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

1

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

Digital Object Identifier 10.4230/LIPIcs...

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 [18], 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

1

implementation tend to diverge over time [4]; 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 [12]. 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) [8] 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 [9]. 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 [7, 11]. 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 [8], 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 4.3);
- 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 understandability 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 Styles of API Contracts

In this section, we discuss issues in specifying the behaviour of API interfaces. For concreteness, we provide Java examples.

2.1 Javadoc and textual specifications

Javadoc [**?**] is the usual notation (and tool) for API specifications in Java; it includes special tags (with symbol) for structuring and pretty-printing code commentary. The Java Platform API specification itself [**?**], for instance, employs Javadoc for specifying "contract[s] between callers and implementations." In those terms, Javadoc may be instrument for applying the Design-by-Contract (DBC) methodology [10], with its pre- and post-conditions around public methods, establishing the expected behaviour for each part (the contract). In the context of distributed software teams, for instance, this kind of documentation is of critical importance.

Consider the bank account API depicted in Figure 1[2]. For simplicity, only method `withdraw` is declared. Tag `@param` includes, for parameter `amt`, a description that suffices as a pre-condition for `withdraw` callers. Likewise, tags `@return` and `@throws` document, respectively, a normal post-condition (if it works correctly) and a exceptional post-condition (if `TransactionException` is thrown).

Contracts in such style use natural language. As a consequence, consistency between specifications and actual code behaviour cannot be automatically enforced, unless one maintains test cases in synchronicity with the Javadoc contracts. However, even test cases are hardly up-to-date to code changes [], so it is hard to imagine that would be applicable. Furthermore, the lack of formality leads to imprecision, ambiguity, and verbosity, potentially leading to program anomalies and faults. This poses translation problems to use natural language description for automated tools, such as in testing or debugging.

[2] we consider API the public members of a Java class, or a Java interface

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

Figure 1 Bank Account API Specification using Javadoc.

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

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

Figure 2 The JML specifications for the bank account.

On the other hand, using natural language does not require special training – although training may be needed to effectively communicate ideas about program behaviour – and allows a high degree of freedom for documentation structuring.

2.2 Formal Contracts

DBC is supported by contruction in a few programming languages (such as Eiffel []), or by extensions (Java Modeling Language (JML) [] for Java and Code Contracts [] for .NET languages) in mainstream programming languages. For Java, JML contracts may be defined as showed in the same API for a bank account, in Figure 2. Pre-conditions are defined by the clause `requires` and (normal) post-conditions by `ensures`. The specification denoted by the `signals` clause is an exceptional post-condition stating that `balance` should be unchanged, when the exception `TransactionException` is thrown.

In this style, formal contracts for methods, unlike natural approaches, precisely describe what must be true when the method is called, what must be true when the method return or when it returns abnormally. A critical property of such contracts is that they are machine-checkable, either by assertion testing [] or static analysiscite. Nevertheless, using JML-like formal contracts might require some level of training, hence becoming, to some extent, hard to read and write, and hence is often used sparingly [2, 12, 16]. In addition, assuming the code in Figure 2 is an API with no available code, the specification is needed to use the API. However, usually formal contracts are only available for the API implementors, thus

becoming not useful for its clients [11].

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. If we refer 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 inquiries: is it possible to have the best of both worlds, mixing informal documentation and contract specification within a unified framework? In this case, what would be the effect of using such approach to API usage and development? This paper tries to enhance evidence on DBC development and languages by focusing on those questions.

3 A Javadoc Extension for API Contracts

In order to perform studies on approaches for specifying APIs, we propose ContractJDoc, a simple extension to the Javadoc-tagging systems with contracts. Its compilation system is built on the top of the AspectJML compiler [13], providing support for runtime checking of the contracts. ContractJDoc allows programmers to document *contract expressions* amid Javadoc header for methods. 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.

In the remaining of this section, we present, based on a running example, how ContractJDoc supports a mixed approach, which combines textual documentation with a limited set of formal features from the examples in Section 2.

3.1 Language Extension Design

The ContractJDoc tags act as traditional Javadoc tags, embedded within block comments. The main idea is to allow a mix between the traditional Javadoc syntax and JML-like notation; JML seems apropriate in this context, since it is built over Java expressions, with exception of a few logical operators (such as `==>` – implication).

