Investigating the Trade-offs between Automatic Checking and Understandability in Java API Documentation

An Author
An Institution
author@authors.com

Abstract

Documenting public routines counts as a key benefit from applying the Design by Contract (DbC) methodology. In this context, DbC's pre- and post-conditions (public contracts) establish conditions and expected results, respectively, which may then be checked at runtime for detecting invalid calls or nonconforming implementations. Despite these benefits, however, programmers resist using formal contracts. Instead, Javadoc and natural language text are the dominant way that public contracts and APIs are documented. Verification assistance is unavailable. In this paper, we report results from empirical studies on integrating contract expressions into Javadoc comments used as public contracts. For this purpose, we designed a small tag-based extension to Javadoc (ContractJDoc) to express pre- and post-conditions among other standard tags. Studies consisted in (1) evaluating effectiveness and understandability of contract expressions within Javadoc, either for API clients or implementors, by means of a experimental study and a judgment survey, and in (2) investigating anomalies that may arise in open source systems with a high rate of Javadoc coverage, when we manually formalize textual public contracts into contract expressions, then checking conformance at runtime. We observed the higher quality of submissions as contracts were less formal, with satisfactory results when applying ContractJDoc with its semi-formal approach. Also, regarding understandability, differences were discerned between different contract styles with variate levels of formality. Given that we found 391 anomalies between contracts and code in one large and four small open source Java systems, the results are promising in establishing hypotheses to the adoption of contract languages in mainstream development, by integrating informal and formal styles of 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, specification languages

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Java programmers tend to consider writing Javadoc comments [1] as a good practice, especially when these comments enhance public methods – this case is especially crucial for API callers and implementors. Despite its recognised value and practice in Java community, understanding how to use public routines using contracts is widely ignored [31]. This situation occurs because the source of documentation is textual comments, which are potentially incomplete and ambiguous. Also, a well-known problem is that documentation and implementation tend to diverge over time [8]; a deveoper may forget to update the Javadoc documentation after performing an implementation change. Such a scenario may

produce faults related to requirements that could go unnoticed until late in the software process.

On the other hand, embedding contracts that follow the Design-by-Contract methodology (DbC) – pre- and post-conditions as checkable assertions for defining a public method's behaviour, which we refer as *public contracts* – has long been advocated by formal methods pioneers for program correctness [11, 21]. However, their adoption is limited [22]. Part of the reason is notational, for example, in Java, there is no built-in support for public contracts. To this end, developers might use contract frameworks like the *Java Modeling Language* (JML) [15] to express the full power of behavioural specifications.

Another reason for their low adoption is that formal contracts, while useful for dynamic checking or static analysis, do not meet documentation needs that are critical to public routines, as the case with third-party libraries [14, 21]. APIs usually do not provide their source code, so clients can only infer the expected behaviour by reading the public documentation, and formal contracts, in this scenario, are seldom applied. Tentative approaches do not seem to address those adoption issues [36]. Developers would undoubtedly benefit from the use of DbC as public contracts, although it seems undesirable discard the simplicity provided by textual Javadoc specifications (Section 2). Research is needed to investigate how different contract styles in public contracts affect programming tasks.

In this paper, we report results from empirical studies (Section 5) on integrating contract expressions into Javadoc comments used as public contracts. For this purpose, we designed and implemented a small tag-based extension to Javadoc (ContractJDoc) to communicate pre- and post-conditions as contract expressions. Its compiler translates these expressions into corresponding runtime assertions.

The first study evaluates the *effectiveness and understandability* [2] of contract expressions within Javadoc, for both API clients and implementors, employing an experimental simulation with 24 Java developers, using three contract styles: Javadoc text, ContractJDoc and JML-like formal contracts. A follow-up judgement survey with 142 Java developers was carried out for amplifying the enquiry on understandability for the contract styles. Finally, we investigated anomalies [1] in Javadoc-rich open source systems that may arise when we manually formalise textual public contracts into contract expressions, before checking conformance at runtime.

The case study provided evidence on the problem of nonconformance between textual Javadoc and the implemented behaviour: we found 391 anomalies between contracts and code in one large and four small open source Java systems. The detected anomalies reflect potential errors already present in the code or mismatches between the code and the contracts (i.e., possible errors in the contracts). Most public contracts detected are post-conditions, but often simple forms of behaviour specifications – more elaborate public contracts would potentially result in even more anomalies, because either more precise specifications would reveal more anomalies or longer specifications would be problematic themselves (Section 6).

Our studies with contract styles, in turn, brought to our attention interesting discussion topics and hypotheses for further studies. We observed the higher quality of submissions as contracts were less formal, with satisfactory results when applying ContractJDoc with its semi-formal approach. Also, regarding understandability, differences were discerned between different contract styles with different levels of formality. In general, Javadoc text was considered straightforward to understand and apply, but ContractJDoc expressions mixed

[1] we refer to these issues as *anomalies* because they are not necessarily bugs, but a mismatch between a specification and its implementation.

with textual presented better results in comparison to formal contracts (in the long tradition of Z specifications [35]), although opinions about understandability, and implementation outcomes, seem to be unrelated (Section 6).

