Investigating the Use of JML Contracts

Alysson F. Milanez
Department of Engineering and Technology
Federal Rural University of the Semi-Arid
(UFERSA)
Pau dos Ferros, Brazil
alysson.milanez@ufersa.edu.br

Igor N. S. Ataíde
Department of Computing Systems
Federal University of Campina Grande
(UFCG)
Campina Grande, Brazil
igor.ataide@ccc.ufcg.edu.br

Tiago L. Massoni
Department of Computing Systems
Federal University of Campina Grande
(UFCG)
Campina Grande, Brazil
massoni@dsc.ufcg.edu.br

***Abstract*—Design by Contract (DBC) is a methodology from formal methods research that aims the construction of quality software. With DBC the contracts become assertions that can be checked at runtime, fostering reliability for developers. In this work, we investigate the use of JML (Java Modeling Language) contracts by means of seven open-source projects and checked whether the contract used has some relationship with the kind of nonconformance that occurs in each project. For the projects considered, the most common contract types were precondition and postcondition. This result suggests developers apparently prefer to write pre- and postcondition clauses in comparison with invariants. Concerning the relationship between number of contract clauses and number of detected nonconformances, we found a moderate negative correlation; indicating that contracts with more clauses do not imply more nonconformances. This result may indicate developers who write bigger contracts tend to follow those contracts more closely.**

***Palavras-chaves*—contract-based programs, JML, nonconformances.**

I. Introduction

In the context of contract-based programs [1], it is interesting to understand how developers use the kinds of contracts available, such as preconditions, postconditions and invariants. This is important because 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 as discussed by Milanez [2].

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 [2].

There are some studies that have tried to investigate the usage of contracts, as Chalin [3] which examined in languages with built-in support for DBC if practitioners tend to write more contracts or Estler et al. [4] that addresses practical questions of the usage of contracts.

Chalin study results indicate that programmers using Eiffel (the only active language with built-in support for DBC) tend to write assertions in a proportion that is higher than for other languages. Estler et al. found a significant use of contracts more than 33% and those contracts are stable over time; furthermore, there is no strong preference for certain kind of contract: however, preconditions tend to be larger than postconditions.

Those previous works did not consider the relationship between the kind of contract developers use and the nonconformances that can be detected.

In this paper, we propose a qualitative and quantitative analysis of the use of contracts and we perform a case study with seven JML (JML is a DBC-enabling for Java) open source projects for analyzing the relationship between nonconformances detected and contracts use.

In all the seven projects considered, the most common contract types were precondition and postcondition. This result suggests developers apparently prefer to write pre- and postcondition clauses in comparison with invariants.

We also investigate the relationship between number of contract clauses and number of detected nonconformances and found a moderate negative correlation; indicating that contracts with more clauses do not imply more nonconformances. This result may indicate developers who write bigger contracts tend to follow those contracts more closely; however, more studies are needed for further evidence about this.

The paper is structured as follows. Section II presents the theoretical background needed to understand this work. Next, Section III describes the research we performed. Finally, we summarize in Section IV the main findings of this work and suggestions for future work.

II. Background

This section provides the background needed to understand the topics discussed through the current work. Section II-A discusses the DBC methodology; then, Section II-B uses an example for present the idea of a nonconformance between source code and contracts. Finally, Section II-C presents the metrics used in the study performed.

A. Design by Contract

Design by Contract [1] (DBC) is a methodology from formal methods research [5] that aims the construction of quality software. DBC is a direct descendant from Hoare's triples [6] – **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): "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) [1], [7]. In this context, clients must satisfy *preconditions* before calling a supplier; suppliers in their turn, have to provide some guarantees over their results (*postconditions*) [7]. Both clients and suppliers may have contracts with respect to their fields – e.g. establishing the range of valid values – (*invariants*) [7].

As discussed by Milanez [2], DBC enables developers to detect design errors during the software development phase, since the design decisions are written in the form of contracts into the source code, thus tools can be used for checking whether the design decisions are being fulfilled. Furthermore, inconsistencies related to the misunderstanding of the requirements can be detected earlier than in scenarios without the use of contracts.