Embedding contracts allows for contract expressions (e.g., pre-conditions) in the existing Javadoc comments and making them machine discoverable through the use of marker brackets within those comments. In this proposal, we consider the textual comments surrounding the formal expressions semantically neutral – we assume the expression does not change its semantics.

The potential benefit of embedding contract expressions in a more natural setting for the developer is his/her ability to remain within a single artifact which is purpose-built for writing API specifications. This is specially true because the overwhelming majority of contracts that programmers write in practice are short and simple [4, 16]. For instance, in 75% of Code Contracts [1] projects, the written contracts are basic checks for the presence of data (e.g., non-null checks) [16]. In such scenarios, there is no additional effort in embedding such contracts in Javadoc comments using our ContractJDoc approach.

3.1.1 Pre-conditions

In ContractJDoc the pre-condition for method `withdraw` from Section 2 can be rewritten as the excerpt in Figure 3.

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

Figure 3 `withdraw`'s pre-condition.

```
/**
 * @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 {...}
```

Figure 4 `withdraw`'s post-condition.

Tag `@param` documents parameter `amt` of `double` type. Besides the usual comments, we have added a boolean expression surrounded by brackets; these brackets indicate assertions internally to the ContractJDoc compiler, so the comments can be turned 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 Post-conditions

We may use ContractJDoc for post-conditions as the example in Figure 4.

Tags `@return` and `@throws` document normal and exceptional post-conditions, respectively, with their respective expressions expressed within brackets. Tag `@old` refers to expressions or fields in their pre-state, used only in post-conditions.

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

3.2 Supporting Infrastructure

The ContractJDoc compiler is based on the open source AspectJML/ajmlc compiler [13, 14, 15]. Unlike the standard JML compiler jmlc [**?**], ajmlc presents code optimizations and improved error reporting [14]. Also, AspectJML enables the modularization of crosscutting contracts that can arise in standard JML specifications [13].

We adapted the front-end of the AspectJML/ajmlc compiler to convert/preprocess the ContractJDoc tags into the corresponding JML features, like pre- and post-conditions. After conversion, the compilation occurs as usual and generates aspects to runtime checking the contracts. See Figure 5 for an overview of the compilation strategy. First, source code with ContractJDoc contract expressions goes through a tag processor and a type checker.

Next, a runtime assertion aspect is generated, which is woven to the source code by the aspectj compiler, producing bytecode with assertions, amenable to runtime checking.

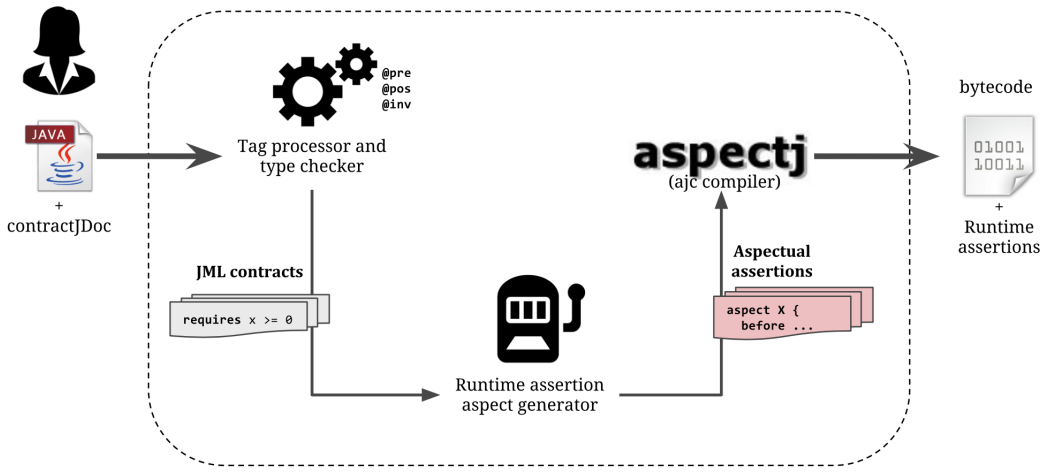


Figure 5 Compilation Infrastructure for ContractjDoc.

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 [17].

4.1 Experimental Simulation

In this study, we investigate the use of external contract expressions – in particular, ContractJDoc, as showed in Section3 – in simulated programming tasks, with respect to applicability and understandability, 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.

