments (Chapter 4). We also performed conformance checking of the contracts automatically generated and performed conformance checking of contracts from a general point of view (Chapter 5).

Moreover, by means of our approaches: CONTRACTJDOC, CONTRACTSUGGESTOR, JMLOK2, and CONTRACTOK we are able to perform conformance checking and detect inconsistencies in three levels of formality: informal – by generating contracts from Javadoc plain comments; semiformal by applying CONTRACTJDOC to Java systems; and formal by analyzing JML and Code Contracts systems.

We now summarize the main findings of this thesis (Sections 7.1 to 7.3) and present prospects for future work (Section 7.4).

7.1 CONTRACTJDOC

CONTRACTJDOC provides a new way for documenting source code in which a developer can specify contracts by adding boolean-valued expressions into brackets and using specific tags into the Javadoc. The CONTRACTJDOC approach tries to fulfill the gap between informal documentation (such as JAVADOC) and formal specification (such as JML [68]) by enabling the developer to write contracts by using default tags from JAVADOC (such as `@param`) and some new tags (such as `@inv`) in a controlled way. CONTRACTJDOC supports preconditions, postconditions, and invariants.

We performed three studies [129] for evaluating our extension to the Javadoc tagging system, the approaches for detecting nonconformances and our classification model. First, we evaluate CONTRACTJDOC by means of three studies: (a) a case study aiming to serve as a proof of concept (Section 3.1); (b) an empirical study with Java developers (Section 3.2); (c) a survey with Java developers for investigating the comprehensibility on three alternatives for specifying behavior in a Java interface — Javadoc, JML, and CONTRACTJDOC (Section 3.3). With the case study, we are able to write contracts to six open-source Java projects previously annotated with Javadoc comments in natural language. The systems totalize 190,655 lines of code and we wrote a total of 3,994 contract clauses. Besides helping us to evaluate the applicability of our language and its compiler, this study allowed us to detect inconsistencies between Javadoc comments and source code, highlighting the importance of being able to check the comments runtime: without checking, those inconsistencies will remain undiscovered.

Twenty-four Java developers with different experience levels participated in our empirical study. As results, 83% of the developers were able to perform the required task without perceiving difficulties. When grouping the results by documenting approach, they considered Javadoc as the less complicated for performing the trial, followed by CONTRACTJDOC.

The comprehensibility survey confirmed the results from the empirical study: developers tend to find Javadoc comments more understandable than JML or CONTRACTJDOC. Thirty-eight percent of the survey respondents chose Javadoc as the most understandable approach regarding interface's behavior, others 32% chose ContractJDoc and 18% chose the same understanding level for all documenting approaches. When asked about the most understandable approach in a general context, 51% answered Javadoc and 29% CONTRACTJDOC; and for 13% the understanding is the same for all of them.

In summary, CONTRACTJDOC is intermediate between Javadoc and JML in terms of comprehensibility. Moreover, survey results did not significantly differ for CONTRACTJDOC and Javadoc, which is promising, since contracts are regarded as hard to read. Therefore, CONTRACTJDOC is an approach that can foster contracts adoption, helping to improve software quality. The approach may benefit freshman developers by providing a rich and accurate description of the systems under development without requiring them to learn/understand a formal language, such as JML [68] or Eiffel [75].

7.2 CONTRACTSUGGESTOR

CONTRACTSUGGESTOR provides an approach for automatically suggesting contracts by analyzing natural language tagged-comments. In this thesis, we applied CONTRACTSUGGESTOR for generating contracts for two properties: non-null and relational. For this, we manually classified every tagged-comment from 21 open-source Java projects summing up to 134,246 comment instances. In order to suggest contracts, we applied supervised machine learning algorithms over the classified comments (Section 4.2).

After generating and checking the contracts (Section 4.3), we applied JMLOK2 over four systems for conformance checking them. As a result, the tool detected six nonconformances: five related to non-null property and one related to relational properties – greater than or equal to (Section 5.1). All nonconformances detected were reported to systems developers, however, they did not answer our contact.

