Design-by-Contract Javadoc for API Documentation: Trade-offs between Dynamic Checking and Understandability

Alysson Milanez
Department of Systems and Computing, UFCG, Brazil
alyssonfilgueira@copin.ufcg.edu.br

Tiago Massoni
Department of Systems and Computing, UFCG, Brazil
massoni@dsc.ufcg.edu.br

Henrique Rebêlo
Informatics Center, UFPE, Brazil
hemr@cin.ufpe.br

Rohit Gheyi
Department of Systems and Computing, UFCG, Brazil
rohit@dsc.ufcg.edu.br

Gary Leavens
Department of Computer Science, UCF, USA
leavens@cs.ucf.edu

Abstract

Documenting API routines counts as a key benefit from applying the Design by Contract (DbC) methodology. In this context, DBC's pre- and post-conditions are used as part of the routine's public interface, for establishing call conditions and expected results, respectively, which could be then checked at runtime for detecting invalid calls or non-conforming implementations. It is well known, however, that programmers, despite this convenience, are resistant to use contracts – for Java programs, in particular, Javadoc natural text and tags are the dominant medium for documenting APIs. As a result, verification assistance is unavailable. In this paper, we report results from empirical studies on integrating contract expressions into Javadoc comments used as external API documentation. For this purpose, we designed a small tag-based extension to Javadoc to express pre- and post-conditions among other standard tags. The studies consisted in (1) evaluating understandability of contract expressions within API Javadoc, either for API developers or API users, by means of a experimental study and a follow-up survey, and (2) assessing its efficacy with a case study in which we manually formalized English contracts from open source Java systems into contract expressions, then checking conformance in runtime. Both quantitative and qualitative data show no significant difference between Javadoc English text and contract expressions, regarding either correctness of performed tasks or contract readability. Also, a straightforward translation of open source APIs to contract expressions was able to find a number of conformance problems between API contracts and implementation, embodying evidence on the usefulness of contract expressions for uncovering errors regarding the relationship between documentation and actual behaviour. These results are promising to research the adoption of contract languages for API documentation.

2012 ACM Subject Classification Software and its engineering → Software verification and validation, Software and its engineering → Domain specific languages

Keywords and phrases design-by-contract, documentation, runtime checking, Javadoc

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Digital Object Identifier 10.4230/LIPIcs...

Supplement Material TBD

Acknowledgements I want to thank ...

1 Introduction

Java programmers tend to consider writing Javadoc comments as a good practice, specially when these comments enhance public interfaces designed as third-party libraries for client programs. Despite its recognized value and practice in Java community, as discussed by [22], understanding how to use third-party libraries can be difficult. This mainly occurs when the source of documentation is only natural language comments that can be incomplete and ambiguous. In addition, a well-known problem is that documentation and implementation tend to diverge over time [5]; a Java programmer may forget to update the Javadoc documentation after performing an implementation change.

On the other hand, embedding contracts (with pre- and postconditions as executable assertions) has long been advocated by formal methods pioneers to precisely express code behavior. However, only a small amount of code has such contracts [16]. Part of the reason is notational, for example, in Java there is no built-in support for contracts, besides `assert` statements. To this end, we need an external contract framework like the Java Modeling Language (JML) [10] to express the full power of behavioral specifications. There is evidence that programmers are more likely to use contracts in languages that support them natively [2], such as Microsoft's Code Contracts [1] and Eiffel [11]. Another problem is that contracts expressed by the existing contract languages may be useful for programmers (internal documentation), but it does not meet the needs of other readers (separate/external documentation), such as third-party libraries [9, 15]. To use those libraries, a programmer should not need to look in the code to find out how to use it. Therefore, to maximize benefits, she must use the combination of such techniques (e.g., Javadoc and JML).

We propose a Javadoc-like language, called ContractJDoc, allowing Java programmers to add contract specifications (pre- and postconditions), in a straightforward way, into Javadoc comments. Only a few extensions are needed to allow contracts, such as invariants. To enable the power of contracts, the ContractJDoc compiler translates the documented contracts into corresponding JML specifications. Then, these JML specifications, equipped with pre- and postconditions, are compiled into runtime conformance checks.

We evaluate our approach by performing three studies: we first apply ContractJDoc to six Javadoc-annotated open source systems in order to analyse ContractJDoc's applicability. A number of previously-undetected inconsistencies between Javadoc and actual behaviour were found in some of the studied systems. Next, we report a study which observed 24 developers programming for Java interfaces with behaviour documented by the conventional Javadoc, JML [10], and ContractJDoc, within a controlled environment. As result, developers found it easier to implement an interface with contracts than writing a client for that interface. Also, in general pure Javadoc was straightforward to understand, but ContractJDoc performed better than JML, as we expected. Last, we investigate the readability of these three documentation approaches for specifying behavior in a Java interface – Javadoc, JML and ContractJDoc, by means of a survey with 142 Java developers. Survey results did not significantly differ for ContractJDoc and Javadoc, which is promising, as contracts are usually regarded as hard to read.

