Contracts in the Wild: A Study of Java Programs*

Jens Dietrich[1], David J. Pearce[2], Kamil Jezek[3], and Premek Brada[4]

1 **School of Engineering and Advanced Technology, Massey University**
Palmerston North, New Zealand
j.b.dietrich@massey.ac.nz

2 **School of Engineering and Computer Science**
Victoria University of Wellington, Wellington, New Zealand
djp@ecs.vuw.ac.nz

3 **NTIS – New Technologies for the Information Society**
Faculty of Applied Sciences, University of West Bohemia, Pilsen, Czech Republic
kjezek@kiv.zcu.cz

4 **NTIS – New Technologies for the Information Society**
Faculty of Applied Sciences, University of West Bohemia, Pilsen, Czech Republic
brada@kiv.zcu.cz

Abstract

The use of formal contracts has long been advocated as an approach to develop programs that are provably correct. However, the reality is that adoption of contracts has been slow in practice. Despite this, the adoption of lightweight contracts — typically utilising runtime checking — has progressed. In the case of Java, built-in features of the language (e.g. assertions and exceptions) can be used for this. Furthermore, a number of libraries which facilitate contract checking have arisen.

In this paper, we catalogue 25 techniques and tools for lightweight contract checking in Java, and present the results of an empirical study looking at a dataset extracted from the 200 most popular projects found on Maven Central, constituting roughly 351,034 KLOC. We examine (1) the extent to which contracts are used and (2) what kind of contracts are used. We then investigate how contracts are used to safeguard code, and study problems in the context of two types of substitutability that can be guarded by contracts: (3) unsafe evolution of APIs that may break client programs and (4) violations of Liskov's Substitution Principle (LSP) when methods are overridden. We find that: (1) a wide range of techniques and constructs are used to represent contracts, and often the same program uses different techniques at the same time; (2) overall, contracts are used less than expected, with significant differences between programs; (3) projects that use contracts continue to do so, and expand the use of contracts as they grow and evolve; and, (4) there are cases where the use of contracts points to unsafe subtyping (violations of Liskov Substitution Principle) and unsafe evolution.

1998 ACM Subject Classification D.1.5 Object-oriented Programming, D.2.4 Software/Program Verification, D.3.3 Language Constructs and Features

Keywords and phrases verification, design-by-contract, assertions, preconditions, postconditions, runtime checking, java, input validation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.9

* This project was supported by a gift from Oracle Labs Australia to the first author and by the Ministry of Education, Youth and Sports of the Czech Republic under the project PUNTIS (LO1506) under the program NPU I.

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 9; pp. 9:1–9:29

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

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.6`

1 Introduction

The idea of providing formal specifications of computer programs in the form of *pre-* and *post-conditions* has a long history in Computer Science. The seminal works of Floyd, Hoare, and Naur proposed rigorous techniques for reasoning about programs and establishing their specifications [65, 58, 82]. Hoare, for example, provided an axiomatic means for relating pre-conditions to post-conditions. By the mid-seventies the vernacular of contracts, specifically pre- and post-conditions, was widespread. The idea of one program mechanically verifying another soon arose, and early efforts included that of King [74], Deutsch [46], the Gypsy Verification Environment [61] and the Stanford Pascal Verifier [78].

A *verifying compiler*, following Hoare's vision, "*uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles*" [64]. The modern era of verifying compilers can be traced back to the pioneering work at Compaq Systems Research Center which led to the Extended Static Checker for Modula-3 and subsequently for Java [45, 57]. Since then a variety of other tools employing contracts have blossomed, including JML [42], Spec# [18, 19], Dafny [76, 77], Why3 [56], VeriFast [69, 68], Frama-C [43, 62] and Whiley [84, 85]. Spark/ADA is a notable exception as a commercially developed system used extensively in industry [70, 17]. Examples of this include *space-control systems* [28], *aviation systems* [37], *automobile systems* [66] and *railway systems* [50].

At this point we must acknowledge that, despite some success stories, tools for compile-time checking of contracts are not in widespread use [27, 81]. Spec# is a pertinent example as a project that aimed to "*build a real system that real programmers can use on real programs to do real verification*" [18]. But, despite considerable investment, the project failed to deliver on this and wrapped up without making it into production. [1] However, one idea stemming from the project has made its way into production. Specifically, *Code Contracts* were introduced in .NET 4.0 which, essentially, constitutes a library for static and runtime checking of pre- and post-conditions [55].

1.1 Contracts and Their Checking

Whilst the adoption of static verification has been hampered by a lack of effective tooling, runtime contract checking remains a cost-effective and pragmatic alternative [39]. Empirical studies have consistently shown runtime contracts as effective at identifying faults and aiding diagnosis [92, 95, 15, 32]. Testing and coverage frameworks compound these benefits by giving mechanisms to exercise contracts and establish when a program is "correct enough" [63, 88].

Our notion of contract respects the general assume-guarantee principle and follows the *Design by Contract* viewpoint promoted by Meyer [80], where contracts are viewed as lightweight specifications: *"The principles of Design by Contract form the basis of the Eiffel approach and account for a good deal of its appeal. Eiffel's contracts are the result of a design trade-off between the full extent of formal specifications and what is acceptable to practicing software developers."*

[1] Despite these comments, we do believe the project was a success in many respects and has helped to advance the field considerably.

A key observation here is that *usability* is as important as the strength of the formalism. That is, techniques which are heavy in formalism and specialized syntax have a low chance of being adopted by ordinary programmers [80]. Simpler forms like type annotations and assertions should therefore have higher adoption rates in general. As an example, Hoare reported that the Microsoft Office source code contained (at that time) around 250M runtime assertions [63].

In practice, contracts manifest themselves in a variety of ways: firstly, testing frameworks typically provide specialised constructs (e.g. JUnit's `assertNotNull()`); secondly, most languages support runtime assertions (e.g. Java `assert`) within the code itself; finally, one can always utilise more ad-hoc methods (e.g. Java `IllegalArgumentException`) and, indeed, a number of libraries have sprung up here (e.g. Guava with its `Preconditions.check*` methods, etc). There are also specific language extensions which support contracts to various degrees. For example, Eiffel [79] and the contract languages of JML [75] and Spec# [18] support runtime contract checking.

1.2 Contracts and Evolution

Another aspect related to the use of contracts in practice is *evolution* — that is, how the contracts vary between different versions of a program and how this can affect its clients. This is important with the prevalence of modern build tools, like Maven and Gradle, which automate dependency resolution. Frameworks like OSGi [98] take this further and resolve dependencies at runtime against components supplied via repositories. Such systems support declarative dependencies using version ranges and, oftentimes, checks normally performed at build time (e.g. testing) are bypassed as dependencies are automatically updated at deployment or runtime. In this context, contracts of different kinds [24] play an important part to safeguard this process of *composition* using *"contractually specified interfaces"* [96]. This is especially true if they can be aggregated in computed and automatically enforced meta-data such as semantic versions [89].