Even in systems in which CONTRACTSUGGESTOR generated only a few contracts: four contracts in Apache Collections, those contracts enabled JMLOK2 for detecting potential nonconformances, highlighting the usefulness of CONTRACTSUGGESTOR. In addition, as CONTRACTSUGGESTOR turns the use of contracts transparent to developers, they will benefit from conformance checking of the code commentary without needing to change their routine of daily work: write source code and natural language comments.

We also performed studies to increase the confidence on the results by JMLOK2 in checking contracts: we run JMLOK2 over 12 Java/JML systems in order to detect and manually classify nonconformances (Section 5.3). JMLOK2 detected and automatically suggested likely causes for 119 nonconformances. From those, 51 are invariant problems and 47 are postcondition problems. With respect to manual classification, Weak precondition and Code error are the most commons: 51 and 38, respectively. We also collect B and D metrics [79] as a proxy for characterizing the difficult for detecting nonconformances in JML programs: 2.68 for B and 2.44 for D, indicating the necessity of at least three changes in average for detecting the nonconformance and the inspection of at least three methods for establishing a likely cause. We also applied CONTRACTOK over 12 open-source systems (Section 5.3). CONTRACTOK detected 63 nonconformances, being 28 precondition errors and 25 postcondition errors. Regarding the likely causes, as for JML, Weak precondition and Code error are the most commons: 40 and 18, respectively. The metrics B and D for the are 6.38 for B and 4.08 for D, indicating the necessity of at more than six changes in average for detecting the nonconformance and the inspection of at least four methods for establishing a likely cause. According to these metrics, detecting nonconformances in Code Contracts systems is harder than in JML systems; in addition, establishing a likely cause was also harder in Code Contracts systems. Therefore, for detecting a nonconformance in Code Contracts systems the developer will need a test more structured than in JML systems, and for classifying s/he will spend more time on analyzing the Code Contracts project's source code and contracts in comparison to JML.

7.3 Review of the Contributions

In this work we provide the following contributions:

- We propose and evaluate a new approach for writing contracts (CONTRACTJDOC)
    - We performed a case study in which we applied CONTRACTJDOC to six open-source Java projects and generated 3,994 contract clauses;
    - We carried out an experimental study and a survey with Java developers for evaluating the readability of CONTRACTJDOC. As results, CONTRACTJDOC was considered intermediate between plain Javadoc and JML;
- We propose and evaluate a contract generation approach (CONTRACTSUGGESTOR)

  - We manually labeled 134,246 comment instances. Then, we created two datasets: one with respect to the non-null property containing all instances, and one with respect to the relational properties containing 1,808 instances;
  - We evaluated the generated contracts with JMLOK2: we applied the tool for conformance checking four Java systems with contracts automatically generated, and it detected six nonconformances;

- We leverage JMLOK2 for C#/Code Contracts context by means of CONTRACTOK [82] and we performed a case study with both tools over 24 systems (12 for each language) in which the tools detected 182 nonconformances. We also manually established likely causes for those nonconformances [80; 81]. The case study with CONTRACTOK over 12 systems was published as "Nonconformance between Programs and Contracts: A Study on C#/Code Contracts Open Source Systems" into SAC'2017 [82].

7.4 Future Work

Concerning CONTRACTJDOC approach, we plan to integrate CONTRACTJDOC and CONTRACTSUGGESTOR for supporting autocomplete when writing the Javadoc comments. For this purpose, we may use the Jaro-Winkler [59; 128] string distance and deep learning [50] for autocompleting the comments.

We also intend to perform new experimental studies for evaluating the applicability of CONTRACTJDOC in the practice of software development. Moreover, we will investigate the relationship between system tests and code commentary by performing new case studies in order to verify consistency between test suites and code commentary.