Since DBC contracts become assertions, they can be checked at runtime, fostering reliability for developers. The 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. In addition, contracts ruling the behavior of the underlying code provide additional data for conformance verification [2].

In the context of Java development, the Java Modeling Language (JML) [8] is a DBC-enabling notation (and corresponding toolset), with contracts as comments within Java code. As presented by Milanez [2], 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 [9] with the specification model-based approach from Larch family of programming languages [10], and some elements from the calculus of refinement.

B. Nonconformances

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 [2].

In addition, JML has a special kind of constraint: 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 [2].

Class `Counter` (Listing 1) declares a constructor and two methods: one for updating values and one for resetting values – visibility is omitted, for simplicity. The invariant in `Counter` enforces that field `count` must be in the range [0, MAX]. The `\old` clause used in the postcondition refers to pre-state value of `count`.

Listing 1. Counter class

```
class Counter{
  final int MAX = 3;
  int count;
  //@ invariant 0 <= count && count <= MAX;
  Counter() {
    count = 1;
  }
  //@ ensures !b || (count == \old(count+1));
  void updateCount(boolean b){
    if(b){
      count++;
    }
  }
  //@ ensures count == 0;
  void resetCount(){
    count= 0;
  }
}
```

In addition to the contracts from our motivating example, JML supports history constraints, which we call constraints for short, are related to invariants. But whereas invariants are predicates that should hold in all visible states, history constraints are relationships that should hold for the combination of each visible state and any visible state that occurs later in the program’s execution. Constraints can therefore be used to constrain the way that values change over time [11].

The class is not in conformance with its contracts, as it presents one nonconformance that can only be detected with a sequence of at least three calls to `updateCount`, with parameter b = *true*. In Listing 2, a test case reveals this problem.

Listing 2. A test case for Counter class

```
Counter c = new Counter();
c.updateCount(true);
c.updateCount(true);
c.updateCount(true);
```

Nonconformances between contract and implementation may be subtle to detect. Regardless of where the bug is located (contract or code, or both); the failure may only arise within a sequence of calls to two or more methods, called in a particular order. Method `updateCount` does not have an explicit precondition, the likely cause suggested is *Weak Precondition*. This problem may be solved by adding a precondition to `updateCount`, relating the method parameter with the current value of `count`.

C. Contracts Complexity

For quantifying contract clauses, we follow Estler et al. [4] approach, in which the total number of contract clauses (#CC) is a proxy for contract complexity. In addition, for quantifying code lines (LOC) – we consider only real source code lines, blank lines, closing brackets and comments are not considered. The *CCo* metric (Equation 1) is the ratio between #CC and LOC. The implementation for counting contract clauses and lines of code is available online.[1]

$$CCo(x) = \frac{\#CC(x)}{LOC(x)} \tag{1}$$

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

Concerning the motivating example (Section II-B), *CCo* metric is calculated as follows: there are 10 code lines, so LOC = 10; there are two invariant clauses and two postcondition clauses, therefore, #CC = 4; thus, *CCo* = 4/10, in other words, *CCo* metric = 0.4. We have 4 contract clauses and 10 lines of code.

III. Methodology

This section presents the study performed. Section III-A shows the research question that motivated the present work; then, the experimental design is presented on Section III-B; next, Section III-C describes the main findings and Section III-D summarizes the discussions; finally, Section III-E presents the limitations of the present work.

A. Research Question

In order to investigate contracts practice in Java/JML projects and discover relations between contracts and detected nonconformances, we formulate the following research question:
RQ. Are there any relationship between contracts complexity, as measured by *CCo* metric (Equation 1) and the number of detected nonconformances?

B. Experimental Design

To answer this question, we select seven open source projects whose nonconformances where manually mapped during previous study [2]. The JML projects (that we will call experimental units henceforth) are: `Bank` a representation of a real bank is presented in the KeY approach book [12], `Bomber` [13] a mobile game, `HealthCard` [14] an application that manages medical appointments into smart cards, `JAccounting` an accounting system, also a case study from the *ajml* compiler project [13].