1.3 Research Questions and Contributions

This paper is concerned with how contracts are used in practice in the world of Java programs. We first examine a number of different ways that contracts can manifest themselves in Java. Then we investigate two related issues: firstly, whether contracts are actually being used and how often; secondly, how they evolve and whether or not they identify breaking changes in client-supplier composition. Specifically, we try to answer the following research questions:

RQ1 *Which language features are used to represent contracts in real-world Java programs?*
RQ2 *How does the use of contracts change throughout the evolution of a program?*
RQ3 *Are contracts used correctly in the context of program evolution in real-world Java programs?*
RQ4 *Are contracts used correctly in the context of subtyping in real-world Java programs?*

Note, RQ4 can be rephrased roughly as: *are there contract-based violations of Liskov's Substitution Principle in real-world Java programs?* In an attempt to answer these questions, we performed a detailed analysis of a data set extracted from the Maven Central repository of Java-based program artefacts which is unbiased with respect to contract use. The contributions of this paper are:

1. We present a classification of contract constructs in existing Java programs and a lightweight static analysis for their identification. Our analysis looks for patterns in the

program source, e.g. the use of Java `assert`, throwing of `IllegalArgumentExceptions`, use of various contract APIs (such as Guava's `Preconditions`) and annotations (like JSR303 and JSR305). Altogether, we investigated the presence of 25 different techniques to represent contracts.

2. We report on an empirical study of 176 projects with 6,934 versions hosted on Maven central, constituting 351,034 KLOC. Our findings suggest that: firstly, contracts of different types are being used (though less than might perhaps be expected); and, secondly, that problems with respect to contracts do indeed arise in the wild in the contexts of subtyping and evolution.

2 Contract Patterns in Java

2.1 Terminology

For the purpose of our analysis, we consider a contract as composed of *contract elements*. Contracts are associated with code artefacts such as methods, fields or classes. The contract elements associated with a method are *pre-* and *post-conditions* which specify the constraints on its input and output values, representing the methods's assumptions and guarantees, respectively. We also consider *class invariants* as contract elements associated with classes and fields. Class invariants are not associated with particular methods, but apply to all (public) methods of the respective class. According to Meyer [80], class invariants can be considered as quantified contract elements for all (public) methods of a class: *"In effect, then, the invariant is added to the precondition and postcondition of every exported routine of the class"*.

Contracts can be used for static verification and/or evaluated at runtime. Contract elements generally fit into the pattern *condition-action-message*, though this is sometimes hidden or implicit. That is, a *condition* that can be evaluated to `true` or `false`, indicating whether the constraint is satisfied or not, and an *action* that is executed in case the constraint is violated. A contract element might also include an optional *message* to provide additional information useful for diagnosis. If the condition and action are explicit, then the element carries its own *enforcement semantics*. For instance, this is the case for assertion-based contracts in Java: at runtime, if assertion checking is enabled and the evaluation of the asserted expression fails, an `AssertionError` is created and thrown. If a contract element is not associated with a condition and action, the enforcement semantics is provided by other means such as naming convention, tooling or documentation. For instance, this is the case for certain annotation-based contracts where pluggable annotation processors are used for this purpose.

Our notion of contract element corresponds to assertions used by Meyer [80] and in Eiffel: *"Eiffel encourages software developers to express formal properties of classes by writing assertions, which may in particular appear in the following roles: .. routine preconditions .. routine postconditions .. class invariants"* [2]. Unfortunately, the term assertion has a slightly different meaning in Java as it is associated with the `assert` statement. As we will discuss in more detail below, `assert` statements can be used to write post-conditions and class invariants, but they are not suitable for pre-conditions. Furthermore, they can also be used in a manner where they do not represent any contract element.

[2] `https://archive.eiffel.com/doc/online/eiffel50/intro/language/invitation-07.html` (accessed 10 January 2017)

Table 1 Contract constructs and their classification.

Category	Example constructs

CREs (2 types)	`IllegalArgumentException`
	`IllegalStateException`
	`NullPointerException`
	`IndexOutOfBoundsException`
	`ArrayIndexOutOfBoundsException`
	`StringIndexOutOfBoundsException`
	`UnsupportedOperationException`

APIs (4 types)	`com.google.common.base.Preconditions.*` (Guava)
	`org.apache.commons.lang3.Validate.*`
	`org.springframework.util.Assert.*`

Assertions (1 type)	`assert` (Java)

Annotations (17 types)	`javax.annotation.*` (JSR305)
	`javax.annotation.concurrent.*` (JSR305)
	`javax.validation.constraints.*` (JSR303, JSR349)
	`org.jetbrains.annotations.*`
	`org.intellij.lang.annotations.*`
	`edu.umd.cs.findbugs.annotations.*`

Other (1 type)	(jContractor)

In the following subsections we discuss the various *categories* of contract element patterns and forms we investigated, and for each one provide examples of concrete *types* of constructs by which they are expressed. The list of categories and the initial set of types was extracted from a study of academic and grey literature (*wikipedia*, *stackoverflow*, *c2.com*). Table 1 summarises the classification. The numbers in the first column indicate the number of patterns found in the respective category; the total number of patterns we considered is 25.

2.2 Conditional Runtime Exceptions (CRE) and Unsupported Operations

This is the most basic approach, and constitutes throwing an exception on condition failure, enforcing the contract at runtime. In *Effective Java*, Bloch suggests using runtime exceptions to indicate programming errors, typically pre-condition violations [26, item 58]. Rudimentary support is provided in the Java standard library through exceptions specifically aimed at signalling violations, such as `IllegalArgumentException`. Listing 1 illustrates an example.

We are particularly interested in these runtime exceptions: `IllegalStateException`, `IllegalArgumentException`, `NullpointerException`, `IndexOutOfBoundsException`, `UnsupportedOperationException` (all in the `java.lang` package). Of these, `UnsupportedOperationException` is especially interesting as it indicates when a method is unavailable. This models the semantics of optional methods (such as `Iterator.remove()`), and also the absence of platform-specific operations (e.g. for the user interface). In this sense, `UnsupportedOperationException` represents the strongest possible pre-condition that cannot be satisfied by any caller. The common usage pattern is that a method only instantiates and throws the exception *without* using a guard condition.

```
1 static public double binomial(int k, int n, double p) {
2 if( (p < 0.0) || (p > 1.0) )
3 throw new IllegalArgumentException();
4 if( (k < 0) || (n < k) )
5 throw new IllegalArgumentException();
6 ...
7 }
```

Listing 1 Use of conditional runtime exceptions in pre-condition checks in `cern.jet.stat.Probability` (in *colt 1.2.0*).

```
1 public static void checkArgument(boolean expression) {
2 if (!expression) {
3 throw new IllegalArgumentException();
4 }
5 }
```

Listing 2 Contract API method defined in `com.google.common.base.Preconditions` (in *Guava 19.0*).