Although the number of participants may seem too small for generalizing any conclusions, we expect the observations to be considered as an exploratory theory to be checked by future studies with larger samples. For experimental simulations, our participant set is within the expected size [].

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 [21], 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[3] 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 ContractJDoc expressions. The following example shows the contract for `Queue.add`, written in plain Javadoc commentary.

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

```
/**
   * 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;
```

Differently, *Client* tasks were carried out based on textual descriptions, although any call to API should fulfill its external contracts, also presented using one of the three documentation options. Implementation of the API were provided as binaries, so, as a result, participants were not given access to the source code. Before sending the results back, 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

Each program sent by the participants were subjected to test suites specially implemented for checking whether every single contract was fulfilled. As a result, we are able to classify each submission according to the number of contract-code inconsistencies.

For the objective questions in the post-study questionaire...

The answers from qualitative post-study questionnaire were subject to a simple content analysis, performed by two researchers, which held a joint session for agreement on the category for each response from the developers.

4.2 Judgment Survey

The goal of the survey is to compare three documentation approaches (Javadoc text, Javadoc with contract expressions and formal contracts) with respect to understability, from the point of view of developers.

4.2.1 Participants

We selected participants by means of a non-probability convenience sample [21]. The survey link was sent to academic and professional mailing lists. In addition, our contacts were asked to follow 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 contacts who received the link – approximately a 20% response rate). From the 142 participants, 51 ($\tilde{3}6\%$) are professionals and 91 are computer science (and related topics) students.

4.2.2 Design and Method

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 [3], in a non-threatening way.

The survey instrument[4] 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, allowing us to validate the survey's questions and structure. The developers who participated, however, did not reported issues on the structure that we used for presenting the needed data for the participation in the study.

Regarding quantitative methods, we applied Wilcoxon rank sum test [6] and Kruskal-Wallis rank sum test [6] for comparing the results for the different groups of participants. Regarding survey results, we applied Oneway ANOVA test [6], The Tukey HSD [6] and pairwise comparisons using t tests with Bonferroni correction [6]; in addition, we applied Wilcoxon rank sum test with continuity correction tests.

4.3 Case Study

This study aims at assessing the usefulness of contract expressions within Javadoc text, with respect to automation benefits, from the point of view of Java developers. We observe the results from applying adding contract expressions to six real, Javadoc-rich open source systems; all their method-level Javadoc annotations are manually translated to contract expressions, before running tests looking for mismatches between specifications and actual method behavior.

4.3.1 Systems Selection

The case study was performed on a convenience sample: six Javadoc-rich open source systems available at GitHub[5] 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.

[4] `https://goo.gl/forms/8W9jUMGCavkkzDj12`
[5] `https://github.com/`

While `ABC-Music-Player`[6] plays music from an ABC file (part of a project assignment from MIT class 6.005), `Dishevelled`[7] 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`[8] is a general-purpose set of Java tools and templates library. On the other hand, `OOP Aufgabe3`[9] aims to manipulate polygons. `SimpleShop`[10] is an electronical shopping system. In addition, `Webprotégé`[11] 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 [4] approach, in which the number of contract clauses is a proxy for contract complexity – as split into preconditions (#Pre), postconditions (#Post), and invariants (#Inv).[12]

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 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.

4.3.2 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 6 presents the steps performed by the

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

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 4.3.1); 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.

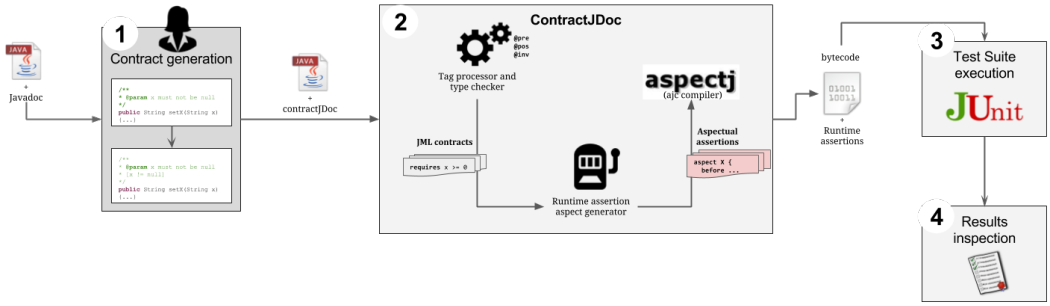


Figure 6 Steps for applying ContractJDoc to Javadoc-annotated systems.