`Mondex` [15] is a translation from an original Z specification, developed in the Verified Software Repository[2] context, `Samples` is a set composed by example programs for educational purposes, written by JML specialists, and `TransactedMemory` [16] is a specific feature of the Javacard API. All seven projects are available online[3].

For those projects, we collected two metrics: #CC - the number of contract clauses and LOC - lines of code. Those metrics are used for establishing the *CCo* metric (defined in Section II-C, Equation 1).

We also used the nonconformances (NCs) detected by JmlOk2 tool [2], [17]. These seven selected projects totalize 103 nonconformances.

C. Results

We summarize in Table I all information collected for each experimental unit. With respect to lines of code (LOC), values vary from 655 to 6,648; the *CCo* values vary from 0.02 to 0.48. When grouped by contract type, $\frac{Pre}{CC}$ vary from 0.161 to 0.634; $\frac{Post}{CC}$ from 0.176 to 0.645; $\frac{Inv}{CC}$ from 0.000 to 0.315; and $\frac{Cons}{CC}$ from 0.000 to 0.033.

Considering the nonconformance ratio - $\frac{\#NCs}{CC}$ - values vary from 0.010 to 0.134. Precondition is the contract type more common at four out of seven projects. Invariant problems are more common at six out of seven projects.

D. Discussion

Concerning contract type, the most common contract clauses were precondition and post-condition: in all evaluated projects those kind of contracts were the most common. For some systems (as `HealthCard` and `TransactedMemory`) the ratio of precondition clauses ($\frac{Pre}{CC}$) is almost three times the ratio of postcondition clauses ($\frac{Post}{CC}$).

In addition, for four projects, precondition clauses are the majority of the written contracts; as a conclusion, the use of precondition clauses in the studied systems outperform the use of postcondition clauses, which corroborates with Estler et al. [4]: there is no preference for certain contract type, however, preconditions, tend to have more clauses than post-conditions.

This result suggests developers tend to write more pre- and postcondition clauses in comparison with invariants. Since invariants are related to fields of the classes, it is an expected finding: the amount of invariant clauses be smaller than the amount of pre- or postcondition clauses. This can be related to the fact usually, Java classes tend to have more methods than fields.

CCo metric or the ratio between each contract type and contract clauses are not enough for arguing about contracts or even code quality. For example, `HealthCard` has precondition clauses ratio of 0.634, *CCo* of 0.47 (the second highest value obtained in this study) but at the same time, JmlOk2 tool was able to detect 41 nonconformances in this experimental unit. On the other hand, `Mondex` has precondition ratio of 0.161, *CCo* of 0.27 but the tool was able to detect only two nonconformances in the project.

Aiming to discover relations between contract clauses and number of nonconformances, we statistically compared if the ratio of CC with LOC had relation with the ratio of NCs with CC. We found a moderate negative correlation (Pearson's correlation coefficient ρ = -0.40), so we cannot argue that contracts with more clauses imply more nonconformances, in contrast with our initial presumption that the number of nonconformances would be positively related to the contracts complexity. Answering our question: we found a moderate negative correlation.

Our result indicate developers who write bigger contracts tend to follow those contracts more closely; however, more studies are needed for more evidence about this.

E. Threats to Validity

This study has some limitations; next, we describe some threats to its validity. A 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.

[2]http://vsr.sourceforge.net/mondex.htm
[3]https://bit.ly/2XerhpM

Table I
FOR EACH SYSTEM, WE PRESENT ITS SIZE MEASURED BY MEANS OF LINES OF CODE (LOC), THE RESULTS OF 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.