For dealing with one of CONTRACTSUGGESTOR limitations (Section 4.4), we plan to consider comments not tagged in order to be able to generate invariants based on class fields comments. A way of reducing this limitation is to improve *extract_comments* for collecting all comments in the Java source code that are delimited by the /** ... */ delimiters and update the datasets for considering those comments. Then, the approach for generating contracts will need to use the information from comments for generating contract invariants. For instance, consider the Source Code 7.1. Line 2 - 4 present a comment concerning a class field `name`, declaring this field as being non-null. By updating CONTRACTSUGGESTOR for considering not tagged-comments the approach will be able to classify this text as related to non-null and the approach for generating contracts will generate an invariant for this class.

Source Code 7.1: A code commentary for an invariant.

```java
public class Product{
  /**
  * The name of a product, must not be null
  */
  protected String name;
  //...
}
```

Moreover, by performing new analysis in the datasets we already have (or even getting new systems) we may discover new patterns for contracts in the natural language texts, e.g. patterns for dealing with mathematical intervals such as “must be value1 <= varName >= value2” and “must be in the range [value1 .. value2]”; these comments can produce contracts like *value1 <= varName && varName <= value2*. In addition, we intend to apply our contract generator in conjunction with JMLOK2 for generating contracts and conformance checking Java APIs. We also plan to apply CONTRACTSUGGESTOR for more systems largely used such as those from Apache and Mozilla foundations.

Another option for future work is to define a natural controlled language for being used in Javadoc comments and to suggest contracts based on the defined constructs. This approach will enable the generation of contracts for more properties and the possibility to mitigate false positives into the generated contracts.

Concerning nonconformance detection, we intend to improve the detection phase of our tools (JMLOK2 and CONTRACTOK) by using other test generation approaches such as EvoSuite [45] or parameterized unit tests [123] and compare the effectiveness of those approaches with the approach currently used. For analyzing the effect of changing the test generation engine, we will perform new experiments with JMLOK2 and CONTRACTOK tools over the same units already evaluated and compare the detection results with those achieved by Randoop-based approaches. We also plan to formally define a conformance relation for the purpose of dynamically conformance checking of contract-based programs.

Concerning conformance checking of contract-based programs, we identified that there are no automated tools that consider refactoring [44] in this context; and the preservation of behavior become a property hard to verify [72]. There are only formal approaches, such as presented by Freitas [46], 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 the construction of reliable programs. This thesis results can also be extended for considering the problem of contracts and programs refinement [3; 19]. Programs are composed of a tuple (Contract, Code), in which *Code* must be in conformance with *Contract*. When a program is refined to (Contract', Code') such that Contract' refines Contract, it is likely that Code' refines Code. In other words, the refinement of contracts implies the refinement of codes – even though the refinement of code may not imply the refinement of contracts.

Bibliography

[1] S. Acharya and V. Pandya. Bridge between Black Box and White Box - Gray Box Testing Technique. *International Journal of Electronics and Computer Science Engineering*, pages 175–185, December 2012.

[2] K. K. Aggarwal, Y. Singh, and J. K. Chhabra. An integrated measure of software maintainability. In *Annual Reliability and Maintainability Symposium*, pages 235–241, 2002.

[3] R.-J. J. Back, A. Akademi, and J. V. Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[4] 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.

[5] 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.

[6] 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 *International Conference on Formal Methods for Components and Objects*, pages 152–174. Springer-Verlag, 2007.

[7] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer-Verlag, 2007.

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

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

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

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

[12] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, R. 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] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[14] T. Chen, H. Leung, and I. Mak. Adaptive Random Testing. In *Asian Computing Science conference on Advances in Computer Science: dedicated to Jean-Louis Lassez on the Occasion of His 5th Cycle Birthday*, volume 3321, pages 320–329. Springer Berlin Heidelberg, 2005.

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

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

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

[18] M. Christakis, P. Müller, and V. Wüstholz. Synthesizing Parameterized Unit Tests to Detect Object Invariant Violations. In *International Conference on Software Engineering and Formal Methods*, pages 65–80, 2014.