2.3 Contract APIs

The next level of sophistication is to provide a *contract API* consisting of wrappers around conditional exceptions (see for example Listing 2). This provides a potentially richer language for expressing contracts, conveys the programmer's intention more clearly, and introduces less clutter. Contract API methods are typically facilitated by making them `static` (i.e. to be used as though locally defined via static imports). Static methods also facilitate fast execution as static dispatch (via `invokestatic`) is used.

The popular *Guava* [6] library contains the `com.google.common.base.Preconditions` class with multiple static `check*` methods (e.g. `checkArgument()`, `checkState()`, etc). The documentation stipulates this class contains *"Static convenience methods that help a method or constructor check whether it was invoked correctly (whether its preconditions have been met)"*. [3] The same document also indicates that these methods are not to be used for other checks (including post-condition and invariant checks): *"It is of course possible to use the methods of this class to check for invalid conditions which are not the caller's fault. Doing so is not recommended because it is misleading to future readers of the code and of stack traces."*. Listing 3 shows some code from the *Hadoop* project illustrating the use of Guava to represent a pre-condition.

Likewise, Apache Commons provides the class `Validate` [1] with similar semantics which the documentation states *"assists in validating arguments"*.[4] Other examples are *Spring Assert* [12] (`org.springframework.util.Assert`) and *valid4j* [13] which has similar goals and is notable for using the hamcrest internal DSL [7] for representing conditions.

There are two caveats concerning contract APIs in practical use. Firstly, they introduce some performance overhead, because the message is always constructed and the condition

[3] `https://google.github.io/guava/releases/19.0/api/docs/com/google/common/base/Preconditions.html` (accessed 10 January 2017)

[4] `https://commons.apache.org/proper/commons-lang/javadocs/api-3.1/org/apache/commons/lang3/Validate.html` (accessed 10 January 2017)

```
1  import com.google.common.base.Preconditions;
2  ...
3  FileDistributionCalculator(Configuration conf,
4  long maxSize,int steps,PrintWriter out) {
5  this.conf=conf;
6  this.maxSize=maxSize==0?MAX_SIZE_DEFAULT:maxSize;
7  this.steps=steps==0?INTERVAL_DEFAULT:steps;
8  this.out=out;
9  long numIntervals=this.maxSize/this.steps;
10 // avoid OutOfMemoryError when allocating an array
11 Preconditions.checkState(numIntervals<=MAX_INTERVALS,
12 "Too many distribution intervals (maxSize/step): " +
13 numIntervals + ", should be less than " +
14 (MAX_INTERVALS+1) + ".");
15 this.distribution=new int[1+(int)(numIntervals)];
16 }
```

Listing 3 Use of the Guava contract API in `org.apache.hadoop.hdfs.tools.offlineImageViewer.FileDistributionCalculator` (in *Hadoop 2.5.0*).

evaluated completely (i.e. to pass them to the contract API method). With conditional exceptions, error messages are only constructed if the condition is violated. The Guava documentation explicitly recommends reverting to conditional exceptions in performance-critical situations for this reason. Secondly, the use of APIs adds a dependency to projects. This makes the use of APIs a less obvious choice. A good example is the decision of the *ElasticSearch* project to remove the use of the Guava pre-condition API for those reasons[5].

Apart from the example APIs mentioned above, it is possible that further contract APIs exist. In some cases, these are only defined and used locally within the scope of a certain project or a group of related projects. One such case will be discussed in section 4.2.

2.4 Assertions

Java has supported assertions through the `assert` keyword since version 1.4 released in 2002. Assertions implement runtime checks by evaluating boolean conditions. If this check fails, an error (`AssertionError`) is thrown.

By default, runtime assertion checking is disabled and an explicit parameter must be used in order to switch assertion checking on when the JVM starts. While the ability to switch off assertions centrally is useful for addressing performance overhead, this has some implication on how assertions can be used. Most importantly, `assert` statements are not primarily intended for checking pre-conditions. An Oracle tech note warns: *"Do not use assertions to check the parameters of a public method. An assert is inappropriate because the method guarantees that it will always enforce the argument checks. It must check its arguments whether or not assertions are enabled."*[6] The same note then outlines the use of asserts in invariants and post-conditions. The note explicitly suggests how to use assertions for class invariants. However, the suggested pattern does not fully comply to the definition

[5] `https://github.com/elastic/elasticsearch/issues/13224` (accessed 10 January 2017)
[6] `https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html#preconditions` (accessed 10 January 2017)

```
1 import javax.validation.constraints.*;
2 ..
3 @Max(-42)
4 public int negate(@Min(42) int i) {..}
```

Listing 4 Using JSR303 annotations for pre- and post-conditions.

of class invariants according to Meyer [80] that requires that the invariants are applied to all public methods of a class. In many cases, the invariants expressed by assertions are *method-local* invariants, such as control flow invariants.

2.5 Contract Annotations

The idea of annotation-based approaches is to add meta-data to artefacts (methods, fields, classes and method parameters) that describe their validity. The standard Java annotation API is widely used for implementation. Some older tools predate the annotation API and simulate annotations using, for example, structured comments. Annotation-based approaches are very declarative in nature, and as such can be interpreted and used by a wide-range of tools for both static and runtime checks. For runtime checks, additional code that enforces the constraints must be generated and deployed. This is often done, for example, using injection-based techniques like AOP [72]. We now examine some popular approaches in more detail, loosely grouped by their major usage.

Bean Validation. The *Bean Validation* specification, JSR303 (version 1.0) [23] and JSR349 (version 1.1) [22], and a popular reference implementation, the hibernate validator [8], aim at providing a set of standard annotations and associated processing APIs for server-based enterprise (J2EE) applications. It offers an API to request validation which must be called explicitly by the programmer. The API is intended for use with higher-level frameworks that intercept program flow to check constraints. This is described in the documentation as follows: *"This service only deals with the actual validation of method parameters/return values itself, but not with the invocation of such a validation. It is expected that this invocation is triggered by an integration layer using AOP or similar method interception facilities such as the JDK's Proxy API or CDI. Such an integration layer would typically intercept each method call to be validated, validate the call's parameters, proceed with the method invocation and finally validate the invocation's return value."*[7].