In summary, the main contributions of this paper are:

- A new approach for documenting source code – ContractJDoc (Section 3);
- A case study applying ContractJDoc to six Javadoc-annotated open source systems (Section 4.4);
- An empirical study with 24 developers programming for Java interfaces with behavior documented by the conventional Javadoc, JML, and ContractJDoc (Section 4.1);
- A comprehensibility survey with 142 Java developers investigating the readability of three documentation approaches for specifying behavior in a Java interface: Javadoc, JML and ContractJDoc (Section 4.2).

1.1 Research Questions

This research work investigates the impact of integrating contract expressions with Javadoc comments in public interfaces, in terms of applicability and usability. We first examine the impact of diverse contract documentation approaches with tasks related to the specification of data structure interfaces; as a follow-up, we inquired developers regarding understandability of contract examples. Second, we emulate the application of contract expressions to Javadoc-rich open source systems and analyzed results from runtime checking. In particular, we intend to answer the following research questions:

RQ1. What is the success rate in implementation tasks, using three forms of public interface contracts?

We report and discuss results from a lab study with Java developers over development tasks involving a documented data structure interface.

RQ2. What are the perceived difficulty in using three forms of public interface contracts?

By means of impressions given by the previous lab study and a follow-up developer survey, we discuss quantitative and qualitative data regarding the forms of contract expressions that are preferred for implementation tasks.

RQ3. Can inconsistencies in Java systems be uncovered if using runtime-checkable contract expressions?

We collected a few open source systems based on their use of Javadoc andapplied contract expressions to each system, evaluating the result in terms of detected conformance errors. Also, we discuss the problems faced when replacing Javadoc comments by contract expressions in the described context.

2 Background on Documentation Approaches

In this section, we discuss the existing problems and benefits in documenting programs. For concreteness, we exemplify documentation approaches for the Java language.

2.1 A Running Example

Figures 1a and 1b illustrate a simple bank account class, documented with JML and Javadoc respectively. For simplicity, we consider only the `withdraw` method.

In the JML specifications (Figure 1a), preconditions are defined by the clause `requires` and (normal) postconditions by `ensures`. The specification denoted by the `signals` clause is an exceptional postcondition, which says that the `balance` should be unchanged, when the exception `TransactionException` is thrown. The invariant defined in class `BankAccount` restricts the account's balance to be always greater than or equal to zero. In the Javadoc side (Figure 1b), the precondition is informally written with the tag `@param`, normal postcondition

```
class BankAccount {
 double balance;
 //@ invariant balance >= 0;

 //@ requires amt > 0 && amt <= bala
 //@ ensures balance == \old(balanc
 //@ ensures \result == balance;
 //@ signals (TransactionException)
 //@   balance == \old(balance);
 double withdraw(double amt)
   throws TransactionException {...
 // ...
}
```

(a) The JML specifications for the bank account.

```
class BankAccount {
 /**
  * The overall balance should be
  * greater than or equal to zero
  */
 double balance;

 /**
  * @param amt
the amount value to withdraw, where
  *
'amt' should be greater than zero and
  *
less than or equal to 'balance'
  * @return
current 'balance' after withdraw
  * @throws
TransactionException 'balance'
  *             remains unchanged
  *
  */
 double withdraw(double amt)
   throws TransactionException {...}
 // ...
}
```

(b) The Javadoc documentation for the bank account.

Figure 1 JML and Javadoc documentation for the bank account.

with `@return`, and exceptional postcondition with `@throws`. And the invariant within a Javadoc block comment above `balance`'s declaration.

2.2 Natural Language Approach

Often natural language documentation is found in the source code, as in Javadocs (see Figure 1b). Consistency between documentation and code cannot be automatically enforced, as there is no formal connection between the comments and the code. The lack of formality leads to imprecision, ambiguity, and verbosity (to explain the constraints). This poses translation problems to use natural language description for automated tools, such as in testing or debugging.

On the other hand, natural language documentation does have its advantages. It does not require special training – although training may be needed to effectively communicate ideas about programs in a natural language, such as English – and allows a high of freedom to a programmer structure the documentation. Furthermore, client's programs may benefit from the available documentation about the program usage.

2.3 Specification/Contract Language Approach

Contracts are a popular tool for specifying the functional behavior of software. A method's contracts, unlike natural approaches, precisely and unambiguously describe what must be true when the method is called (precondition), what must be true when the method return (normal postcondition) or when it returns abnormally (exceptional postcondition). In addition for object-oriented languages, contracts can describe object invariants that must hold for an

object in all of its visible states.