Concerning the types of external contracts, we group the contracts according to the approach introduced by XXXXX et al. [16]: 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 used in this study are available in a replication package.[13] 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.

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

We start by describing the results for each study in detail, before proceeding to summarize and discuss the observed effects.

5.1 Experimental Simulation

The selected participant set encompasses graduates in Computer Science or Software Engineering – either working in industry (41.6%) or M.Sc. and Ph.D. candidates (58.4%). In Table 3 we summarize the results from the 24 trials, from which five – 21% – were delivered with at least one fault detected by our test cases. All participants assigned to textual specifications in Javadoc delivered correct programs. By contrast, half of the programs delivered

[13] `https://goo.gl/y08or2`; in order to run the ContractJDoc compiler, the folder *aspectjml-lib* must be copied into the folder of each system.

Table 3 Experimental results, for each treatment (textual Javadoc *JavaDoc*, ContractJDoc *ContJDoc* and formal contracts *Formal*. For each API (Queue or Stack) and Task (*Cli* if a client for the API was implemented, *Sup* if an implementation for the API was provided), participants are listed (*Part*) along with the result (*Res*) from our test cases.

		JavaDoc		ContJDoc		Formal	
DataStr	Task	Part	Res	Part	Res	Part	Res
Queue	Cli	p9	✓	p11	✓	p7	✓
	Cli	p10	✓	p12	✓	p8	✓
	Sup	p21	✓	p23	✓	p19	✓
	Sup	p22	✓	p24	✓	p20	×
Stack	Cli	p3	✓	p5	✓	p1	×
	Cli	p4	✓	p6	✓	p2	✓
	Sup	p15	✓	p17	×	p13	×
	Sup	p16	✓	p18	✓	p14	×

Table 4 Reason (fault) for failures in participants' results

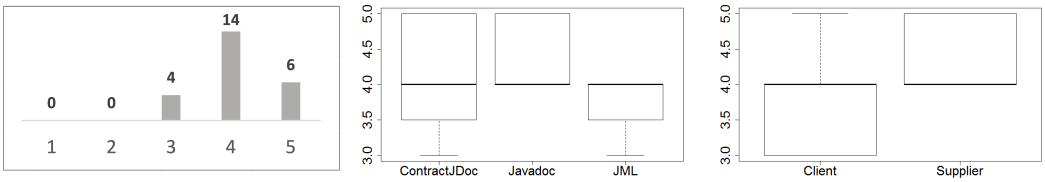
ContractJDoc			
p17	Stack	Sup	Post-condition violation on method `remove`
Formal contracts			
p1	Stack	Cli	Failure to expect correct exceptions from Post-condition on method `removeAccountTop`
p13	Stack	Sup	Post-condition violation on method `pop`
p14	Stack	Sup	Post-condition violation on method `pop`
p20	Queue	Sup	Post-condition violation on method `remove`
Part	**API**	**Task**	**Type of Fault**

by participants using APIs with formal contracts were faulty (four out of eight). One faulty program was delivered by a participant from the ContractJDoc group. Regarding task, faulty programs were mostly present in implementing in API implementations – four of them, if compared with only one in API clients.

Table 4 details the faults detected by the test cases. After detailed analysis of the programs, we classified each fault in terms of the violated contract. All clients fulfilled the specified pre-conditions for calling the API methods – p1's API client raise an exception which was incompatible with the method's postcondition. All four faults unveiled in API implementations resulted from failure in satisfying the specified post-condition in methods removing data from the given structure (two on `Stack.pop` and two on `Queue.remove`).

After they sent their code back, we asked participants about understandability of the API specifications, using a Likert-like scale which ranges from 1 (less understandable) to 5 (very understandable). The results for 24 answers are summarized in Figure 7a, where we can see assessments are split between 3 to 5. Most participants evaluated understandability as 4 (58%).

Assuming a distinct perspective, we grouped understandability assessments by the assigned documentation approach and task – Figures 7b and 7c, respectively, depict their distribution using boxplots. Visually, the assessments for Javadoc APIs are all above the median (4), while ContractJDoc assessments fluctuate around that median value. Formal contracts, in turn, were all assessed as 3 or 4. Nevertheless, statistical tests (in this case, Kruskal-Wallis non-parametric test) showed no difference between the groups (p-value = 0.15, 95% confidence level).