[19] A. Cimatti, R. Demasi, and S. Tonetta. Tightening a Contract Refinement. In R. De Nicola and E. Kühn, editors, *Software Engineering and Formal Methods*, pages 386–402. Springer International Publishing, 2016.

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

[21] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[22] D. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In *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 *International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 108–128. Springer-Verlag, 2004.

[24] A. Corazza, V. Maggio, and G. Scanniello. Coherence of comments and method implementations: a dataset and an empirical investigation. *Software Quality Journal*, Nov 2016.

[25] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *J. Mach. Learn. Res.*, 7:551–585, Dec. 2006.

[26] O. Dahl, E. Dijkstra, and C. Hoare. *Structured Programming*. Academic Press Ltd., 1972.

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

[28] G. A. G. L. David Cok, John L. Singleton. Openjml. https://github.com/OpenJML/OpenJML, 2018 (accessed April 24, 2018).

[29] J. developers. Jfreechart. https://sourceforge.net/projects/jfreechart/files/1.%20JFreeChart/1.0.13/jfreechart-1.0.13.zip/download?_test=goal, 2018 (accessed April 24, 2018).

[30] D. Dillman, J. Smyth, and L. Christian. *Internet, Phone, Mail, and Mixed-Mode Surveys: The Tailored Design Method*. Wiley Publishing, 4th edition, 2014.

[31] A. Downey. *Think Python*. O'Reilly Media, 2012.

[32] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224. ACM, 1999.

[33] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 69(1-3):35–45, Dec. 2007.

[34] H. Estler, C. Furia, M. Nordio, M. Piccioni, and B. Meyer. Contracts in practice. In *International Symposium on Formal Methods*, pages 230–246. Springer-Verlag New York, Inc., 2014.

[35] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *ACM Symposium on Applied Computing*, pages 2103–2110. ACM, 2010.

[36] M. Fähndrich and F. Logozzo. Clousot: Static contract checking with Abstract Interpretation. In *International Conference on Formal Verification of Object-oriented Software*, pages 10–30. Springer-Verlag, 2010.

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

[38] E. Foundation. Swt: The standard widget toolkit. https://www.eclipse.org/swt/, 2018 (accessed April 24, 2018).

[39] T. A. S. Foundation. Apache commons collections. https://archive.apache.org/dist/commons/collections/source/commons-collections-3.2.1-src.tar.gz, 2018 (accessed April 24, 2018).

[40] T. A. S. Foundation. Apache commons math. http://commons.apache.org/proper/commons-math/download_math.cgi, 2018 (accessed April 24, 2018).

[41] T. A. S. Foundation. Apache log4j. https://svn.apache.org/repos/asf/logging/log4j/trunk, 2018 (accessed April 24, 2018).

[42] T. A. S. Foundation. Apache lucene. http://archive.apache.org/dist/lucene/java/lucene-2.9.3-src.zip, 2018 (accessed April 24, 2018).

[43] T. A. S. Foundation. Apache xalan. http://svn.apache.org/repos/asf/xalan/java/trunk, 2018 (accessed April 24, 2018).

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

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

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

[47] L. Friendly. *The Design of Distributed Hyperlinked Programming Documentation*, pages 151–173. Springer London, 1996.

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

[49] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè. Automatic Generation of Oracles for Exceptional Behaviors. In *International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 213–224. ACM, 2016.

[50] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

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

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

[53] M. L. Heuer. Dishevelled. `https://github.com/heuermh/dishevelled`, 2018 (accessed April 24, 2018).

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

[55] T. Hoare. Null references: The billion dollar mistake. Talk at QCon London, 2009. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare.

[56] R. E. Holger Brands. Glazed lists. `https://github.com/glazedlists/glazedlists/releases/tag/glazedlists-1_8_0`, 2018 (accessed April 24, 2018).