Nevertheless, a formal specification language, like JML, can require some level of training, hence becoming, to some extent, hard to read and write, and hence is often used sparingly [2, 16, 20]. In addition, assuming the code in Figure 1a as within a library, – where the source code is unavailable – we need the specification to reason about the library's usage. But, usually, the specifications like the ones in Figure 1a are only available for the implementor, thus becoming not useful for its clients [15].

2.4 The Documentation Dilemma

It is clear that we face a dilemma with respect to program documentation. If we use JML to provide formal documentation for contracts, the result is a more precise documentation, with the possibility of automatic checks. However, also results in a less flexible documentation in terms of using natural language to structure it. In addition, formal specifications are more interesting for the implementor/maintainer of the Java program (Section 4.1).

If we go back to a more flexible and informal documentation approach such as Javadoc, we face the lack of precision and potential ambiguity, even though Javadoc comments are more useful for third-party libraries users. This dilemma leads us to the following research question: Is it possible to have the best of both worlds? That is, can we mix informal documentation and contract specification within a unified approach?

In the following, we discuss how ContractJDoc provides means to combine the benefits of the existing documentation approaches discussed previously.

3 A Javadoc Extension for API Contracts

ContractJDoc for Java is an extension to the Javadoc-tagging systems with contracts surrounded by brackets. Its compiler extends AspectJML [17], with support for runtime checking. ContractJDoc allows programmers to document precise source code's behavior as part of Javadoc commentary. With just a few new tags, in addition to the standard Javadoc tags, one can write contracts as Javadoc comments that are compiled to runtime checkable code. The ContractJDoc approach fulfills the gap between informal documentation (as with Javadoc) and formal specification (such as JML [10] or Code Contracts [1]).

In the remaining of this section, we present how ContractJDoc supports Java code documentation with both informal and formal features of the documentation approaches discussed in Section 2. The presentation is informal and based on our running example.

3.1 ContractJDoc Design for Java

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. This mix is used to tackle the dilemma explained in the end of Section 2.

Embedding contracts means 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 specially true because the overwhelming majority of contracts that programmers write in practice are short and simple [5, 20]. For instance, 75% of Code Contracts [1] projects, the written contracts are basic checks for the presence of data (e.g., non-null checks) [20]. For scenarios like these,

there is no additional effort in embedding such contracts in Javadoc comments using our ContractJDoc approach.

3.1.1 Documenting Preconditions

Recall the precondition illustrated in Section 2. We discussed two ways to document such a precondition, a formal one in JML (see Figure 1a) and an informal one with plain Javadoc comments (see Figure 1b). In ContractJDoc that precondition can be rewritten to the following:

```
/**
 * @param amt the amount value to withdraw,
 *    where [amt > 0 && amt <= balance]
 */
double withdraw(double amt)
  throws TransactionException {...}
```

Tag `@param` documents parameter `amt` of `double` type. Besides the usual comments, we have added a Java-like boolean expression surrounded by brackets; these brackets indicate assertions internally in our tool; ContractJDoc compiles the above Javadoc comments into an executable precondition checking. An alternative (not showed) is to replace tag `@param` by `@requires` or `@pre`. Both can be used to document a precondition constraint; the main difference is that they are not part of the standard Javadoc tagging system.

3.1.2 Documenting Postconditions

A postcondition expresses the obligations of a supplier (and respectively the rights of a client), specifying the result of a method. As discussed in Section 2, two kinds of postconditions are usually found in Javadoc and JML, one related to normal return, and another related to exceptional return. Let us use ContractJDoc to document postconditions:

```
/**
 * @return amt the current balance after withdraw,
 *    that is [@return == balance]
 * @throws TransactionException the 'balance' does
 *   not change, that is [balance == @old(balance)]
 */
double withdraw(double amt)
  throws TransactionException {...}
```

Tags `@return` and `@throws` document the two kinds of postcondition, with their respective assertions expressed within brackets. Note that both use two special tags within the assertion brackets. Tag `@return` appears again within the brackets. This inner is allowed (in ContractJDoc) and allows one to use the value of the method's returns to write the contract regarding the normal postcondition. A similar tag, `@result` derived from the JML syntax, can also be employed instead of the inner use of the `@return` tag. Moreover, tag `@old` (also derived from JML) refers to expressions or fields in their pre-state. This tag can only be used by the Javadoc tags related to postconditions.

As with preconditions, ContractJDoc offers three tags for expressing postconditions. For normal postconditions, the similar JML-based tags `@ensures` and `@post` can be used in place of the standard one `@return`. In relation to the `@throws` tag, the standard Javadoc offers a surrogate tag `@exception`. Derived from JML, we can also employ the `@signals` tag to document and constrain exceptional behavior.