(a) Total Results.

(b) Results by approach.

(c) Results by task.

Figure 7 Results from Understandability Assessment of API specifications, as Perceived from Participants.

Table 5 Categories in Participants' quotes

Category description	#quotes
Specifications were clear and understandable	7
Specifications were vague	7
Suggestions to change the specification	4
The surrounding text helped understanding	2
Total trust in the contracts	2
Contracts were ignored	1
Hard to read the logic	1
Questions about the task	1
Text was confusing	1

Concerning the task performed by the developers, higher understandability is reported by participants assigned to supplier tasks, at least visually. Still, the Wilcoxon rank sum test [6] reported no significant difference (p-value = 0.07, for confidence level of 95%).

Furthermore, we asked the participants, with an open-ended question, to provide comments on their tasks. We used *open coding* [**?**] to analyze the collected answers, by inspecting them quote by quote and detecting categories, representing the key ideas in the data. The quotes were organized in 11 categories, which are presented in Table 5, in conjunction with the number of quotes. One quote might classified in more than one category – three participants did not provide any answer.

We found that most quotes either value the specifications or suggest the contracts should have been stronger (seven each). As an example of the latter, p6, which was assigned an ContractJDoc API, wrote "I hesitated over the `pop` method, due to its exception; there should be more information about the exception to be thrown in each case". Others made explicit suggestions on how the specification should be, from their previous understanding of Queue and Stack APIs. Interestingly, we found a few quotes that expressed total trust in the contracts, dismissing defensive programming as advocated by the Design by Contract methodology [10]. Another category encompasses quotes that remark how the Javadoc text around the contract expressions helped them to understand the API specification.

5.2 Judgment Survey

As a follow-up for the lab study, we showed three versions of the Stack API as a web survey, asking questions about those approaches to contract specifications; we suggested respondents to play the client role of the API, as if they were about to instantiate it and call its methods.

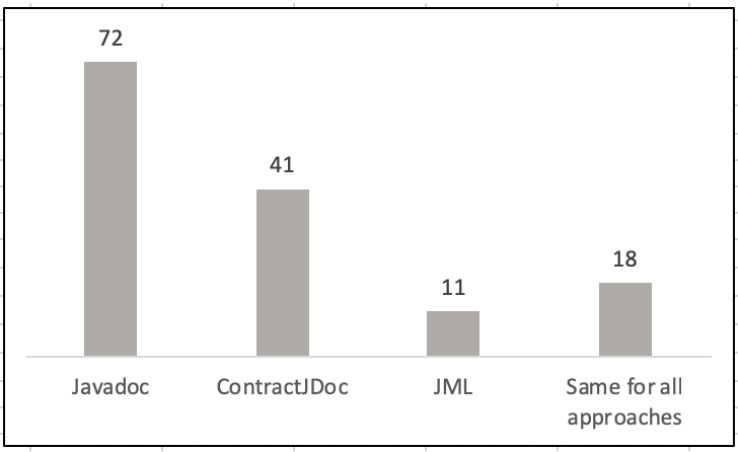


Figure 8 Surveyed Verdicts on the Most Understandable API Documentation Approach.

Respondents were asked to assess understandability of each approach (again, using a 1–5 Likert scale), in addition to providing a verdict asking which approach they think is the most understandable as an API specification. For this, they were given three options – one for each approach – and a fourth, neutral option.

From the universe of developers who received the invitation, answers from 142 Java developers were considered valid. From those, 51 are graduate professionals (36%) and 91 are undergraduate and graduate students in computer-related degrees (64%). Having some experience with Java was a requirement for being a participant; more than 73% (105) have experience with either open source or industrial projects. Likewise, 74% (105) of the respondents declared to have more than one year of Java programming. Regarding formal contract languages for specifying APIs (Eiffel, JML or similar), most respondents – 67% – had no previous experience.

With respect to the survey answers, it is showed in Figure **??** that 50.7% (72) of the respondents chose Javadoc as the simplest approach to understand the provided API specification. While only 11 (7%) chose formal contracts as the most understandable approach and 18 (12%) were indifferent, a almost one third of the respondents (41, almost 29%) chose the mix of text and contract expressions – ContractJDoc.

After being presented with three versions of the same API, the distribution of assessments for each documentation approach is showed in Figure **??**. Answers for Javadoc and ContractJDoc were distributed around a median of 4.0; for the first, assessments range from 3 to 5, while assesments for the latter were no higher than 4. Likewise, formal contracts were evaluated with 4 the highest, however with 3 with center value.