[57] iText Software team. itext. `https://svn.code.sf.net/p/itext/code/trunk/`, 2018 (accessed April 24, 2018).

[58] J. v. Z. J. S. S. M. v. d. B. d. G. D. R. T. O. James Strachan, Robert Burrell Donkin. Commons betwixt. `http://commons.apache.org/dormant/commons-betwixt/`, 2018 (accessed April 24, 2018).

[59] M. A. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989.

[60] Joda.org. Joda time. `https://github.com/JodaOrg/joda-time`, 2018 (accessed April 24, 2018).

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

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

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

[64] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001.

[65] D. Kramer. Api documentation from source code comments: A case study of javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation*, pages 147–153. ACM, 1999.

[66] J. Landis and G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

[67] G. Leavens, A. Baker, and C. Ruby. JML: A Notation for Detailed Design. *Behavioral Specifications of Businesses and Systems*, pages 175–188, 1999.

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

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

[70] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[71] R. G. J. V. C. M. E. Matthew Horridge, Tania Tudorache. Webprotégé. `https://github.com/protegeproject/webprotege`, 2018 (accessed April 24, 2018).

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

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

[74] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

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

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

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

[78] B. Meyer, A. Kogtenkov, and E. Stapf. *Avoid a Void: The Eradication of Null Dereferencing*, pages 189–211. Springer London, 2010.

[79] A. Milanez. Enhancing conformance checking for contract-based programs. Master's thesis, Federal University of Campina Grande, 2014.

[80] A. Milanez. A Case Study on Classifying Nonconformances in Java/JML Programs. Technical Report SPLAB-2015-005, Software Practices Laboratory, Federal University of Campina Grande, Nov 2015.

[81] A. Milanez. A Baseline for Classifying Nonconformances in C#/Code Contracts Programs. Technical Report SPLAB-2016-002, Software Practices Laboratory, Federal University of Campina Grande, May 2016.

[82] A. Milanez, B. Lima, J. Ferreira, and T. Massoni. Nonconformance between Programs and Contracts: A Study on C#/Code Contracts Open Source Systems. In *ACM Symposium on Applied Computing*, pages 1219–1224. ACM, 2017.

[83] A. Milanez, T. Massoni, and R. Gheyi. Categorizing Nonconformances Between Programs and Their Specifications. In *Brazilian Workshop on Systematic and Automated Software Testing*, 2013.

[84] A. Milanez, D. Sousa, T. Massoni, and R. Gheyi. JMLOK2: A tool for detecting and categorizing nonconformances. In *Brazilian Conference on Software: Theory and Practice (CBSoft) - Tools Session*, pages 69–76, 2014.

[85] A. F. Milanez. Fostering design by contract by exploiting the relationship between code commentary and contracts. `http://massoni.computacao.ufcg.edu.br/home/jmlok/milanez-thesis`, 2018.

[86] D. Narayanan. Abc-music-player. https://github.com/deepakn94/ABC-Music-Player, 2018 (accessed April 24, 2018).

[87] S. Nawar and A. M. Mouazen. Comparison between random forests, artificial neural networks and gradient boosted machines methods of on-line vis-nir spectroscopy measurements of soil total nitrogen and total carbon. *Sensors*, 17(10), 2017.

[88] D. Nielsen. Tree boosting with xgboost why does xgboost win "every" machine learning competition? Master's thesis, Norwegian University of Science and Technology, 2016.

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

[90] C. Pacheco, S. Lahiri, and T. Ball. Finding Errors in .Net with Feedback-directed Random Testing. In *International Symposium on Software Testing and Analysis*, pages 87–96. ACM, 2008.

[91] C. Pacheco, S. Lahiri, M. 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.

[92] Pase. Simpleshop. https://github.com/pase/simpleshop, 2018 (accessed April 24, 2018).

[93] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[94] Y. Pei, C. Furia, M. Nordio, and B. Meyer. Automatic program repair by fixing contracts. In *International Conference on Fundamental Approaches to Software Engineering*, pages 246–260. Springer-Verlag New York, Inc., 2014.