Bean Validation represents post-conditions as constraints on method return values. An example is given in Listing 4. The standard states that *"As of version 1.1, Bean Validation constraints can also be applied to the parameters and return values of methods of arbitrary Java types. Thus the Bean Validation API can be used to describe and validate the contract (comprising pre- and postconditions) applying to a given method ("Programming by Contract", PbC)."* [22, sect. 1.2]. The Bean Validation standard also contains several restrictions to ensure correct behavioural subtyping according to the Liskov's Substitution Principle (LSP) [22, sect. 4.5.5].

[7] `https://docs.jboss.org/hibernate/validator/4.2/api/org/hibernate/validator/method/MethodValidator.html` (accessed 10 January 2017)

Static Checking. Various tools offer limited static analysis of annotations, such as for `null` analysis. The *Checker Framework* [83] provides annotations that can then be checked via compiler plugins. Many IDEs and static analysis tools provide similar capabilities for finding bugs at compile time, such as *Eclipse*, *IntelliJ* and *FindBugs*. This has lead to an unfortunate situation where annotations such as `@NonNull` and `@Nullable` with the same name exist in different name spaces. To rectify this, JSR305 aims to establish a set of standard annotations [90].

The *Java Modelling Language (JML)* is a mature framework that aims to bring full-fledged programming by contract to Java, and uses comment-based annotations to express constraints. The latest version of *OpenJML* also supports true annotations. JML supports both runtime checks and static verification [75] using additional tools like ESC/Java2 [42].

There are numerous other, somehow less popular approaches to annotation-based contracts, including *oval* [11], *CoFoJa* [4], *Jass/ModernJASS* [20], *lombok* [73], c4j [2], and the dormant *iContract* [51], *AssertMate* [5], *javadbc* [10] and *chex4j* [3] projects.

2.6 Other Approaches

While the above patterns cover the majority of cases, other means of expressing contracts exist in the Java world. *jContractor* [71] is unique in associating constraints with methods via naming conventions. For instance, the pre-conditions for a method named `push` are written by implementing a method `push_Precondition`. Constraints can also be written in separate *contract classes* which are again recognised by a certain naming convention. Behavioural subtyping is supported by aggregating inherited contracts (with *"or"* for pre-, and *"and"* for post-conditions). Contracts are weaved into code using bytecode instrumentation.

3 Methodology

In the following subsections we discuss how we obtained, processed and analysed the data when looking for the use of contracts in Java programs.

3.1 Data Sets

We initially considered several curated data sets, such as the Qualitas Corpus [97] and DaCapo [25]. However, we found DaCapo to be too small, outdated and without evolution data, and found that Qualitas does not contain the latest version of many programs and completely omits some widely used libraries (including Guava). Furthermore, for Qualitas, the projects do not have a canonical format making automated analysis difficult. Instead, we chose to extract a data set from the Maven Central Repository.

The Maven Central Repository is a simple directory-based repository of open-source projects. It contains a large number of Java programs in a canonical structure with meta-data that facilitates automated analysis. We used the ranking of projects by popularity from `https://mvnrepository.com/`, where popularity is determined by the number of incoming dependencies from other projects hosted on Maven. We extracted our data set as follows: first, we parsed the name, group and version of the first 200 artefacts from the MVN Repository website on the 3 August 2016; second, we used the search API [8] to download all available versions of the respective artefacts; finally, we removed projects for which Java source code

[8] http://search.maven.org/#api (accessed 10 January 2017)

Table 2 Data set metrics.

metric	value
programs	176
program versions	6,934
compilation units	2,233,298
unparsable compilation units	223
classes	2,787,686
methods (all)	22,263,421
constructors (all)	2,465,260
methods (public and protected)	18,744,459
constructors (public and protected)	2,002,327
KLOC incl comments	351,034

was not available (e.g. projects containing only Scala source code, or projects consisting only of meta data, etc). This resulted in 176 projects with 6,934 versions, and with an overall size of 4.6GB.

Metrics extracted from our data set are reported in Table 2. The number of compilation units corresponds to top-level classes but excludes inner classes. There were some compilation units where parsing failed, but they were relatively few (less than 0.01 %) and should not significantly impact our results. The number of classes and methods is significant here, since contracts are primarily applied to – and analysed for – these program elements. Note that only `public` and `protected` methods/constructors were considered in our studies. This is because `private` members do not play any role from a program's clients viewpoint and, from the perspective of evolution, cannot introduce breaking changes. Thus, to be consistent across our various experiments, we excluded them. Overall, the amount of code investigated is similar to the data set used in [53]; however, we use only *released versions* and not *revisions*, and have therefore significantly more variability in our data set.

Finally, we note that our data set does not include project dependencies. Since we also study contracts in inheritance hierarchies, we considered including the dependency closure of each project to ensure all supertype references could be resolved. We investigated this and found that this would have added another 14,832 versions from 972 programs, increasing the overall size by 5.9 GB. Unfortunately, this would have slowed down our experiments considerably. We therefore opted against including dependencies, but added the source for *openjdk 1.8.0_91*, assuming that a vast number of supertype relationships can be resolved against the Java core class libraries, and since the core libraries are known for their high level of stability (i.e. public APIs don't disappear between JDK versions).

3.2 Contract Element Classification

The studies reported on in this paper focus on the *usage*, *classification* and *evolution* of contract elements found. This requires a simple and mechanical means to classify the contract elements. In particular, we cannot attempt to determine the programmer's intention behind a particular programming construct (e.g. whether throwing an exception guarded by a conditional is checking a pre- or post-condition, etc). Fortunately, there are many signals that we can use to help classify concrete program code constructs as contract elements:

- **CREs.** As discussed previously, the names of runtime exceptions in many cases indicate they are designed for signalling contract violations (e.g. `IllegalArgumentException`),

and standard Java literature clearly indicates that the purpose of certain runtime exceptions is to enforce pre-conditions [26, item 58]. Our data analysis methods exploit this to classify the uses of (conditional) runtime exceptions accordingly.

- **APIs.** These provide a potentially richer language for expressing pre- and potentially also post-conditions, and class and method names and documentation usually signal their purpose. The purpose of all APIs we have encountered and investigated is to represent pre-conditions only.
- **Assertions.** In contrast to those above (and as already discussed in section 2.4) `assert` statements are not intended for checking pre-conditions. Therefore, we can only infer that some assertions might represent post-conditions or class invariants. We therefore decided to include assertions as potential contract elements in the study. But, we take a conservative approach and do not to classify them as this could have a significant impact on the precision of the study.
- **Annotations.** A special case are annotation-based approaches. Here the type of contract can often be inferred from how the annotation is used. For instance, consider again Listing 4. Here, two JSR303 annotations are used. The annotation on the method is actually a contract element on the method return value and is therefore a post-condition, while the `@Min` annotation is a contract element on the parameter and is therefore a pre-condition. Annotations on classes and fields are interpreted as class invariants.
- **Other.** *jContractor* contract elements can be easily classified based on the naming patterns used. However, we did not include this in the classification scheme used as we did not find any use of *jContractor* in the data set used in this study.