Differently from the experimental simulation from Section 5.1, statistical tests show significant difference between the assessments. The application of One-way ANOVA, which is robust enough for non-normal distributions [6], detected distinction between the three groups of answers (p-value < 0.05). A corresponding post hoc analysis – Tukey HSD with Bonferroni correction [6] produced the following p-values for each pairwise comparison: Javadoc-ContractJDoc = 0.012, JML-ContractJDoc = 0.000, and JML-Javadoc = 0.000. The results then express significant difference between all approaches.

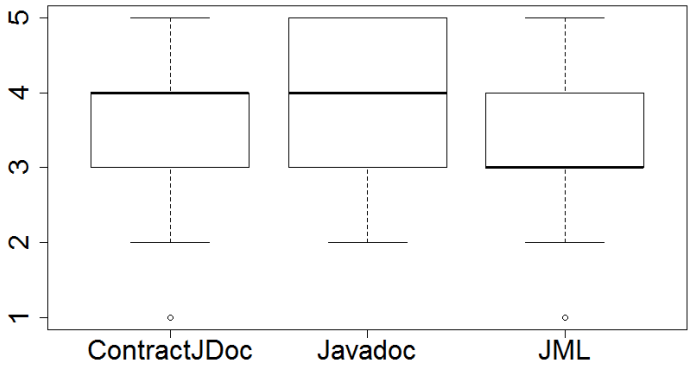


Figure 9 Surveyed Assessment for Each Documentation Approach.

Table 6 Results from Study Replacing Javadoc with ContractJDoc.

System	Faults	Time(s)	TCs	Clauses	CommCase	AppSpec	Repet
Dishevelled	381	434	2,643	2,655	1,536	151	968
Webprotégé	0	713	0	930	717	282	1,214
ABC-Music-Player	2	14	30	115	42	11	62
Jenerics	7	20	44	190	156	0	34
OOP Aufgabe3	1	4	11	54	16	30	8
Total	**391**	**1,185**	**2,728**	**3,993**	**2,497**	**282**	**1,214**

5.3 Case Study

Table 6 exhibits the results of translating real Javadoc specifications to ContractJDoc in each system. Column **Faults** presents the failures raised by the original test suite (whose size is showed in Column **TCs**), with source code enhanced by ContractJDoc, while Column **Time** measures the seconds spent in compiling the system instrumented with ContractJDoc before running the test cases. **Clauses** displays how many contract clauses were translated into contract expressions, which are classified – according to XXX et al. [16] – in Columns **CommCase**, **AppSpec** and **Repet**.

From the selection of open source systems, only *WebProtégé*'s test suites were unavailable; we translated their clauses nonetheless due to the richness of its Javadoc specification. The size of the available test suites is proportional to the system size. *Dishevelled* is much larger than the other selected systems, naturally presenting a higher rate of nonconformances, which provided us a profuse source of analysis about the differences between specification and implementation. About the time...

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*, most contracts are classified as common-case (57.51%) – only 5.57% are application-specific. Similarly, 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. For *WebProtégé* common-case contract expressions amount to 77.51%.

6 Discussion

This section comprises discussion on the results from our empirical studies: experimental simulation, judgment survey and case study – using, as basis, the aforementioned research questions, and threats to validity.

6.1 RQ1. What is the Effect of the Documentation Approach on API Usage and Implementation Tasks?

Concerning code correctness, all participants assigned to Javadoc APIs produced code in accordance with the contracts – our manually-produced test cases did not detect any contract violation. One ContractJDoc API implementation was delivered with a single fault, while half of the participants assigned to API with formal contracts presented at least one fault. These results may suggest participants had trouble understanding API formal contracts, assuming their self-reported experience in Java programming and their intention to complete the assignment correctly. Since we did not provide our test cases, participants were asked to test at their discretion, and failure in fulfilling contracts as specified was not detected by them. For example, we established the following postcondition for `Queue.remove`, using a JML-like notation

```
/**
```

p20's submission presented an implementation to `remove` ...