3.1.3 Documenting Invariants

Beyond the support for pre- and postconditions, ContractJDoc make the use of invariants also available by means of `@inv` tags. 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 applies for all methods from a class. For invariants, we follow the semantics of the JML language. For more information, please refer to [10]. The invariant contract, described in Section 2, can be written as follows:

```
class BankAccount {
 /**
  * @inv The overall balance should be [balance >= 0]
  */
 double balance;
 //...
}
```

Invariant declarations may be placed above the field declarations (as in the example), or above the class declaration as a valid Javadoc block comment.

3.2 ContractJDoc's Supporting Infrastructure

We implemented the ContractJDoc compiler in the top of the open source AspectJML/ajmlc compiler [17, 18, 19]. Unlike the original JML compiler (jmlc), ajmlc presents code optimizations and improved error reporting [18]. Differently from jmlc, AspectJML also enables the modularization of crosscutting contracts that can arise in standard JML specifications [17].

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 runtime checking the contracts. See Figure 2 for an overview of the compilation strategy.

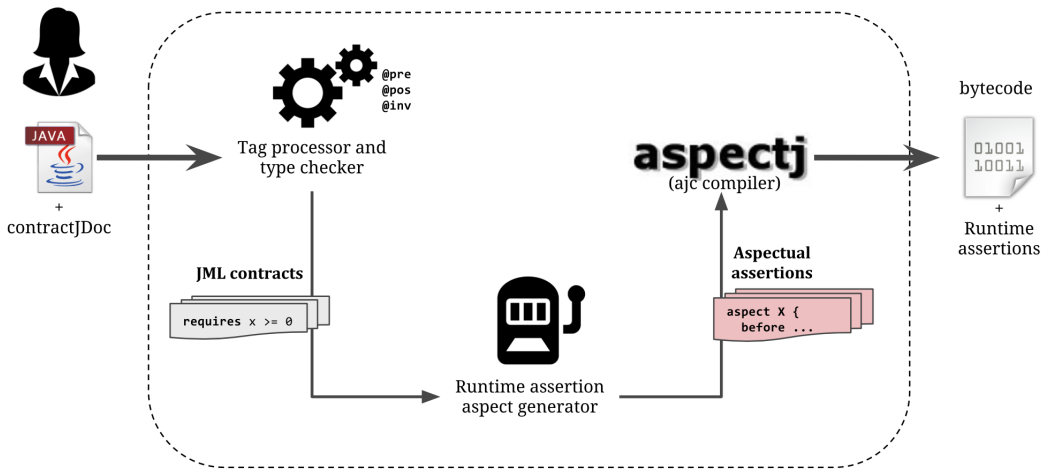


Figure 2 ContractJDoc Compilation. First, a source code with ContractJDoc contracts passes through a tag processor and type checker. Then, the assertions generated are runtime checked and AspectJ compiler produces a bytecode with assertions.

4 Methodology

In the following subsections we discuss data collection and analysis for the performed studies. For naming the studies, we follow the terminology for Software Engineering research strategies from Stol and Fitzgerald [21].

4.1 Experimental Simulation

In this study, we investigate the use of contract expressions – in particular, the Javadoc extension showed in Section3 – in simulated programming tasks, with respect to applicability and usability, from the point of view of Java developers.

4.1.1 Participants

We selected participants among professional developers assigned to projects in the context of an major R&D Institute in Brazil. Recruitment was carried out by invitation; from estimated 150 invited professionals, 24 of them accepted the call and then, later, received the assignment by e-mail. The only requirement was previous experience with Java. From the 24 recruited developers, 10 of them had computer science (or related) degrees, while the remaining were carrying out undergraduate studies in the field.

4.1.2 Study Design

Our experiment was designed to assess the effect of a given documentation approach on implementation tasks which depend on documented APIs. Participants are assigned to use a single documentation approach (Factor 1) from the following: plain Javadoc natural language, our Javadoc extension and formal contracts. Table 1 displays the treatments for those two factors. For each of those approaches, two kinds of task are considered (Factor 2): a *Supplier* task, in which the participant must program the implementation of a Java API class, and a *Client* task, which demands development of a client code for the API. An additional variation we included regards the particular API assigned to an arbitrary participant (Factor 3): *Stack* or *Queue*. We chose to use simple and well-known interfaces trying to avoid the confounding effect that lack of knowledge could bring to the study.

Table 1 Factors and treatments of the empirical study.

Factors	Treatments
Task	Client
	Supplier
Approach	Javadoc extension
	Javadoc
	formal contracts

We use a factorial design [25], randomly assigning participants to each combination of treatments; each triple <approach, task, API> is called a trial. Since there are three documenting approaches, two tasks and two Java interfaces, there are 12 possible trials, so each of them was carried out by two participants, resulting in a balanced design. The assignment Participant – Trial is performed by using a completely randomized design in order to minimize bias.

4.1.3 Experimental Procedure

The experiment was performed offline, i.e., participants received the experimental package via an online Survey platform[1] that we use to collect the results.