The results are promising in establishing hypotheses to the adoption of contract languages in mainstream development, by integrating informal and formal styles of specifications for public routines. Broad adoption of DbC in this context would bring the benefit of automatic verification to avoid mismatches between documentation and actual behaviour, as long as issues with flexibility and clarity are adequately addressed.

1.1 Research Questions

This research work investigates the impact of integrating contract expressions with Javadoc comments in public interfaces. In particular, we intend to answer the following research questions:

RQ1. What is the Effect of the Contract Style on API Usage and Implementation Tasks?
We report and discuss quantitative and qualitative results from an experimental simulation with Java developers over development tasks involving APIs documented with three contract styles: textual Javadoc, ContractJDoc (Section 3), and JML-like formal contracts.

RQ2. What is the Effect of the Contract Style on the Understandability of API Specifications?
Using impressions given by the experimental simulation and a follow-up judgement survey, we discuss quantitative and qualitative data regarding the understandability of contract styles. Our approach to *understandability* is evaluate the comprehension of functionalities provided by a given routine, in terms of the expected inputs and outputs; however, it is a composite construct, whose measurement may be limited [16].

RQ3. What Kinds of Anomalies are Uncovered if Contract Expressions replace Javadoc Specifications?
We collected open source systems based on their use of Javadoc and applied contract expressions to each system, evaluating the result regarding detected anomalies (mismatches between specification and program behaviour). Also, we discuss the problems faced when replacing Javadoc comments by contract expressions in the described context.

2 Styles of Public Contracts

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

2.1 Javadoc and textual specifications

Javadoc [1] is the usual notation (and tool) for public routines (methods) in Java; it includes special tags (with symbol `@`) for structuring and pretty-printing code commentary. The Java Platform API specification itself [9], for instance, employs Javadoc for specifying *"contract[s] between callers and implementations."* In those terms, Javadoc may be a tool for applying the Design-by-Contract (DbC) methodology [18], 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, because it adds preciseness to the communication between client and implementor roles.

Consider the bank account interface depicted in Figure 1. 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` express,

```
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
  *
  */
 BigDecimal withdraw(BigDecimal amt)
   throws TransactionException {...}
 // ...
}
```

Figure 1 Bank Account Javadoc Specification.

respectively, a normal post-condition (if it works correctly) and an 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 with code changes [10], so it is hard to imagine that it would be applicable to contracts. Furthermore, the lack of formality leads to imprecision, ambiguity, and verbosity, potentially causing program anomalies and faults. On the other hand, using natural language does not require specialised training – although training may be needed to communicate ideas about program behaviour effectively – 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 [17]), or by extensions (Java Modeling Language (JML) [15] for Java and Code Contracts [3] for .NET languages) in mainstream programming languages. For Java, JML contracts may be defined as also showed 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 [2].

In this style, formal contracts 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 analysis [4]. Nevertheless, using JML-like formal contracts might require some level of training, becoming, to some extent, hard to read or write, and thus is often used sparingly [4, 22, 29]. Besides, if the code in Figure 2 were an interface, with no available code, the specification would be required. However, usually formal contracts are only available for the interface implementors, or not available at all [21].

Therefore, we face a dilemma concerning program documentation. If we use formal

[2] Usually, formal contract languages and extensions include **class invariants**, which establish conditions that must remain true between executions of a given class. In this paper, we do not focus on invariants since their use as public contracts is rare, as it would require public or abstract fields.

```
class BankAccount {
 BigDecimal balance;

 //@ requires amt.compareTo(new BigDecimal("0") > 0
 //@ && amt.compareTo(balance) <= 0;
 //@ ensures balance.equals(\old(balance.subtract(amt)));
 //@ ensures \result.equals(balance);
 //@ signals (TransactionException)
 //@   balance.equals(\old(balance));
 BigDecimal withdraw(BigDecimal amt)
   throws TransactionException {...}
 // ...
}
```

Figure 2 The JML specifications for the bank account.

contracts, the result is more precise documentation, with the possibility of automatic checks. Contrarily, by using an informal documentation approach such as Javadoc, we face the lack of precision and potential ambiguity, despite its flexibility and simplicity. This trade-off 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 an approach to API clients and implementors? This paper improve evidence on those questions.

3 A Javadoc Extension for Public Contracts

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 [24], providing support for runtime checking of the contracts. ContractJDoc allows programmers to document *contract expressions* amid Javadoc header for methods. With new tags, in addition to the standard Javadoc tags, one can write contracts as Javadoc comments that are compiled to runtime checkable code. In this section, we present, based on the example from Section 2, how ContractJDoc supports a mixed approach, which combines textual documentation with formal notation.

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 appropriate in this context since it is built over Java expressions, adding 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 are 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 artefact which is purpose-built for writing public specifications. This is especially true because the overwhelming majority of contracts that programmers write in practice are short and simple [8, 29]. For instance, in 75% of Code Contracts [3] projects, the written contracts are basic checks for the presence of data (e.g., non-null checks) [29]. In such scenarios, there is no additional effort in embedding

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

Figure 3 `withdraw`'s pre-condition.

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

Figure 4 `withdraw`'s post-condition.