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

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

[97] G. R Willi. Oop aufgabe3. `https://github.com/rwilli/aufgabe3`, 2018 (accessed April 24, 2018).

[98] J. Radatz, F. Jay, R. Mayer, S. Gloss-Soler, M. Migliaro, J. Daly, and A. Salem. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[99] A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.

[100] H. Rebêlo, G. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. Zimmerman, M. Cornélio, and T. Thüm. Aspectjml: Modular specification and runtime checking for crosscutting contracts. In *International Conference on Modularity*, pages 157–168. ACM, 2014.

[101] H. Rebêlo, R. Lima, M. Cornélio, G. Leavens, A. Mota, and C. Oliveira. Optimizing JML Features Compilation in ajmlc Using Aspect-Oriented Refactorings. In *Brazilian Symposium on Programming Languages*, pages 117–130, 2009.

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

[103] Z. Reitermanov. Data splitting. In *WDS'10 Proceedings of Contributed Papers*, pages 31–36, 01 2010.

[104] M. Riedel. Jenerics. `https://github.com/mriedel/Jenerics`, 2018 (accessed April 24, 2018).

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

[106] D. Rosenblum. Towards a Method of Programming with Assertions. In *International Conference on Software Engineering*, pages 92–104. ACM, 1992.

[107] R. Sanasam, H. Murthy, and T. Gonsalves. Feature selection for text classification based on gini coefficient of inequality. In H. Liu, H. Motoda, R. Setiono, and Z. Zhao, editors, *Proceedings of the Fourth International Workshop on Feature Selection in Data Mining*, volume 10 of *Proceedings of Machine Learning Research*, pages 76–85, 2010.

[108] 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.

[109] T. Schiller, K. Donohue, F. Coward, and M. Ernst. Case Studies and Tools for Contract Specifications. In *International Conference on Software Engineering*, pages 596–607. ACM, 2014.

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

[111] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[112] A. Shtof, A. Agathos, Y. Gingold, A. Shamir, and D. Cohen-Or. Geosemantic Snapping for Sketch-Based Modeling. *Computer Graphics Forum*, 32(2):245–253, 2013.

[113] X. Shu and D. Yao. Program anomaly detection: Methodology and practices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1853–1854. ACM, 2016.

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

[115] M. Staats, M. Whalen, and M. Heimdahl. Programs, Tests, and Oracles: The Foundations of Testing Revisited. In *International Conference on Software Engineering*, pages 391–400. ACM, 2011.

[116] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API Documentation. In *International Conference on Software Engineering*, pages 643–652. ACM, 2014.

[117] L. Tan, Y. Zhou, and Y. Padioleau. aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs. In *International Conference on Software Engineering*, pages 11–20. ACM, 2011.

[118] S. Tan, D. Marinov, L. Tan, and G. Leavens. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *International Conference on Software Testing, Verification and Validation*, pages 260–269, April 2012.

[119] C. team. Cobertura. `https://github.com/cobertura/cobertura`, 2018 (accessed April 24, 2018).

[120] E. S. team. Eclipse smarthome™ project. `https://github.com/eclipse/smarthome`, 2018 (accessed April 24, 2018).

[121] J.-M. team. Joda-money. `https://github.com/JodaOrg/joda-money`, 2018 (accessed April 24, 2018).

[122] N. Tillmann and J. de Halleux. Pex-White Box Test Generation for .NET. In *International Conference on Tests and Proofs*, pages 134–153, 2008.

[123] N. Tillmann and W. Schulte. Parameterized unit tests. *SIGSOFT Software Engineering Notes*, 30(5):253–262, Sept. 2005.

[124] M. Torchiano. Documenting pattern use in Java programs. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 230–233, 2002.

[125] J. Tretmans. A Formal Approach to Conformance Testing. In *International Workshop on Protocol Test Systems VI*, pages 257–276. North-Holland Publishing Co., 1994.