The experimental package consisted of (i) a statement of consent, (ii) a pretest questionnaire, (iii) instructions and materials to perform the experiment, and (iv) a post-test questionnaire. The instructions contained what we expected from the participants: they were asked to perform an implementation task (a supplier or a client code) for the provided API. Each participant received one of the following tasks: create a supplier code for an API or a client code using the API methods. We also asked each participant to fulfill a pre-study questionnaire reporting their programming experience (with respect to Java and contract-based programming experience).

During the assignment, participants were allowed to review additional information – part of the experimental package – about the assigned documentation approach. For *Supplier* tasks, participants should have implemented variations of well-known stack or queue operations whose intended behaviour was documented as external contracts. The following example shows the contract for `Queue.add`, written in plain Javadoc commentary.

```
/**
   * Inserts the specified account into the queue
   * if it is possible to do so
   * immediately without violating capacity
   * restrictions, returning true upon
   * success and throwing an AccountQueueException
   * if no space is currently available.
   * @param acc - the account to be added.
   *      Must not be null.
   * @return true if the operation occurs with success.
   * @throws AccountQueueException - if the queue is
   *      full or the account is null.
   */
  public boolean add(Account acc) throws AccountQueueException;
```

Likewise,

Before sending the results back to us, we asked the participants to answer a a post-experiment questionnaire, in which we collected qualitative information about the developers' view of each task.

We performed a pilot with three developers in order to fit the questionnaires' structure. As a result, we changed the way of making the artifacts available to participants. At first, we were making the documented interface available in a link and the working dataset in another. Pilot participants highlighted this fact, indicating that a single package containing all Java classes should be located in a single URL.

4.1.4 Research Method

labelsec:labmethod

4.2 Judgment Survey

We conducted an exploratory study that involved data collection through a survey with Java development professionals. The goal of the survey is to compare three documentation

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

approaches (Javadoc, ContractJDoc, and JML) with respect to comprehensibility, from the point of view of developers.

4.2.1 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 [4], in a non threatening way.

The survey instrument[2] 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: 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 understandable documentation approach: one specific – related to the provided interface; and one general, concerning the use of the approach in a general context.

We also conducted a pilot concerning the questions and the structure of the programs being used. The pilot consists in asking three Java developers to test the setup for the survey. The pilot allowed us to validate the survey's questions and structure. We did not have to change anything. The developers who participated did not reported issues on the structure that we used for presenting the needed data for the participation in the study.

4.2.2 Survey Participants

The survey participants are also extracted by means of a non-probability sampling technique – convenience sampling [25]: 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). From the 142, 51 are professionals and 91 are students.

4.3 Statistical Methods

In our experiment, we applied Wilcoxon rank sum test [8] and Kruskal-Wallis rank sum test [8] for comparing the results according to our treatments. For the survey results, we applied Oneway ANOVA test [8], The Tukey HSD [8] and pairwise comparisons using t tests with Bonferroni correction [8]; in addition, we applied Wilcoxon rank sum test with continuity correction tests.

4.4 Case Study

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 real, Javadoc-rich open source systems; all their method-level Javadoc

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

annotations are manually translated to ContractJDoc, before running tests looking for mismatches between specifications and actual method behavior.

4.4.1 Systems Selection

The case study was performed on a convenience sample: six Javadoc-rich open source systems available at GitHub[3] 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`[4] plays music from an ABC file (part of a project assignment from MIT class 6.005), `Dishevelled`[5] 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`[6] is a general-purpose set of Java tools and templates library. On the other hand, `OOP Aufgabe3`[7] aims to manipulate polygons. `SimpleShop`[8] is an electronical shopping system. In addition, `Webprotégé`[9] is a collaborative ontology development environment for the Web. Those systems amount to more than 190 KLOC. See Table 2 for details in terms of code lines (LOC), total contract clauses (#CC) we were able to write – following [5] approach, in which the number of contract clauses is a proxy for contract complexity – as split into preconditions (#Pre), postconditions (#Post), and invariants (#Inv).[10]

Table 2 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	1,411	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	929	351	579	0
Total	**190,655**	**3,993**	**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 para-

[3] `https://github.com/`
[4] `https://github.com/deepakn94/ABC-Music-Player`
[5] `https://github.com/heuermh/dishevelled`
[6] `https://github.com/mriedel/Jenerics`
[7] `https://github.com/rwilli/aufgabe3`
[8] `https://github.com/pase/simpleshop`
[9] `https://github.com/protegeproject/webprotege`
[10] The clauses correspond to the contracts we applied in each system.

meters. 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.

4.4.2 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 methods' source code. As result, they wrote 3,993 contract clauses: 1,952 preconditions, 2,029 postconditions, and 19 invariants (see Table 2). Figure 3 presents the steps performed by the researchers when applying ContractJDoc to the systems. The process is composed of four steps: 1) generation of the contracts based on the natural language comments available (as showed in Section 4.4.1); 2) compilation of the contracts 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.