Figure 1 summarises the classification algorithm employed in this study. In classifying contract elements according to the above rules, we do not consider the relative position of a particular check within a method. That is, one might argue that a check near the entry of a method is "likely" to be a pre-condition check. However, our experience suggests that it is quite common to find legitimate pre-condition checks embedded deep within a method's body. Listing 3 illustrates such an example taken from a real-world codebase. The contract on line 11 should be classified as a pre-condition check, but we note it is not located near the method's entry. Indeed, if we just consider its relative position within the method, then it would look more like a post-condition check. One could further argue that this use actually denotes a class invariant as it checks the state of an object (rather than the parameters of the method). What is more, concepts like "at method entry" or "before method exit" are further complicated – for the purpose of source code analysis – by the presence of comments and the potential presence of injected code from cross-cutting concerns such as logging or security checks. Sometimes these concerns are present in source code, but often tools like AOP [72] are used to inject or "weave" additional code (into source or byte code) at method entry and/or exit.

For this study, we therefore decided to use a set of classification rules extracted from the definition of the respective construct language. For annotations in particular, we take into account the type of annotation as discussed above.

3.3 Methodology for Contract Usage Study

This study looks at and classifies the usage of the several types of contract elements across our dataset, providing also the base data for subsequent studies described below. The approach taken was to identify the contract element using source code analysis, which is able to check all annotations including those which might be removed by the compiler.

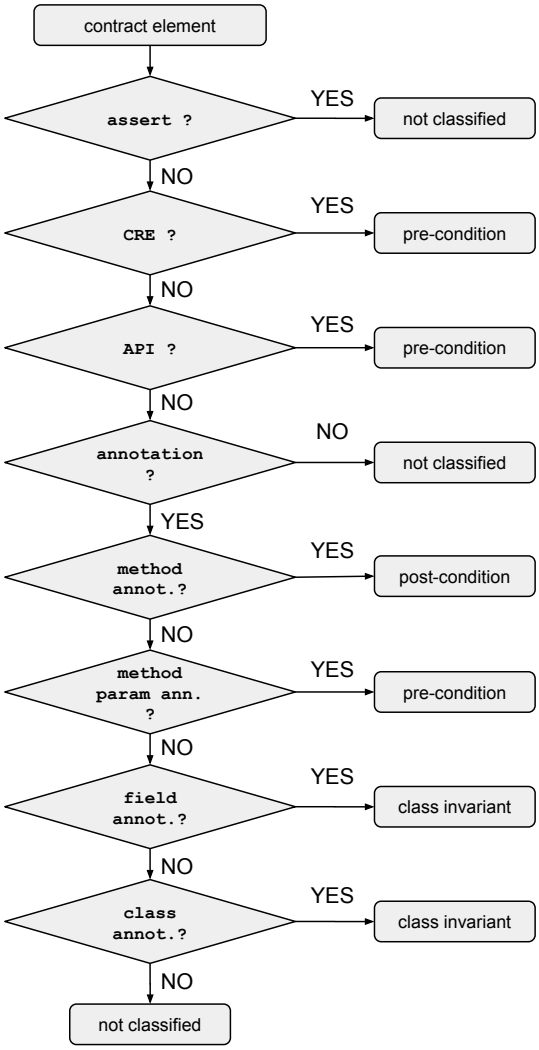

Figure 1 Contract element classification algorithm.

Furthermore, comment-based annotations used by some older tools (e.g. JML) also require source code-based analysis. Analysing conditional runtime exceptions and assertions in this manner was relatively straightforward. However, for the remaining contract patterns, we investigated their actual use in two stages.

In stage one, we developed screening scripts that looked for any sign that a certain pattern of contract construct might be present. These scripts use simple text matching algorithms and look mainly for the presence of type-specific package names (for APIs and annotations) or comment patterns (for comment-based annotations). These scripts revealed that only the following API and annotation-based contract types are present in programs in our data set: *Commons Validate*, *Guava preconditions*, *Spring asserts*, *Bean Validation (JSR303, JSR349)*, *JSR305*, *FindBugs*, *IntelliJ*, *Lombok*. We however removed *Lombok* because of its particular contract semantics: annotations are translated into conditional runtime exceptions at build time, and this would have lead to double-counting. Lombok is also only used by itself.

We later found that the preprocessing screening scripts produced false positives for *FindBugs* and *IntelliJ* tool-bound annotations. Specifically, the data set contains programs

defining *FindBugs* contract annotations but not actually using them, and *IntelliJ* annotations are only referenced in comments (using their fully qualified class names). The result of this was that we did not find any instances of the respective tool-bound contract patterns in actual use, and we do not report them explicitly in the results tables.

In stage two of the extraction, a collection script was used to extract contract construct data and export it to `JSON` files stored for further analysis. This script uses a set of pluggable extractors for each contract pattern, the extractors perform a detailed AST analysis using the Java parser API [9].

3.4 Methodology for Evolution Study

The evolution study asked how contracts change between adjacent program versions, from the viewpoint of what effects this can have on a program's existing clients. This required us to identify the adjacent versions which is easy for projects that use common versioning schemes (i.e. `<major,minor, micro>` plus an optional qualifier or build number). However, some projects do not follow this convention which makes adjacency detection rather difficult. For instance, we encountered cases with letters in the major version number and cases with alphanumeric qualifiers with unclear semantics. The script which detects adjacent versions therefore uses a set of rules to correctly order versions by qualifier status (such as *alpha*, *beta*, *release candidate*, *final*, etc) in addition to the numerical versioning scheme. However, this still left us with 138 program versions that did not fit; we therefore blacklisted those and excluded them from the evolution study as it was not clear how they fit into a linear program evolution.

The evolution study uses contract data extracted in the previous step, and builds *diff records* that contain contracts for the same artefact (method or class) in two adjacent program versions. These records are then classified using pluggable *diff rules* to detect contract evolution patterns such as non-critical changes (e.g. only the program messages are changed), the addition of post-conditions, etc. These diff rules are not intended to provide a completely precise classification – this might actually be impossible, but they can be used to automatically classify a vast number of simple cases.

Diff rules do not capture cases when contract elements specified informally (for instance, in comments) are formalised using any of the approaches described above, or vice versa. This study is about the correct use of contracts in the context of evolution, and correctness is defined with respect to actual program behaviour. While informal contracts specify intended program behaviour, they do not influence actual behaviour. Therefore, (de)formalising contract elements is a potentially critical operation.