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. Tag `@param` documents parameter `amt` of `BigDecimal` 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 pre-condition checking. An alternative (not showed) is to replace tag `@param` by `@requires` or `@pre`. Both can be used to document a pre-condition 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 pre-conditions, ContractJDoc offers three tags for expressing post-conditions. For normal post-conditions, 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 behaviour.

3.2 Supporting Infrastructure

The ContractJDoc compiler [5] is based on the open source AspectJML/ajmlc compiler [24, 25, 26]. Unlike the standard JML compiler, ajmlc presents code optimisations and improved error reporting [25]. Also, AspectJML enables the modularisation of crosscutting contracts that can arise in standard JML specifications [24].

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.

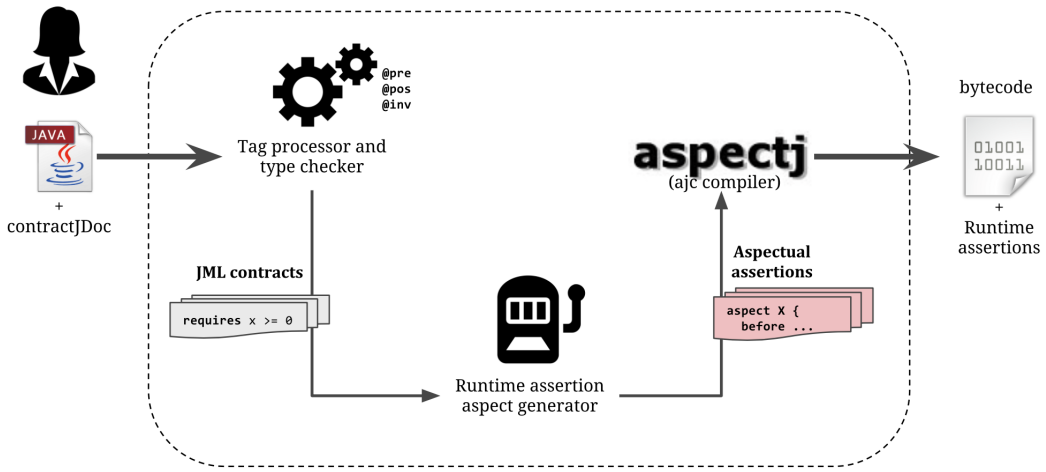


Figure 5 Compilation Infrastructure for ContractjDoc.

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 [30]: an experimental simulation (Section 4.1.1), a judgment survey (Section 4.2.1) and a case study (Section 4.3). The package for these studies are available online [20].

4.1 Experimental Simulation

In this study, we investigate the use of public contracts in simulated programming tasks, concerning effectiveness 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 a major R&D Institute in Brazil. Recruitment was carried out by invitation; from estimated 150 invited professionals, 24 of them accepted and later received the assignment material 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 is small for generalising any conclusions, we expect the observations to be considered as an

exploratory model to be checked by future studies with larger samples. For experimental simulations, our participant set is within the expected size [30].

4.1.2 Study Design

The study was designed to assess the effect of a given contract style on implementation tasks which depend on documented APIs. Participants were assigned to use a contract style (Factor 1) from the following: *Textual Javadoc,* ContractJDoc *and JML-like formal contracts*. Table 1 displays the treatments for those two factors. For each of those styles, two kinds of task are considered (Factor 2): a *Supplier* task, in which the participant must program the implementation of a Java API, and a *Client* task, which demands development of 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
Approach	ContractJDoc
	Textual Javadoc
	JML-like formal contracts
Task	Client
	Supplier

We use a factorial design [34], randomly assigning participants to each combination of treatments; a trial is the triple $< style, task, API >$. Since there are three contract styles, two types of task and two APIs, there are 12 possible trials, so each of them was carried out by two participants, resulting in a balanced design. The assignment of participants was random, to minimise bias.

4.1.3 Experimental Procedure

The experiment was performed offline, i.e., participants received the experimental package via an online survey platform 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 class that implements the API, or a client code using the API's methods. We also asked each participant to fulfil a pre-study questionnaire reporting their programming experience (concerning Java and contract-based programming experience).