Surprisingly, no participant assigned to formal contracts reported any problems in their subjective answer — p13 and p20 suggested changes to contracts, arguing they were vague and should be made stronger, while p14 asked a simple question about the task; p1 even reported "the documentation and contracts (...) are clear". Even though developers understand the relevance of API formal contracts for the task, they may have misinterpreted the expected behavior, violating it with their implementation. In order to avoid such scenario, some sort of automatic verification would be critical. Since research has showed that developers often resist in applying formal specifications [], contract expressions amid textual specifications might be useful.

p17 – the only participant assigned to ContractJDoc whose implementation violated a contract clause – remarked he/she "*trusted the contract expressions*", which raises the issue of, by rejecting defensive programming [], one failing to notice contract restrictions, such as a basic assumption to ensure a post-condition clause. Again, test cases specific to the contract expressions should have made it easier to detect the nonconformance.

Four of those faults were submitted by implementers, which might be predictably more common, as API Clients could rely on simple tests or even additional compilation checks for not adding faults like the one added by p1. This outcome may also be explained by the awareness required in using methods provided by the API: one needs to read the documentation available in order to know how to use the methods, whereas implemention could be considered more straightforward – in this simulation, the APIs make up well-known data structures, thus their specifications may have been neglected – although none of the participants' comments explicitly evidence this speculation.

All faults in the experiment failed to ensure post-conditions. Studies show these contracts are the trickier to write [], and, by extension, might be also harder to follow (by API Clients)

or fulfill (by API implementers). It is also noticeable that no pre-condition violations were detected. We might speculate that, because pre-conditions are the most commonly used type of API contract [], developers are likely to be concerned about fulfilling them from the start. Pre-conditions might be as well a deliverance to developers in two ways: API clients are inclined to comply right way, before calling any method, so they soon accomplish their contract obligation; and API implementers can assume the pre-condition for neglecting defensive programming, writing thus less code.

Furthermore, qualitative data from participants do not present any reported issues with pre-conditions – eight quotes are, as a matter of fact, *compliments* to their clarity. On the other hand, nine participants make at least one remark about the trouble in understanding or accepting the post-conditions as they were provided.

6.2 RQ2. What is the Effect of the Documentation Approach on the Understanbility of API Specifications?

From the assessments by the participants in our experiment, no statistical effect was perceived in using either of the three approaches. All participants, in fact, perceived all API specifications as having medium to high understandability (Figure 7a).

Nevertheless, by analyzing the assessments in together with the provided qualitative data, we notice a trend: Javadoc as the top approach and formal contracts as the least understandable approach. The ContractJDoc approach, mixing Javadoc and contract expressions, was assessed as intermediate, which is illustrated in Figure 7b. This outcome is expected, as developers tend to favour informal styles for documentation – a known obstacle for the adoption of DBC []. In such scenarios, the approach employed in ContractJDoc is promising for a more gradual adoption of DBC in mainstream programming languages like Java.

Reinforcing this conclusion, the judgement survey with 142 respondents analogous results. In this case, however, differences between all groups are statistically significant – in detail, (effect sizes and pairwise) /ldots Larger effect sizes when formal contracts are compared with either Javadoc and ContractJDoc, and a smaller effect size between Javadoc and ContractJDoc (higher understandability for the first). ContractJDoc was considered understandable for 75% of the answers, if regard as such assessments 4 and 5.

Turning our attention back to experimental simulation data, participants reported higher understandability of API specifications when they were assigned to its implementation. Nevertheless, more submitted API implementations were faulty, in comparison to API clients (Table 4). We then could possibly infer that the relationship between the perceived understandability and actual outcome of using API contracts might be orthogonal, although statistical evidence is absent for a more credible conclusion.

6.3 RQ3. What Kinds of Nonconformances can be Uncovered if Javadoc Specifications are Replaced by Contract Expressions?

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. Those results corroborate with other research []

Data needed here: number of pre-, post-, invs- ...

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.

Data needed here: number of violations for each of two groupings: (1. by pre-, post-, inv-) (2. CommCase, AppSpec, Repet.) One paragraph of discussion for each.

Add Example. 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.

Add Examples. 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.

Add Examples. 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. **Discuss open source software evolution, with citations**.

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.

6.4 Limitations and Threats to validity

Decisions about the design of our studies were taken specifically to mitigate threats to validity. However, other threats remain.

Construct validity refers to correctly measuring the dependent variable – for our experimental simulation, correctness of the implementation and understandability of API specifications. There is always the risk that participants present respondent bias [], due to their knowledge about the experiment and the researchers (. . . we tried to give as less information about the study as possible).