Experimental Unit	**LOC**	$\frac{Pre}{CC}$	$\frac{Post}{CC}$	$\frac{Inv}{CC}$	$\frac{Cons}{CC}$	**CCo**	**#NCs**	$\frac{\#NCs}{CC}$	**Contract type**	**NC type**
Bank	792	0.524	0.341	0.135	0.000	0.16	3	0.024	Pre	Post and Inv
Bomber	6,258	0.355	0.645	0.000	0.000	0.02	5	0.041	Post	Post and Inv
HealthCard	2,156	0.634	0.231	0.102	0.033	0.47	41	0.040	Pre	Inv
JAccounting	6,648	0.505	0.485	0.010	0.000	0.03	26	0.134	Pre	Inv
Mondex	655	0.161	0.598	0.224	0.017	0.27	2	0.011	Post	Inv
Samples	3,855	0.372	0.497	0.129	0.001	0.48	18	0.010	Post	Post
TransactedMemory	1,779	0.505	0.176	0.315	0.003	0.17	8	0.027	Pre	Inv

We followed this approach for its simplicity, as it has been used by related research [4].

With respect to external validity, 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.

IV. CONCLUSIONS AND FUTURE WORK

In this work, we investigated the practice use of contracts in seven JML open source programs. In almost all projects, the most common contract clauses were precondition and postcondition, in contrast with the results of Estler et al. [4] who found no difference between the kind of contracts used.

This result suggests developers apparently prefer to write pre- and postcondition clauses in comparison with invariants. Since invariants are used, for example, for establishing the range of valid values, it is expected that the amount of invariant clauses be smaller than the amount of pre- or postcondition clauses. Usually, Java programs tend to have more methods than fields.

Furthermore, when investigating the relationship between contract complexity and number of nonconformances detected, we found a moderate negative correlation (ρ = -0.40); so, we cannot argue that contracts with more clauses imply more nonconformances. Our result indicate developers who write bigger contracts tend to follow those contracts more closely; however, more studies are needed for further evidence about this.

As future work, we plan to extend this work for considering more Java/JML open source projects and to others languages (such as Eiffel [18], Code Contracts [19], and Spec# [20]), and to define more metrics on contracts. We also intend to create a model for guiding developers in the process of writing contracts.

REFERENCES

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

[2] A. F. Milanez, "Fostering Design By Contract by Exploiting the Relationship between Code Commentary and Contracts," Ph.D. thesis, Federal University of Campina Grande, 2018.

[3] P. Chalin, "Are practitioners writing contracts?" in *Rigorous Development of Complex Fault-Tolerant Systems*, M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, Eds. Springer Berlin Heidelberg, 2006, vol. 4157, pp. 100–113.

[4] H.-C. Estler, C. Furia, M. Nordio, M. Piccioni, and B. Meyer, "Contracts in practice," in *Formal Methods*, C. Jones, P. Pihlajasaari, and J. Sun, Eds. Springer International Publishing, 2014, vol. 8442, pp. 230–246.

[5] P. Gibbins, "What Are Formal Methods?" *Information and Software Technology*, vol. 30, no. 3, pp. 131–137, 1988.

[6] C. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[7] B. Meyer, "Applying "Design by Contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[8] G. Leavens, A. L. Baker, and C. Ruby, "Preliminary Design of JML: A Behavioral Interface Specification Language for Java," *SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, 2006.

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

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

[11] 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.

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

[13] 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*, 2009, pp. 117–130.

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

[15] P. Schmitt and I. Tonin, "Verifying the Mondex Case Study," in *International Conference on Software Engineering and Formal Methods*, 2007, pp. 47–58.

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

[17] A. Milanez, D. Sousa, T. Massoni, and R. Gheyi, "JMLOK2: A tool for detecting and categorizing nonconformances," in *CBSoft (Tools session)*, 2014.

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

[19] M. Barnett, M. Fahndrich, and F. Logozzo, "Embedded Contract Languages," in *Symposium on Applied Computing*. ACM, 2010, pp. 2103–2110.

[20] M. Barnett, M. Fähndrich, R. Leino, P. Müller, W. Schulte, and H. Venter, "Specification and verification: The Spec# Experience," *Communications of the ACM*, vol. 54, no. 6, pp. 81–91, 2011.