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

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

[128] W. E. Winkler. The state of record linkage and current research problems. Technical Report Statistical Research Report Series RR99/04, U.S. Bureau of the Census, Washington, D.C., 1999.

[129] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[130] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A Framework for Generating Object-oriented Unit Tests Using Symbolic Execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381. Springer-Verlag, 2005.

[131] E. Yourdon. *Techniques of Program Structure and Design*. Prentice Hall PTR, 1st edition, 1986.

[132] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin. Automatic Model Generation from Documentation for Java API Functions. In *International Conference on Software Engineering*, pages 380–391. ACM, 2016.

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

[134] H. Zou, J. Zhu, S. Rosset, and T. Hastie. Multi-class adaboost. *Statistics and its Interface*, 2:349–360, 2009.

Appendix A

Evaluating the supervised machine learning algorithms

Below we present the results of our evaluation of the supervised machine learning algorithms available in Scikit-learn [93] package.

Results for the evaluation of the ML algorithms for supervised learning

Training set: 41,763 comment instances

Testing set: 13,921 comment instances

For this evaluation we considered a subset of the Non-null dataset composed of 13 projects:
From ContractJDoc Case Study:

- ABC-Music Player
- Dishevelled
- Jenerics
- OOP AufGabe
- SimpleShop
- WebProtégé

From @Comment paper:

- Collections
- GlazedLists
- JFreeChart
- JodaTime
- Log4J
- Lucene
- Xalan

Those projects totalize 55,684 comment instances: 5,961 non-null and 49,723 others.

AdaBoost. An AdaBoost classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

```
        Accuracy: 0.99913799296
         precision   recall  f1-score   support
  non-null      1.00      0.99      1.00      1449
    others      1.00      1.00      1.00     12472
```

```
avg / total       1.00      1.00      1.00     13921
```

Passive Aggressive Classifier: The passive-aggressive algorithms are a family of algorithms for large-scale learning. They are similar to the Perceptron in that they do not require a learning rate. However, contrary to the Perceptron, they include a regularization parameter C.

```
            Accuracy: 0.99877882336
             precision    recall  f1-score   support
   non-null       0.99      1.00      0.99      1484
     others       1.00      1.00      1.00     12437

avg / total       1.00      1.00      1.00     13921
```

Multi-layer Perceptron classifier. This model optimizes the log-loss function using LBFGS or stochastic gradient descent. The advantages of Multi-layer Perceptron are the capability to learn non-linear models; capability to learn models in real-time (online learning).

```
            Accuracy: 0.99877882336
             precision    recall  f1-score   support
   non-null       1.00      0.99      0.99      1449
     others       1.00      1.00      1.00     12472

avg / total       1.00      1.00      1.00     13921
```

Linear Support Vector Classification: Similar to SVC with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples. This class supports both dense and sparse input and the multiclass support is handled according to a one-vs-the-rest scheme.

```
            Accuracy: 0.99870698944
             precision    recall  f1-score   support
   non-null       0.99      0.99      0.99      1465
     others       1.00      1.00      1.00     12456

avg / total       1.00      1.00      1.00     13921
```

Voting Classifier. The idea behind the VotingClassifier is to combine conceptually different machine learning classifiers and use a majority vote or the average predicted probabilities (soft vote) to predict the class labels. Such a classifier can be useful for a set of equally well-performing model in order to balance out their individual weaknesses. By using: linear_model.PassiveAggressiveClassifier(C=1.0),
ensemble.AdaBoostClassifier(n_estimators=100), and svm.LinearSVC(random_state=0), the result is presented below.

```
        Accuracy: 0.99863515552
         precision    recall  f1-score   support
  non-null     0.99      0.99      0.99      1449
    others     1.00      1.00      1.00     12472

avg / total     1.00      1.00      1.00     13921
```