During the assignment, participants were allowed to review additional information – part of the experimental package – about the assigned contract style. For *Supplier* tasks, participants should have implemented variations of well-known stack (or queue) operations whose intended behaviour was documented using one of the three contract styles. The following example shows the contract for `Queue.add`, written in textual Javadoc:

```
/**
   * 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 fulfil its external contracts, also presented using one of the three documentation options. Implementation of the API was provided as binaries, so participants were not given access to the source code. Before submitting the programs, we asked the participants to answer 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 adjust the questionnaires' structure. As a result, we changed the way of making the artefacts 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, indicating that a single package containing all Java classes should be located in a single URL.

4.1.4 Research Method

Each program sent by the participants were subjected to a test suite especially implemented by the researchers for checking whether every single contract was fulfilled – *there was a test case for each contract clause*. As a result, we can classify each submission according to the number of faults detected by these tests. To analyse the understandability assessments in the post-study questionnaire, we applied simple visualisation techniques (boxplots and bar charts) and statistical tests; in this case, we used the *Wilcoxon rank sum test* [13] for testing difference between groups, with non-normal data.

The answers from the 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. We used *open coding* [23] to analyse the collected answers, by inspecting them quote by quote and detecting categories, representing the key ideas in the data.

4.2 Judgment Survey

The goal of the survey is to compare three contract styles (Javadoc text, ContractJDoc and formal contracts) concerning understandability, from the point of view of Java developers.

4.2.1 Participants

We selected participants utilising a non-probability convenience sample [34]. The survey link was sent to academic and professional mailing lists. Also, 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

were sent the link – approximately a 20% response rate). From the 142 participants, 51 (36%) are professionals, and 91 are computer science 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 behaviours in response to specific questions [7], in a non-threatening way. The survey instrument begins with a purpose of clarification along with a consent term. Then, a characterisation of the respondent is given by 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 shown, then some questions related to the understanding of the behaviour of a class implementing the interfaces based on the comments available is asked.

We applied *Likert-scale* questions. In two questions we ask the developers to choose the most understandable documentation approach: one specific – related to the provided contract style; 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 adjust the survey's questions and structure. The developers who participated, however, did not report issues on the structure that we used for presenting the needed data for the participation in the study.

Regarding quantitative methods, we applied *Wilcoxon, and Kruskal-Wallis rank sum tests* [13] for comparing the results for the different groups of participants. Regarding survey results, we applied *one-way analysis of variance (ANOVA), Tukey HSD* and pairwise comparisons using *t-tests with Bonferroni correction* [13].

4.3 Case Study

This study aims at assessing the usefulness of contract expressions within Javadoc text, concerning automation benefits, from the point of view of Java developers.

4.3.1 Systems Selection

The case study was performed on a convenience sample: five Javadoc-rich open source systems available at GitHub[3] repository. The selected systems presented the highest rate of method-level Javadoc commentary including the following set of key phrases: *"must be", "must not be", "should be", "should not be", "greater than", "not be null", "less than"*. 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 test suite, which is run during the case study to detect anomalies. We were able to find five systems meeting these criteria. See Table 2 for details about the selected systems.

The manual translation abides by the following criteria: method-level comments were considered pre-conditions 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`

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

Table 2 Case study Systems. LOC shows the code lines (LOC), total contract clauses (#CC) and a brief description.

System	LOC	Description
ABC-Music-Player	1,973	Music player from ABC files
Dishevelled	110,577	UI components for complex data structures.
Jenerics	2,538	General-purpose set of Java tools and templates.
OOP Aufgabe3	353	Routines for polygon manipulation.
Webprotégé	74,742	Environment for ontology development.
Total	**190,655**	

`&& notes.size() >= 2]`, and post-conditions 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]`.

4.3.2 Experimental Procedure and Research Method

Three researchers applied ContractJDoc in five existing open-source systems available at GitHub (Table 2). 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). Figure 6 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 shown in Section 4.3.1); 2) compilation of the contracts by means of ajmlc-contractjdoc compiler, 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 analysed and anomalies are investigated.

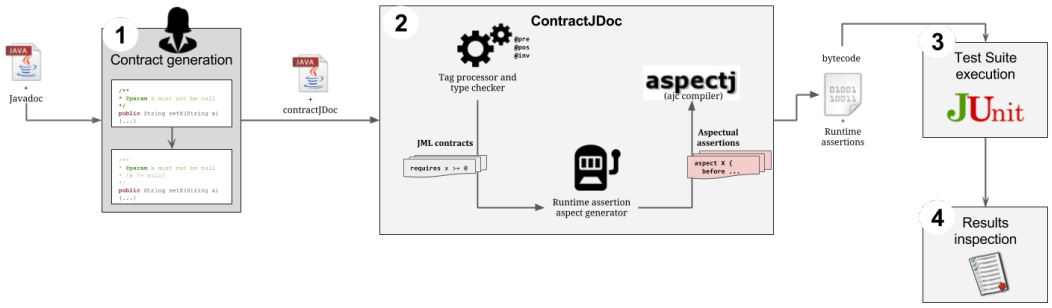


Figure 6 Steps for applying ContractJDoc to Javadoc-annotated systems.

We group the contracts according to the approach introduced by Schiller et al. [29]: 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.

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	×

All systems used in this study are available in a replication package.[4] Concerning the verification performed after applying ContractJDoc contracts into the systems, we used the test suites available with the purpose of identifying problems (four systems include a test suite). Every test case that failed prompted for further investigation for confirming the presence of an anomaly.

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. Besides, we enhanced the compiler and added features 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 summarise and discuss the observed effects. All the studies results are available online for replication purposes [27].

5.1 Experimental Simulation

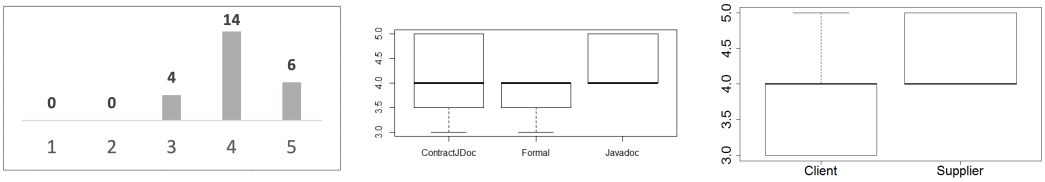
The participant set encompasses graduates in Computer Science or Software Engineering – either working in industry (41.6%) or M.Sc. and Ph.D. candidates (58.4%). In Table 3 we summarise the results from the 24 trials, from which five – 21% – were delivered with at least one fault detected by our test cases. All participants assigned to textual specifications in Javadoc delivered correct programs. By contrast, half of the programs delivered by participants using APIs with formal contracts were faulty (four out of eight). One faulty program was delivered by a participant assigned to ContractJDoc. Regarding task, faults were mostly in API implementations – four of them, if compared with only one in API clients.