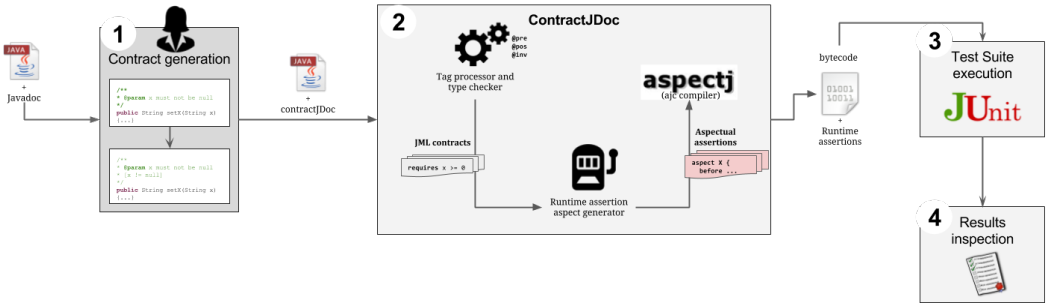


Figure 3 Steps for applying ContractJDoc to Javadoc-annotated systems.

Concerning the kind of written contracts, we group the contracts according to the approach of [20]: application-specific contracts (AppSpec.) – the kind of contracts that enforce richer semantic properties; common-case contracts (Com.Case) – the kind of contracts that enforce expected (common) program properties; 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.[11] Concerning the verification performed after applying ContractJDoc contracts into the systems, we used the test suites available with 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.

[11] `https://goo.gl/y08or2`; in order to run the ContractJDoc compiler, the folder *aspectjml-lib* must be copied into the folder of each 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 simplify the process of applying ContractJDoc in existing projects.

5 Results and Discussion

We present now the results obtained by each study performed.

5.1 Case Study

Table 3 presents the results of applying ContractJDoc to each system. Column #Clauses displays the number of clauses manually added in each system. Column #Errors presents the number of errors 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 after applying ContractJDoc contracts. Columns #Com.Case to #Repet. show the contract clauses added in each system grouped by type (following the definitions from [20]).

Table 3 Case study Results.

System	#Clauses	#Errors	Time (s)	#Tests	#Com.Case	#AppSpec.	#Repet.
ABC-Music-Player	115	2	14	30	42	11	62
Dishevelled	2,655	381	434	2,643	1,536	151	968
Jenerics	190	7	20	44	156	0	34
OOP Aufgabe3	54	1	4	11	16	30	8
SimpleShop	50	0	5	0	30	11	9
Webprotégé	930	0	713	0	717	79	133
Total	**3,993**	**391**	**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 remains 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 remains 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.

5.1.1 Case Study

For all systems (see Table 2), we wrote more pre- and postconditions than invariants. This result has two explanations: first, 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 into `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.

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.

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.

5.2 Empirical Study

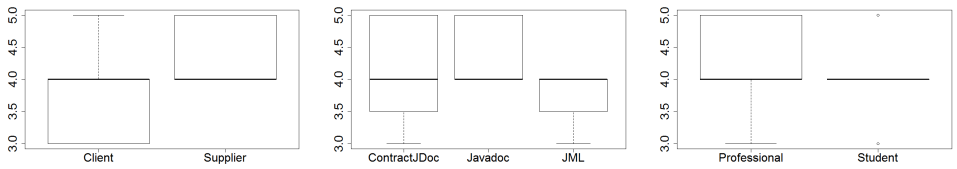
27 Java developers participated in our experiment. From those, we randomly discarded three in order to maintain a balanced number of participants in each trial. Thus, we maintained 24 developers, from those, 10 are working in industry (41.6%) and 14 are students (58.4%).

Concerning the task performed by the developers, we present in Figure 4a 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 [8] (p-value = 0.07, confidence level of 95%).

When grouped by documenting approach (see Figure 4b), the Kruskal-Wallis rank sum test showed no difference between the approaches (p-value = 0.15).

When grouped by experience (Figure 4c), 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.



(a) (b) (c)

Figure 4 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.

5.2.1 Empirical Study

We proceed with the discussion over 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 4(a)), the statistical test did not provide evidences for a significant difference between the difficulty perceived by developers when performing the required task.