We use the following set of diff rules, corresponding to the principles of substitutability:

1. `Unchanged` – the two contracts compared are the same.
2. `IgnoreOrderAndMessage` – the order of contract constructs attached to an artefact and the message (used as message in exception -, annotation- and API-type contracts) of some of these constructs has changed. Although scenarios can be constructed where this changes the semantics of a program, this change is probably benign.
3. `PreconditionsStrengthened` – a pre-condition has been added to a method, or a clause (boolean expression) has been added to an existing pre-condition using the `&` or `&&` operator. This is potentially critical.
4. `PreconditionsWeakened` – a pre-condition has been removed from a method, or a clause has been added to an existing pre-condition using the `|` or `||` operator. This is benign.
5. `PostconditionsWeakened` – a post-condition has been removed from a method, or a clause has been added to an existing post-condition condition using the `|` or `||` operator.

This is potentially critical.

6. `PostconditionsStrengthened` – a post-condition has been added to a method, or a clause has been added to an existing post-condition using the `&` or `&&` operator. This is benign.
7. `NullablePostconditionRemoved` – the removal of a `Nullable` post-condition annotation is not considered as a significant weakening of guarantees made, this is classified as a benign change.

The data reported later in sections 4.2 and 4.4 show that with these simple rules, a large percentage of contract changes can be automatically classified.

3.5 Methodology for LSP Study

In this study we look for violations of Liskov's Substitution Principle (LSP), i.e. we look for contradictory specifications between super- and sub-classes, and then use the respective contracts to analyse whether they are used correctly. The experimental setup uses the same infrastructure as the evolution study, the main difference is that the diff records are extracted differently. Here we look for contracts on methods in an override relationship and analyse imports and inner classes to compute precise inheritance information. We also filter the extracted diff records in order to remove those that duplicate the same issue in a different version of the same program.

For the internal representation of contract data, we encode methods similarly to descriptors used in byte code, but with return types removed. This allows us to capture overriding with covariant return types.

3.6 Verifiability

The data sets (raw data, and contract data extracted) are available here: `https://goo.gl/2R28gS`. The code used for extraction and analysis is available from `https://bitbucket.org/jensdietrich/contractstudy`. The repository `readme.md` contains detailed instructions how to build the project, add the data for analysis, and reproduce the results reported.

4 Results

4.1 Contract Usage (RQ1)

Table 3 reports the different types of contract elements and their appearance in the data set. In column 3, the overall number of contract elements of the respective type is reported. These numbers are high. However, one reason for this is that the same elements are counted again and again in different versions. We therefore also computed the number of contract elements found for the latest versions of each program within the data set, this is reported in column 4. These values are much lower. We also investigated the number of programs using constructs of the respective type in any version. These numbers are displayed in column 5. The adoption of the various API and annotation-based approaches is surprisingly low, and even the number of projects using assertions is lower than expected.

We also computed the gini coefficient [60] in order to measure the distribution of constructs amongst the latest versions of each program. The gini is very high at 0.74 indicating that while there are a few projects that use contracts very intensively, the vast majority of projects do not use them significantly. Interestingly, the gini computed for the distribution of assertions is 0.83 — much higher than the overall gini. On the other hand, the gini for

Table 3 Number of contract elements found in dataset by type.

type	category	constructs (all ver.)	constructs (latest v.)	programs
assert	assertion	131,340	3,284	52
conditional runtime exceptions	CRE	484,964	15,720	155
unsupported operation exception	CRE	123,966	3,084	122
guava preconditions	API	49,021	1,188	6
spring assert	API	100,232	2,148	13
commons validate	API	879	110	6
JSR303, JSR349	annotation	586	20	1
JSR305	annotation	33,281	911	6

Table 4 Top programs using contracts (latest versions only). The numbers in brackets are the numbers of contract elements found in the respective program.

category	programs
CRE	open-jdk (3,695), elasticsearch (1,348), lucene-core (612), netty (553), hadoop-common (550)
API	guava (948), spring (661), spring-test (262), spring-web (218), spring-core (208)
assertion	lucene-core (1,000), elasticsearch (656), open-jdk (390), gwt-user (371), gwt-servlet (371)
annotation	guava (859), reflections (46), hibernate-validator (20), annotations (4), jsr305 (2)

the distribution of APIs is lower (0.6), indicating a more equal distribution. This seems counter-intuitive as there are many more programs using assertions than contract APIs. But while more programs use assertions, most of them use very few — in several cases only one single assertion.

Table 4 shows the programs with the highest usage of contract elements of each type, with indication of their number in the respective latest version. This data also shows the uneven distribution of contract usage, as the numbers quickly trail off.

We also investigated popular combinations of contract types. We found that 16 programs do not use any contracts, 32 programs use one type of contract element (of which 28 use conditional runtime exceptions), 63 use two types of contracts (the most popular combination being unconditional "unsupported operation" exceptions and conditional runtime exceptions with 54 occurrences). There were 59 programs with three types, 4 with four types and finally 2 programs with five types of contracts (*elasticsearch* and *guava* both use assertions, conditional and unconditional exceptions, the Guava contract API and JSR305 annotations).

Finally, in Table 5 we report the classification of the contract elements found. As discussed earlier, a precise classification is not possible. But this data suggests that pre-conditions are more frequently used than post-conditions. This would still be the case if all assertions encountered (and reported as not classified in Table 5) were classified as post-conditions.

A possible explanation for the dominance of pre-conditions is the high level of reuse of (library) code in general, and of open source programs in particular. This implies that modern libraries have to provide defensive API surfaces to deal with unknown clients. As Meyer [80] notes: *"A pre-condition violation indicates a bug in the client (caller). The caller did not observe the conditions imposed on correct calls. A post-condition violation is a bug*

Table 5 Number of contract elements found in dataset by classification.

kind	constructs (all versions)	constructs (latest version)	programs
pre-condition	786,723	22,969	160
post-condition	2,413	112	6
class invariant	3,793	100	5
not classified	131,340	3,284	52

in the supplier (routine). The routine failed to deliver on its promises.". Therefore, by using pre-conditions, clients can shift the responsibility to comply to clients. This has many practical advantages with respect to program maintenance: if a program fails and an illegal argument or similar exceptions occur in stacktraces due to a failed pre-condition, this makes it very clear who is to blame, and reduces the workload on the side of the supplier as it does not have to deal with bug reports.

While this may explain the relative popularity of pre-conditions, it does not explain why post-conditions are not as widely used. One possible reason is the widespread use of unit testing. Since method callers are often unknown at build time, tests are written that create synthetic callers in test fixtures. The test assertions comparing computed values against test oracles are basically post-conditions specialised for a particular fixture. We note that tests written in modern testing frameworks like *junit* are following a contract-oriented pattern: "if the assumptions (pre-conditions) are true before the method under test is invoked then the assertions (post-conditions) must be true after the method under test has been invoked". But the focus is clearly on the post-condition check, and we believe that many developers are not aware that pre-condition checks are supported by *junit* in the form of `org.junit.Assume` or (less explicit) by *TestNG* through `org.testng.SkipException`.