Table 4 details some of the results from participants' submissions, including the faults detected by the test cases. After a detailed analysis of the programs, we classified each fault regarding the violated contract. *All clients fulfilled the specified pre-conditions for calling*

[4] `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 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**



(a) Total Results.

(b) Results by approach.

(c) Results by task.

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

API methods – p1's client raises an exception which was incompatible with the method's post-condition. *All four faults unveiled in API implementations resulted from failing to satisfy the specified post-condition* in methods removing data from the given structure (two on `Stack.pop` and two on `Queue.remove`).

After they submitted the programs, 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 summarised in Figure 7a, in which assessments are spread between 3 and 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. 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, the Kruskal-Wallis non-parametric test) showed no difference between the groups (p-value = 0.15, 95% confidence level). Concerning the task performed by the developers, raw data shows a slightly higher understandability by participants assigned to supplier tasks. Still, the Wilcoxon rank sum test [13] 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. The quotes were organised in 11 categories, which are presented in Table 5, in conjunction with the number of quotes; one quote may be classified in more than one category. Three participants did not provide an 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 a 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

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

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 [18]. 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

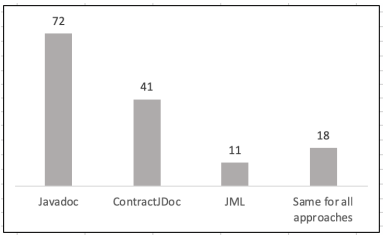
As a follow-up, we showed three versions of the Stack API as a web survey, asking questions about those approaches to contract specifications; we suggested respondents play the client role of the API as if they were about to call its methods. Respondents were asked to assess understandability of each 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 contract style – 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.

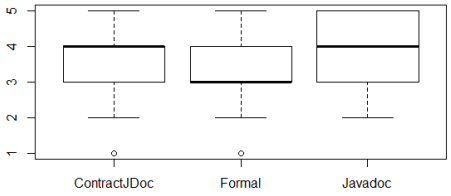
Concerning the survey answers, Figure 8a shows 50.7% (72) of the respondents chose Javadoc as the most straightforward approach to understanding the provided API specification. While only 11 (7%) chose formal contracts as the most understandable approach and 18 (12%) were indifferent, an almost one-third 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 contract style is showed in Figure 8b. 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 centre value.

Differently from the experimental simulation from Section 5.1, statistical tests show a *significant difference between the assessments*. The application of One-way Analysis of Variance (ANOVA), which is robust enough for non-normal distributions [13], detected a



(a) Surveyed Verdicts on Understandability.



(b) Assessment for Each Contract Style.

Figure 8 Results from the Judgement Survey.

distinction between the three groups of answers (p-value < 0.05). To estimate the assessment differences between each pair of groups, we applied post hoc analysis – Tukey HSD with Bonferroni correction [13]. Cohen's effect sizes are shown in Table 6, expressing significant difference between all styles. Using formal contracts as baseline, ContractJDoc presented greater understandability grades by a medium effect size; the same with Javadoc, but with a large effect size. With ContractJDoc as baseline, the effect of Javadoc over ContractJDoc is smaller, but also medium (d=0.343).

5.3 Case Study

Table 7 exhibits the results of translating real Javadoc specifications to ContractJDoc in each system. The detected anomalies reflect potential errors already present in the code or mismatches between the code and the contracts (i.e., possible errors in the contracts). Column **Anom** presents the anomalies detected by the original test suite (whose size is showed in Column **TCs**), with source code enhanced by ContractJDoc, while Columns **T** and **CJDT** respectively provide the seconds spent in compiling the system before and after ContractJDoc instrumentation, both before running the test cases. **Clauses** displays how many contract clauses were translated into contract expressions, which we classified as pre-conditions (**Pre**) or post-conditions (**Post**). The alternative classification for contract types – according to Schiller et al. [29] – is 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 anomalies, which provided us with an abundant source of analysis about the mismatches between specification and implementation. Regarding compilation overhead, instrumenting the code with runtime assertions added considerable time for compiling the systems. This process may be subject

Table 6 Effect sizes (Cohen) from Tukey HSD with Bonferroni correction

Group Pairs	Effect size
Javadoc → ContractJDoc	0.343
ContractJDoc → formal	0.516
Javadoc → formal	0.902

Table 7 Results from Case Study with ContractJDoc.

System	Anom	T(s)	CJDT(s)	TCs	Clauses	Pre	Post	CommCase	AppSpec	Repet
Dishevelled	381	59	434	2,643	2,661	1,411	1,250	1,542	151	968
Webprotégé	–	19	713	–	931	352	579	719	79	133
ABC-Music-Player	2	1	9	30	115	41	74	42	11	62
Jenerics	7	1	20	44	190	105	85	156	–	34
OOP Aufgabe3	1	1	4	11	54	28	26	16	30	8
Total	**391**	**82**	**1,185**	**2,728**	**3,951**	**1,937**	**2,014**	**2,497**	**282**	**1,214**

to optimizations, which were not our priority for this case study.

Concerning the kind of contracts, the only unit for which we wrote more application-specific contracts was the `OOP Aufgabe3` system (55% of the written contracts are application-specific). On the other hand, in `ABC-Music-Player`, more than 90% of the contracts remain between common-case (contracts that avoid unexpected cases, such as the absence of data, blank strings or empty collections) and repetitive code (contracts that repeat exact statements from the code). 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 remain 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 a basis, the research questions, and threats to validity.

6.1 RQ1. What is the Effect of the Contract Style on API Usage and Implementation Tasks?

Regarding code correctness, *all participants assigned to Javadoc APIs produced code fulfilling the contracts* – our manually-produced test cases did not detect any contract violation. One ContractJDoc API implementation was submitted 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 programs at their discretion – they had access to ajmlc to do runtime assertion checking in their own tests, and they did not detect a failure in fulfilling contracts as specified. For example, we established as post-condition for `AccountQueue.remove`, using JML-like formal contracts (Figure 9), that the return value be a non-null `CheckingAccount`. For this API, p20's submission presented an implementation that removed an account from the queue *neglecting a previous emptiness test*, which allows for null returns, then violating the contract.

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 behaviour, violating it with their implementation. To avoid such a scenario, some sort of