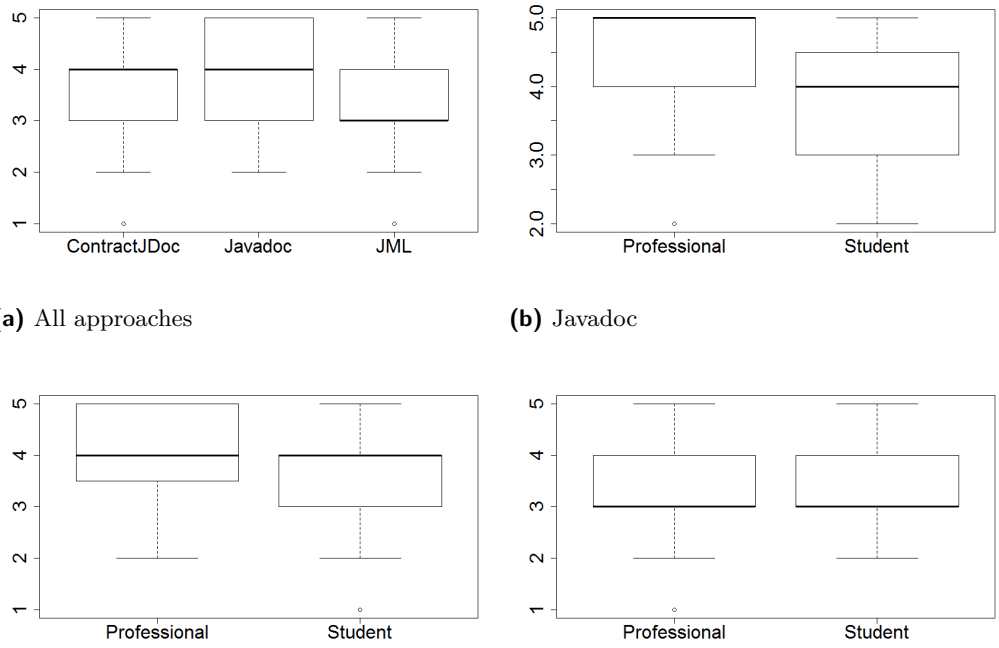
Q3. Although not supported by statistical tests since Kruskal-Wallis rank sum test showed no difference between the approaches (p-value = 0.15), Javadoc and ContractJDoc were perceived as being easier than JML (see Figure 4b). This result indicates ContractJDoc as an approach in an intermediate level 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 the 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 been perceived the supplier task as less difficult than the client task, 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).

5.3 Comprehensibility Survey

142 Java developers answered the survey. From those, 51 are professionals (36%) and 91 are students (64%).



(a) All approaches

(b) Javadoc

(c) ContractJDoc

(d) JML

Figure 5 Subjects' answers to the individual evaluation of comprehensibility for each documentation approach. And answers grouped by experience for each approach.

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 the comprehensibility of the documentation approaches evaluated (see Figure 5a). By performing an Oneway ANOVA test [8] and a corresponding post hoc analysis we were able to distinguish the three approaches (p-value < 0.05). The Tukey HSD [8] and pairwise comparisons using t tests with Bonferroni correction [8] produced the following p-values: Javadoc-ContractJDoc $= 0.012$, JML-ContractJDoc $= 0.000$, and JML-Javadoc $= 0.000$.

When analyzing data grouped by experience (Figures 5b to 5d) 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 easier for comprehensing than Students (p-value $= 0.012$ and p-value $= 0.004$, respectively).

5.3.1 Comprehensibility Survey

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 5a.

An interesting result came from the analysis of the difficulty grouped by experience (Figures 5b to 5d): students and professionals have perceived the same level of difficulty for

JML, which is promissing as contract-based languages are usually considered harder to be understood by people with less experience (students, in our survey).

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

6 Limitations and Threats to validity

External validity refers to generalization. Due to its size, results from the case study cannot be generalized; its purpose is evaluating 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.

Another risk is that only 24 developers participated in the experimental study and 142 in the survey and those samples are not representative for the community of Java developers. Furthermore, we used only two similar data structure interfaces (queue and stack). In other domains with more complex structures, the results may vary. In addition, the survey 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.

Internal validity refers to causation: are changes in the dependent variable necessarily the result of manipulations to treatments? All material for the empirical study and 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.

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

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 also 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. In addition, 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.

7 Related Work

Contracts. As discussed, each contract-based approach chooses a different trade-off between expressivity/preciseness, verbosity, freedom, and tooling (e.g., runtime checking). This is the case of JML [10] and Microsoft's Code Contracts [1]. Both enable one to provide full behavioral specifications and their runtime checking. Nevertheless, they lack support to allow informal specifications or to be available at third-party library clients [15]. They differ in the way they are written. The former is written as Java comments in code, whereas the latter is syntax-based and therefore often verbose. Without tool support to extract meaningful specifications the contracts provided by Code Contracts are even less interesting for third-party libraries since they are embedded in C# programs. Differently from these languages, the contracts expressed in ContractJDoc are already embedded in Javadoc comments, which are the standard approach to documenting Java programs and more likely to be available to third-party libraries.

Javadoc Comments. @TCOMMENT [23] is an approach for testing Javadoc comments, specifically method properties about null values and related exceptions. The approach consists 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 component generates random tests for these methods, checks the inferred properties, and reports inconsistencies. By using ContractJDoc, a developer is able to write contracts richer than those for checking null values and exceptions (as presented in Section 4.4).