4.2 Contract Evolution (RQ2)

We also tried to answer the question whether there is some evidence that programs use more contracts as they evolve, similarly to [53] but with a coarser granularity due to the extent of our study. To answer this, we divided the number of contracts found by the number of methods, and compared the respective ratio between the first and last version of each program. The aggregated result of this experiment is shown as a box plot in Figure 2. The median value of the ratio does not change much (from 0.021 to 0.023) between the first and the last version within the version ranges investigated. This indicates that if projects use contracts, they keep using them.

To see if this observation is of importance, we also considered the growth of the size of the respective programs. The average growth in the number of methods between the first and the last version is 174 % (although this is dominated by a few outliers — for instance, the first version of *spring-core* in the dataset (1.2.1) has 324 methods, while the last version (4.3.2.RELEASE) has 3,276 methods). But even when considering the median, the number of methods still increases significantly, by 68.5 %. This means that the overall number of contract elements used increases proportionally with the size of the programs.

There are only two programs in the data set for which the number of contract elements used declines significantly between the first and the last version investigated, both in relative and absolute terms: *httpclient* (from 1,154 methods with 298 contract elements in version 4.0-beta1 to 2,772 methods with 26 contract elements in version 4.5.2) and the related *httpcore* (from 923 methods with 252 contract elements in version 4.0-beta2

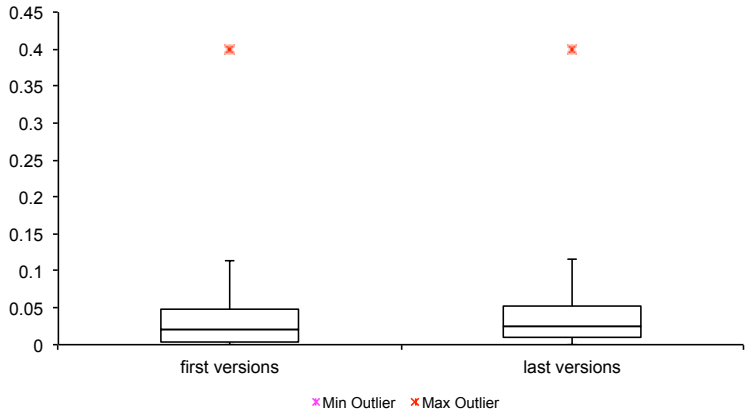

Figure 2 Comparison of contract-to-method ratios between first and last versions

to 1,584 methods with 46 contract elements in version 4.4.5). A more detailed analysis shows that both projects adopted a shared project-specific contract API very similar to Guava's `Preconditions`, the respective API is `org.apache.http.util.Args`, introduced in *httpcore 4.3-alpha1*. When taking this into account, the situation changes: *httpclient-4.5* has 584 call sites for methods defined in `Args` corresponding to API-type contracts , while this class is not used at all in 4.0-beta1. While `Args` provides a centralised API for input validation that is used to replace `IllegalArgumentExeptions`, there are also two documented cases where `IllegalStateExceptions` are replaced by a project-specific checked exception (`ConnectionClosedException`) that are not captured in our analysis[9].

We also looked into contract evolution in programs that made heavy use of contracts in their respective first version. For this purpose, we filtered out programs with at least 100 contract elements in their first version. This is the case for 27 programs. Of these programs, only three show significant changes in contract usage, which we defined by a change of the contract element-to-method count ratio larger than 0.1. Two of those programs are *httpclient* and *httpcore*, already discussed above, the third program is *lucene-core* that shows a significant *increase* of contract usage between versions 2.3.0 (3,096 methods with 158 contract elements) and 6.1.0 (8,954 methods with 1,612 contract elements), respectively.

From this we conclude that projects that use contracts continue to do so, and expand the use of contracts as they grow and evolve, presumably because contracts are seen as beneficial.

4.3 Contract Safety and Program Evolution (RQ3)

Next, we looked for *evolution patterns*. In particular, we were interested in cases where contract evolution was unsafe in terms of substitutability. This means, if there are cases where a client using an API with a contract could break after an upgrade because an API method had either strengthened its pre-conditions, or weakened its post-conditions. Table 6 gives an overview of the results. As discussed in section 3.4, our classification is not complete as it is not feasible to precisely capture the notion of strengthening and weakening constraints if the respective constructs can be written in a full-fledged programming language. But we did extract some interesting results, and discuss some examples in more detail.

[9] `https://archive.apache.org/dist/httpcomponents/httpcore/RELEASE_NOTES.txt` (accessed 10 January 2017)

Table 6 Contract evolution data result summary.

evolution	critical	count
unchanged	no	652,395
minor change	no	1,512
pre-conditions weakened	no	12,675
post-conditions strengthened	no	18
pre-conditions strengthened	yes	2,777
post-conditions weakened	yes	7
unclassified	?	5,028

Table 7 Contract hierarchy data result summary

evolution	critical	count
unchanged	no	351
minor change	no	193
pre-conditions weakened	no	40
post-conditions strengthened	no	0
pre-conditions strengthened	yes	1,242
post-conditions weakened	yes	0
unclassified	?	556

In *slf4j-api* (logging library) the class `org.slf4j.LoggerFactory` has the method `getILoggerFactory()`. A JSR305 `Nonnull` post-condition annotation is present in this method in version 1.7.8 but removed in version 1.7.9. This breaks the guarantees made to clients using this class. Note that the change happens during a *micro* version change, which is supposed to maintain API compatibility according to the rules of semantic versioning.

In *commons-cli* (CLI library), the method `addValue(String)` in `org.apache.commons.cli.Option` has a (rather complex) implementation in version 1.0, but support for this method was then removed in version 1.1 by throwing an `UnsupportedOperationException` with the message *"The addValue method is not intended for client use. ..'*.

4.4 LSP Study (RQ4)

We analysed our data set for cases where contracts gave an indication of potential violations of Liskov's Substitution Principle (LSP), and found numerous such cases summarised in Table 7. Closer inspection of the data also revealed certain programming patterns where runtime exceptions were not being used to communicate violated contracts, but to return information to the applications. A good example for this are certain adapters in ASM 5.0 that perform various checks on byte code, such as `org.objectweb.asm.util.CheckSignatureAdapter`. To do so, these adapters have to override `visit` methods in the adapter supertypes. The rules to check for are implemented using the unconditional runtime exception pattern. I.e., a runtime exception is thrown if the check fails. Here runtime exceptions are used with a semantics similar to return values.

We provide some examples of programs that violate the rules of behavioural subtyping according to how they use contracts.