A Bagging classifier. A Bagging classifier is an ensemble meta-estimator that fits the base classifiers each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

```
        Accuracy: 0.9985633216
         precision    recall  f1-score   support
  non-null     0.99      1.00      0.99      1471
    others     1.00      1.00      1.00     12450

avg / total     1.00      1.00      1.00     13921
```

Gradient Boosting for classification. Gradient Tree Boosting is a generalization of boosting to arbitrary differentiable loss functions. GBRT is an accurate and effective off-the-shelf procedure that can be used for both regression and classification problems. Gradient Tree Boosting models are used in a variety of areas including Web search ranking and ecology.

```
        Accuracy: 0.998419653761
         precision    recall  f1-score   support
  non-null     1.00      0.99      0.99      1449
    others     1.00      1.00      1.00     12472
```

```
avg / total       1.00      1.00      1.00     13921
```

DecisionTreeClassifier is a class capable of performing multi-class classification on a dataset.

```
            Accuracy: 0.998347819841
             precision    recall  f1-score   support
  non-null       0.99      0.99      0.99      1471
    others       1.00      1.00      1.00     12450

avg / total       1.00      1.00      1.00     13921
```

Perceptron: The Perceptron is another simple algorithm suitable for large-scale learning. By default: It does not require a learning rate. It is not regularized (penalized). It updates its model only on mistakes.

```
            Accuracy: 0.998060484161
             precision    recall  f1-score   support
  non-null       0.99      0.99      0.99      1484
    others       1.00      1.00      1.00     12437

avg / total       1.00      1.00      1.00     13921
```

Extra Trees Classifier. An extra-trees classifier. This class implements a meta estimator that fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting.

```
            Accuracy: 0.997701314561
             precision    recall  f1-score   support
  non-null       0.99      0.98      0.99      1471
    others       1.00      1.00      1.00     12450

avg / total       1.00      1.00      1.00     13921
```

Stochastic Gradient Descent Classifier: Stochastic gradient descent is a simple yet very efficient approach to fit linear models. It is particularly useful when the number of samples (and the number of features) is very large.

```
        Accuracy: 0.997701314561
         precision    recall  f1-score   support
  non-null       0.99      0.99      0.99      1484
    others       1.00      1.00      1.00     12437

avg / total       1.00      1.00      1.00     13921
```

Random Forest Classifier. In random forests, each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set. In addition, when splitting a node during the construction of the tree, the split that is chosen is no longer the best split among all features. Instead, the split that is picked is the best split among a random subset of the features. As a result of this randomness, the bias of the forest usually slightly increases (with respect to the bias of a single non-random tree) but, due to averaging, its variance also decreases, usually more than compensate for the increase in bias, hence yielding an overall better model.

```
        Accuracy: 0.997054809281
         precision    recall  f1-score   support
  non-null       0.99      0.98      0.99      1471
    others       1.00      1.00      1.00     12450

avg / total       1.00      1.00      1.00     13921
```

Bernoulli Naive Bayes. Like MultinomialNB, this classifier is suitable for discrete data. The difference is that while MultinomialNB works with occurrence counts, BernoulliNB is designed for binary/boolean features.

```
        Accuracy: 0.960563177933
         precision    recall  f1-score   support
  non-null       0.75      0.95      0.84      1471
    others       0.99      0.96      0.98     12450

avg / total       0.97      0.96      0.96     13921
```

Gaussian Naive Bayes. GaussianNB implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian.

```
            Accuracy: 0.65713669995
             precision    recall  f1-score   support
   non-null       0.23      0.94      0.37      1471
     others       0.99      0.62      0.76     12450

avg / total       0.91      0.66      0.72     13921
```

Multinomial Naive Bayes. The multinomial Naive Bayes classifier is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work.

```
            Accuracy: 0.952158609295
             precision    recall  f1-score   support
   non-null       0.93      0.59      0.72      1471
     others       0.95      0.99      0.97     12450

avg / total       0.95      0.95      0.95     13921
```