Zhai et al. [26] 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.

Testing. Clousot [6] statically checks C#/Code Contracts programs. The approach is based on abstract interpretation and analyzes annotated programs to infer facts (including loop invariants), and it uses this information to discharge proof obligations. AutoTest [13] is a collection of tools that automate the testing process for Eiffel programs. In AutoTest, contracts are used as oracles to expected outputs for conformance checking of the programs; furthermore, AutoTest uses a randomly-guided tests generation (ARTOO [3]) and supports mixing manual and automated test. JmlOk2 [14] is a tool for dynamically detecting and classifying nonconformances in contract-based programs, applying randomly-generated tests (RGT) for detecting nonconformances, and a heuristics-based approach for nonconformance classification. These tools are for contract-based languages; in contrast, we propose and implement an approach for writing contracts in the same language as the source code, such as available in JML and Eiffel, improving documentation.

Empirical Studies. There are three main related empirical studies about contract usage [20, 5, 2]. One common conclusion about is that, in practice, developers use simple and short contracts [20, 5]. For instance, [20] showed that 75% of the projects' Code Contracts are checks for the presence of data (e.g., non-null checks). In our case study (Section 4.4), almost 93% (3,711 contract clauses out of 3,994) of the contracts we wrote remains between checks for the presence of data and statements repeating the method's return. Chalin studied

84 Eiffel [11] projects and pointed out that developers are more likely to use contracts in languages that support them natively, like Eiffel [11] or Code Contracts [1]. To support both conclusions, to write simple contracts in a native manner, ContractJDoc allows a Java programmer to write those contracts as usual Javadoc comments.

8 Conclusions

In this work, we present a new approach for writing comments and an application of this approach for Java context. The approach allows the use of Design by Contract [12] in a format closer to traditional Javadoc comments. When evaluating ContractJDoc we found the approach almost as understandable as traditional Javadoc comments, with the advantage of being able to check behavior at runtime. When comparing JML with ContractJDoc, the latter features less specification constructs, although results indicate programmers may feel more comfortable with it when writing precise behavior for methods. In addition, we found evidence that ContractJDoc is more readable than JML.

ContractJDoc contracts enable runtime checking by using a language similar to the traditional Javadoc, and our compiler (ajmlc-contractjdoc) supports new constructs of Java language, such as the features from the Java 8, such as lambda expressions. Furthermore, as JML contracts, contracts in ContractJDoc may be used in place of defensive programming, by specifying valid inputs for the methods and shortening the source code.

As future work, we plan to leverage ContractJDoc for supporting autocomplete when writing the Javadoc comment. For this purpose, we can use the Jaro-Winkler [7, 24] string distance for autocompleting the comments.

References

1 M. Barnett, M. Fähndrich, and F. Logozzo. Embedded Contract Languages. In *Symposium on Applied Computing*, pages 2103–2110. ACM, 2010.

2 P. Chalin. Are practitioners writing contracts? In Michael Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *Rigorous Development of Complex Fault-Tolerant Systems*, pages 100–113. Springer-Verlag, 2006.

3 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.

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

5 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.

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

7 Matthew 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.

8 G. Kanji. *100 Statistical Tests*. Sage, 2006.

9 G. Leavens. The future of library specification. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 211–216. ACM, 2010.

10 G. Leavens, A. Baker, and C. Ruby. JML: A Notation for Detailed Design. In B. Rumpe H. Kilov and W. Harvey, editors, *Behavioral Specifications for Businesses and Systems*, chapter 12, pages 175–188. Springer US, 1999.

11 B. Meyer. Eiffel: programming for reusability and extendibility. *ACM SIGPLAN Notices*, 22(2):85–94, 1987.

12 B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

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

14 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 (Tools session)*, pages 69–76, 2014.

15 D. Parnas. *Precise Documentation: The Key to Better Software*, pages 125–148. Springer Berlin Heidelberg, 2011.

16 N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *International Symposium on Software Testing and Analysis*, pages 93–104. ACM, 2009.

17 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.

18 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.

19 H. Rebêlo, S. Soares, R. Lima, L. Ferreira, and M. Cornélio. Implementing java modeling language contracts with aspectj. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 228–233. ACM, 2008.

20 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.

21 Klaas-Jan Stol and Brian Fitzgerald. A Holistic Overview of Software Engineering Research Strategies. In *2015 IEEE/ACM 3rd International Workshop on Conducting Empirical Studies in Industry*, pages 47–54. IEEE, may 2015. URL: `http://ieeexplore.ieee.org/document/7167427/`, `doi:10.1109/CESI.2015.15`.

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

23 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. IEEE Computer Society, 2012.

24 William 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.

25 C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, and B. Regnell. *Experimentation in Software Engineering*. Springer, 1st edition, 2012.

26 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.