```
/**
public interface AccountQueue {
...
  /*@
   @ ensures \result != null && \result instanceof CheckingAccount;
   @ signals (AccountQueueException ex)
      (isEmpty() || headElement() instanceof SavingsAccount);
   @*/
  public Account remove() throws AccountQueueException;
```

Figure 9 AccountQueue.remove with formal contracts.

automatic verification would be critical. Since research has shown that developers often resist in applying formal specifications [22, 8], 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 [18], 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 anomalies.

Four of those faults were submitted by API implementors, 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 to know how to use the methods, whereas implementation 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 made any remark about it.

All faults in the experiment failed to ensure post-conditions. These contracts are trickier to write [28], and, by extension, might be also harder to follow (by API Clients) or fulfil (by API implementors). It is also noticeable that no pre-condition violations were detected. We might speculate that developers are likely to be concerned about fulfilling pre-conditions from the start.Furthermore, qualitative data from participants do not present any reported issues with pre-conditions – eight quotes are *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 Contract Style on the Understandability of API Specifications?

From the assessments by the participants in our experiment, no statistical difference was perceived in using either of the three styles. *All participants perceived all API specifications as having medium to high understandability* (Figure 7a). Nevertheless, by analysing the assessments 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 [22]. In such scenarios, the approach employed in ContractJDoc is promising for more gradual adoption of DbC in mainstream programming languages like Java.

Reinforcing this conclusion, the judgement survey with 142 respondents similar 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 we regard 4 and 5 scales as such.

If we turn 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 infer that *the relationship between the perceived understandability and actual outcome from using the public contracts might be orthogonal*, although statistical evidence is absent for a more credible conclusion.

6.3 RQ3. What Kinds of Anomalies are Uncovered if Contract Expressions replace Javadoc Specifications?

For all systems (see Table 2), *we translated more post- than pre-conditions.* Post-conditions are often the most significant and relevant decisions about the requirements [28, 19], so the result is expected. For `ABC-Music-Player` and `WebProtégé` projects, we detected and translated almost twice as many post-conditions as pre-conditions. In `ABC-Music-Player` this is related to the number of accessor methods available and for `WebProtégé` – the difference is due to the comments available.

As example, we detected anomalies in `ABC-Music-Player`; two of the pre-conditions in class `sound.Utilities` are violated by client classes (e.g. class `MainTest`). Javadoc comments such the following excerpt from `sound.Utilities` allowed trivial translations:

```
/**
    * Computes the greatest common divisor between two integers.
    * @param a - must not be >= 0 [a >= 0]
    * @param b must be > 0 [b > 0]
    ...
**/
```

One of the client classes tried to call such method with b as 0 (zero). Another example of anomaly was found in the `WebProtégé` system, regarding exceptional post-conditions. The Javadoc from interface `OWLLiteralParser.parseLiteral` state the following post-conditions:

```
/**
    * @return The {@link OWLLiteral} that {@code text} was parsed into.
    * @throws OWLLiteralParseException If the literal text is malformed.
    * @throws NullPointerException
         if {@code text} is {@code null}
         or {@code language} is {@code null}.
         [text == null || language == null]
    */
```

In this case, the implementation for `parseLiteral` does not throw `NullPointerException` when the indicated scenario occurs. These examples suggest the main reason behind these anomalies: *the requirements changed during the system's lifecycle, causing the Javadoc to be obsolete in comparison with the program.* In this scenario, incentives to update the specification are non-existent, since they are pure text – analysis of conformance is hardly automatable.

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. Nevertheless, we were able to detect and correct anomalies between source code and comments.

6.4 Threats to validity

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

Construct validity refers to correctly measuring the dependent variable – for our experimental simulation, the correctness of the implementation and understandability of contract styles. There is always the risk that participants present *respondent bias* [7], due to their knowledge about the experiment and the researchers; we expect to mitigate this issue by providing as less information about the study as possible in the experimental package sent to the participants.

Our demographics questionnaire as answered by 27 developers, but we discarded three submissions due to experimental balance and discrepant experience levels (in this case, too low). 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 threaten the construct validity. For dealing with these threats, we performed a pilot before applying the survey and used the results from the pilot to improve the survey structure.

A known threat is related to the *understandability construct*, which is complex and not directly measurable. We limited the concept to the direct comprehension of functionalities provided by a given routine, in terms of the expected inputs and outputs [16]. In the questions, it is explicitly stated that readers "*should grade how they were able to grasp as funcionality and restrictions from the contracts*"; yet, we assume that answers might have deviated from its original purpose, for which there is no trouble-free solution [2].

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 would be more critical to the measurement of the dependent variable (number of anomalies). Still, `WebProtégé` did not include test cases – we did not generate new tests, to avoid additional bias –, so only `Dishevelled` end up as a significant source of anomalies.

Internal validity refers to causation: are changes in the dependent variables necessarily the result of treatment manipulation? A possible problem in our experimental design is that it is hard to determine whether faults were introduced due to the assigned contract style. We tried to mix qualitative data with the experimental outcomes to have a more transparent 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 compromise the detected anomalies. For that, three 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 generalised; its purpose is evaluating the applicability of contract expressions. The sample is not representative since there is no available estimate for projects of interest in GitHub, then a probability sample is hard to achieve. Our approach is as systematic as feasible in selecting the evaluated project – manual translation does not scale, so the sample contains only five projects. Therefore, those systems may not be representative of the real use of Javadoc in real systems; however, we were able to detect real 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 [15] and Microsoft's Code Contracts [3]. Both enable one to provide full behavioural specifications and their runtime checking. They differ in the way they are written; the former is written as Java comments in the 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 public contractss since they are embedded in C# source code. 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 public contracts.

Recent work by Dietrich et al. [6] 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." [18] Several applications of contracts promote them as relevant pieces of information for routines' users [36]. For instance, Java asserts are seldom used as pre-conditions. 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 public contracts, as we focus on this paper. Nevertheless, in consonance with our studies, they 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 implementations to deal with unknown API clients. Most of their study's sample projects are APIs, so it is likely that Javadoc commentary contained public contracts, but were not considered in their analysis.

Javadoc Comments. @TCOMMENT [32] is an approach to 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 can write contracts richer than those for checking null values and exceptions (as presented in Section 4.3).

Zhai et al. [36] present a technique that builds models for Java API functions by analysing the documentation. Their models are simpler implementations in Java compared to the original ones and hence easier to analyse. More importantly, they provide the same functionalities as the original functions. They argue that API documentation, like Javadoc and .NET documentation, usually contains information about the library functions, such as the behaviour 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 to improve model generation.

Empirical Studies. There are three main related empirical studies about contract usage [29, 8, 4]. One common conclusion about is that, in practice, developers use short and straightforward contracts [29, 8]. For instance, [29] 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 [17] projects and pointed out that developers are more likely to use contracts in languages that support them natively, like Eiffel [17] or Code Contracts [3]. To support both conclusions, ContractJDoccould be used to write simple contracts natively, as traditional Javadoc comments.

8 Conclusions

In Java programs, textual Javadoc prevail for documenting API behaviour [36]. Javadoc's use is widespread, serves for important purposes – tools for generating documentation – but lacks, of course, automatic checking of the program against its specification. The trade-off between the pragmatics of textual descriptions and formal contracts amenability to automatic analysis is explored in this paper, by three empirical studies. We evaluated the effect on API usage and implementation tasks of three public contract styles in Java programs: textual Javadoc, JML-like formal contracts, and a third option, a small tag-based extension to Javadoc to express pre- and post-conditions amid text and other standard tags, which we call ContractJDoc. The first and second studies evaluated the effect of specification styles on the efficacy and understandability of contract expressions, either for API implementors or users, through an experimental simulation with 24 recruited participants, and a judgment survey with more than a hundred Java developers. In the third study, we evaluated the potential of detecting anomalies by manually formalising textual Javadoc contracts from open source Java systems into contract expressions, then checking conformance at runtime.