Understandability assessment is specially dependent on the experience. . . our demographics questionnaire as answered by 27 developers, we discarded three because of experience level being discrepant. For the judgment survey, we have made similar arrangements for recruiting developers with comparable levels of experience, although uniformity is harder to achieve in that case. The order in which we display the documented interfaces on the survey form and the absence of open-ended 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.

For the case study, `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`, and 313 times bigger than `OOP Aufgabe3`. Therefore, results on those two systems are much more critical to the measurement of the dependent variable (number of nonconformances).

Internal validity refers to causation: are changes in the dependent variables necessarily the result of manipulations to treatments? A possible problem in our experimental design is that it is hard to determine the faults were introduced due to the assigned documentation approach. We tried to mix qualitative data with the experimental outcomes in order to have clearer view of task performing, but, despite our encouragement, developers provided less commentary than what we expected. Our speculations about the results certainly demand more experiments for inferring a causation relationship between faults and documentation approach. Also, all experimental and survey material was available only in English, but a large part of the respondents are likely to not have English as their first language, which may have affected their submissions and answers.

For the case study, our translation of Javadoc comments into contract expressions may be inaccurate, which could compromised the detected nonconformances. For that, two of the authors of this paper split the task of translating the comments, and one of them reviewed the translations carried out by other.

External validity refers to generalization. Conducting the experimental simulation on small APIs, with simple methods, may not be representative. Moreover, the fact that the authors defined the APIs contains the risk of bias. Still, we employed a factorial design using two participants for each treatment, varying the task (Client or Implementation) and the API (Queue or Stack), and observed a range of behaviours.

Likewise, due to its size, results from the case study cannot be generalized; its purpose is evaluating applicability of contract expressions. 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 actual inconsistencies between Javadoc comments and source code.

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 [8] 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 [11]. 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.

Recent work by Dietrich et al. [] catalogued 25 techniques, found within the code of 170 open-source systems, that are considered contracts. They include specialized constructs such as the ones offered by JUnit (e.g., `assertNotNull`), java asserts within the code itself, *ad hoc* methods with exceptions (e.g., Java `IllegalArgumentException`), in addition to external contract libraries, such as Guava). Their approach is based on the assumption that developers are more likely to adopt simpler forms of contracts, such as type annotations and assertions. The notion of contracts respects "the general assume-guarantee principle and follows the *Design by Contract* viewpoint promoted by Meyer, where contracts are viewed as lightweight specifications." []

Regardless of their use as lightweight runtime assertions, software designers may as well apply contracts as specification assets for APIs. Several applications of contracts promote them as relevant pieces of information for routines' users cite. For instance, Java asserts cannot be used as preconditions. Most constructs considered as lightweight contracts (runtime exceptions, Guava, Apache and Spring APIs, Java asserts) can only be found within methods' implementations, thus not amenable for API documentation, as we focus on this paper.

In consonance with our studies, the authors observe a prevalence of pre-conditions over post-conditions; suggested reasons for this result include library code reuse, in which modern libraries have to provide defensive APIs to deal with unknown clients. Given a large portion of the sample projects consists of APIs, most of the contracts are likely to be found in Javadoc commentary, which means that ContractJDoc, if used, could increase this number considerably.

Javadoc Comments. @TCOMMENT [19] 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 4.3).

Zhai et al. [22] 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.

Empirical Studies. There are three main related empirical studies about contract usage [16, 4, 2]. One common conclusion about is that, in practice, developers use simple and short contracts [16, 4]. For instance, [16] shows that 75% of the projects' Code Contracts are checks for the presence of data (e.g., non-null checks). In our case study (Section 4.3), 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. Also, Chalin studied 84 Eiffel [9] projects and pointed out that developers are more likely to use contracts in languages that support them natively, like Eiffel [9] or Code Contracts [1]. To support both conclusions, ContractJDoccould be used to write simple contracts natively, 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 [10] 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 [5, 20] 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 D. Dillman, J. Smyth, and L. Christian. *Internet, Phone, Mail, and Mixed-Mode Surveys: The Tailored Design Method*. Wiley Publishing, 4th edition, 2014.

4 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.

5 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.

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

7 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.

8 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.

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

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

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

12 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.

13 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.

14 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.

15 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.

16 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.

17 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`.

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

19 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.

20 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.

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

22 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.