Javadoc and ContractJDoc groups were more successful in submitting anomaly-free programs, while half of the formal contract group violated at least one contract clause. We speculate, by this outcome and by analysing the qualitative data provided by the participants, that developers, despite their awareness of the specified behaviour, were having trouble in interpreting the expressions. Moreover, all anomalies happened with post-conditions, which matches conclusions from previous research [8, 19] perceiving the discomfort when dealing with this kind of contract, in comparison with pre-conditions.

We also used the experimental setting for assessing understandability [2] of three contract styles. In the answers, Javadoc contracts were regarded as the most understandable, followed by the mixed ContractJDoc style and, as the least understandable; nevertheless, those

differences were not statistically significant. As a follow-up study, we carried out a judgement survey with 142 Java developers with at least one-year experience, in which the results were the same, this time with significant pairwise-difference effect sizes between all three styles. This outcome is expected, as developers tend to favour informal styles for documentation [22]. It remains to be investigated whether an intermediate approach such as ContractJDoc is understandable enough for fostering the use of DbC in API specifications.

Finally, our case study applying contract expressions in open source systems resulted in finding 381 anomalies in a single Javadoc-rich system (`Dishevelled`) and ten more in other three toy systems (`ABCMusicPlayer`, `Jenerics` and `OOP Aufgabe3`), showing the potential of using contract expressions in existing Javadoc specifications. Other interesting findings about those systems include their preference for post-conditions over pre-conditions, although most contracts were classified as *common case* – enforcing expected program properties, such as nullable variables.

Despite their limitations, we believe the studies presented in this paper prompt the research community to investigate some hypothesis with future enquiry. Making sense of the specified behaviour in contracts is undoubtedly a necessary condition to providing a program that conforms to the specification, but it is not sufficient; other forces should be considered, such as tool support, enforcing test cases or even the developer's lack of discipline. Additionally, we could investigate whether contract conformance may be undermined by the omission of defensive programming promoted by DbC – as it becomes pointless to add program checks to certain scenarios, other important conditions might be neglected.

In DbC – especially for public contracts –, pre-conditions seem easier to regard and fulfil, if compared to post-conditions. However, post-conditions are probably more fundamental to elaborate and useful contracts [28]. Research could be carried out to test this hypothesis, and investigate the rationale behind the observations. Finally, the reason for the low adoption of DbC, at least for formal contracts, must be subject of further consideration. As we could observe in our case study with open-source systems, keeping the specification in sync with the program seems to be the main issue, requiring extra resources often not available in software teams with tight schedules. Automatic generation of contracts from programs [36, 32] seems a reasonable option to address this issue.

For ContractJDoc, we intend to improve the tags for considering more complex contract expressions, and carry out more studies on its usability and efficiency, in particular for specifying APIs – adaptation to other programming languages, such as C#, would be relatively straightforward. For instance, we could add support to *autocomplete*, applying Jaro-Winkler string distance [12, 33] as a basis. Moreover, the overhead ContractJDoc adds to the instrumented code must be addressed with optimizations, perhaps using an alternative framework rather than Aspectj as foundation.

References

1 Javadoc Technology. `https://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/index.html`. Accessed: 2018-01-10.

2 Automatically Assessing Code Understandability: How far are we? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 417–427, 2017.

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

4 P. Chalin. Are Practitioners Writing Contracts? In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 100–113. Springer-Verlag, 2006.

5 ContractJDoc Compiler. `https://www.dropbox.com/sh/x5ai73u3jmjqyb3/AADxX6-y4KUJk968woS36jEFa?dl=0`, 2019. Accessed: 2018-01-11.

6 Jens Dietrich, David J Pearce, Kamil Jezek, Premek Brada, and Jens Dietrich. Contracts in the Wild: A Study of Java Programs. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, 2017.

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

8 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.

9 James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

10 Dan Hao, Tian Lan, Hongyu Zhang, Chao Guo, and Lu Zhang. Is This a Bug or an Obsolete Test? In *Lecture Notes in Computer Science*, 2013.

11 C. A. R. Hoare. An Axiomatic Basis For Computer Programming. *Communications of the ACM*, 1969.

12 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.

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

14 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.

15 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.

16 J. Lin and K. Wu. A Model for Measuring Software Understandability. In *Sixth IEEE International Conference on Computer and Information Technology (CIT'06)*, pages 192–192, Sep. 2006.

17 B. Meyer. Eiffel: Programming for Reusability and Extendibility. *ACM SIGPLAN Notices*, 22(2):85–94, 1987.

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

19 Alysson Milanez, Bianca Lima, José Ferreira, and Tiago Massoni. Nonconformance between programs and contracts: a study on C#/code contracts open source systems. In *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 1219–1224, 2017.

20 Studies Package. `https://www.dropbox.com/sh/hcymky5um99kuas/AADktcMTOUId9o5I4REzXniQa?dl=0`, 2019. Accessed: 2018-01-11.

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

22 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.

23 Jason Matthew Cameron Price. Coding: Open Coding. In *Encyclopedia of Case Study Research*. 2010.

24 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.

25 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.

26 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.

27 Data (Results). `https://www.dropbox.com/sh/26ky9fvg54vch1z/AADqLCBAczL3hcTtyKyYDF1aa?dl=0`, 2019. Accessed: 2018-01-11.

28 D.S. Rosenblum. Towards A Method Of Programming With Assertions. *International Conference on Software Engineering*, pages 92–104.

29 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.

30 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.

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

32 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.

33 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.

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

35 Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall International, 1996.

36 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.