In *hibernate-core-3.5.0*, the class `org.hibernate.dialect.Dialect` is subclassed by `IngresDialect` in the same package. `Dialect` implements the method `getLimitString(String,int,int)`, the implementation does not throw a runtime exception. It is overridden in

`IngresDialect`, and an `UnsupportedOperationException` is thrown if the second argument (`offset`) is negative. This is a case of unsafe substitution, where a pre-condition on a parameter is added in a subclass. There is no indication in the documentation of the method in `Dialect` warning developers that a runtime exception might be thrown in overriding methods.

In *spring-webmvc-3.2.11.RELEASE*, in the package `org.springframework.web.servlet.tags.form`, `FormTag` extends `AbstractHtmlElementTag` and overrides `setCssErrorClass(String)`. While the implementation in the superclass is a plain setter, the overridden method throws an unsupported operation exception with the message *"The 'cssErrorClass' attribute is not supported for forms"*. There is again no indication in the super class that some subclasses might not support this method.

5 Limitations and Threats to Validity

5.1 Data Set

The data set used only consists of open source programs. This is a consequence of (1) our methodology that requires source code to be analysed and (2) the simple fact that we did not have access to real-world commercial code. It is therefore not clear whether our results apply to closed-source commercial programs. We also suspect the data set is biased towards libraries as they are re-used by many other programs (hence have a higher ranking on Maven Central). In particular, this could cause an under-reporting of annotation-based contracts which might be more common in J2EE applications using frameworks providing those annotations.

5.2 Contract Extraction

While we carefully studied academic as well as grey literature for references to tools and APIs used to represent contracts in Java, there is no guarantee that our list is complete. The bigger limitation however is that we did not capture project-specific techniques such as custom annotations or APIs. We discovered one such case in *httpclient*, discussed in more detail in Section 4.2.

Our extraction of pattern-based contracts could lead to under-reporting. First, we could have considered other runtime exception classes. The ones we used were chosen after inspecting the documentation and assessing their suitability of expressing pre-conditions. Our choice is consistent with the various contract APIs which use exactly those classes to offer API-based pre-condition checking. But there could still be (project-specific) classes we missed. Furthermore, we might have missed certain patterns for how these exceptions are used. We can at least approximate the worst case scenario for this by counting all instantiation sites for the respective exception classes. We found 841,815 instantiation sites across all versions. This compares to 624,269 contracts found (combined conditional and unconditional exceptions), i.e. we have a precision of *at least* 74 % for this type of contract construct.

One of the annotation-based APIs we investigated has a proprietary mechanism for "contract inheritance". JSR305 [90] defines the annotation `javax.annotation.ParametersAreNullableByDefault` with the following semantics: if a class is annotated, then all method parameters in all methods of this class are *nullable* by default. There are only two annotations we are aware of that have this semantics, and we did not model this in this study. This therefore leads to an under-approximation of the contracts found in programs.

We already discussed the special case of assertions (see Section 2.4). This leads to some over-approximation in the overall number of contract elements extracted as not all assertions represent contract elements, and to an under-approximation of post-conditions extracted as we do not classify assertions. This is discussed in the result section in detail.

5.3 Evolution

There are two limitations here. Firstly, our analysis under-approximates inheritance-related problems as we miss some inheritance relationships due to the fact that we did not investigate the full dependency closure of our data set, as discussed in section 3.5.

Secondly, our analysis is completely mechanical and detects possible problems, but does not attempt to weigh them and to assess their actual impact. For instance, many issues detected in the *PreconditionsStrengthened* category flag potential problems that may have an impact on clients. But many of these changes are cases where contracts are introduced to methods. This might just be a case of making existing *"closet contracts"* [16] more explicit. In many cases this will change the way in which a contract violation is reported, i.e. the type of runtime exception that is being thrown. This is of course a semantic change that could break existing clients, but it is unlikely that it actually does. A similar case is when a project decides to change its approach to contracts completely, for instance by replacing contract API calls by runtime exceptions or vice versa. We are aware of one such case, discussed is section 2.3. Secondly, evolution issues impact on clients with separate lifecycles. This means that even incompatible changes of public methods may not be critical if they are not part of the public API of a given program. This is partially caused by the properties of the Java programming language that offers no easy way to enforce *program-private* access to APIs[10].

5.4 LSP Study

The analysis shares some issues around the validity of potential problems discussed with the evolution study. A specific issue are LSP violations with annotation-based contracts. These contracts are usually deployed using injection-based techniques, and at this stage the respective contract framework can take care of merging the constraints of methods with the constraints of overridden methods in order to satisfy LSP. We found however that only 2.76 % of the diff records extracted and investigated use annotations.

With the more explicit, code-based approaches like APIs and explicit runtime exceptions, this kind of *contract merging* would require the use of `super`. We excluded all methods using `super` references from the analysis for this reason, but this again produces a conservative under-approximation of potential LSP issues. However, only 2.77 % of diff records refer to methods overriding with `super`.

6 Related Work

We now review a cross-section of related work, paying particular attention to empirical work and contract languages.

[10] Although this can be achieved via classloader-based add-on technologies, such as OSGi.

6.1 Empirical Studies

Casalnuovo *et al.* undertook an empirical study of the 100 most popular C/C++ projects on GitHub [32]. Their primary interest was the connection between assertions and defect occurrences and their main finding was that the presence of assertions in a method had a small (but significant) effect on reducing defects within it. In taking these measurements they correctly identified — and controlled for — a number of well-known confounds, such as method size and number of contributors. However, they later identified a flaw in their experimental setup related to the reliance on `git` for identifying the enclosing method of a commit [31]. Having fixed and repeated this part of the study, they subsequently found the *opposite* result — namely, that the presence of assertions in a method had a small (but significant) effect on *increasing* the number of defects. Indeed, this is perhaps more intuitive as one expects the presence of assertions to increase the *observability* of faults [99, 39]. That said, the authors conservatively concluded *"that there is no evidence that non-test asserts have an effect on defects"*. Of most relevance here was their finding that 69 out of 100 projects contained *"more than a minimal presence"* of `assert` statements. This contrasts with our observations that only 52 out of 176 projects used them, and one explanation for this maybe their focus on C/C++ projects compared with our focus on Java projects. For example, Java developers may eschew `assert` statements in favour of conditional runtime exceptions which are always enabled (Table 3 supports this to some extent). Finally, another interesting finding of Casalnuovo *et al.* was that *"methods with asserts are more likely to take on the role of hubs"* (roughly speaking, methods which call many other methods). Whilst our results do not provide any specific insight into this, it would certainly be interesting to see whether contracts are similarly correlated (as one might expect them to be).

Another relevant work is that of Estler *et al.* who examined contract usage in practice [53]. Their empirical study looked at a suite of projects written in Eiffel, C# and Java across 7700 revisions and totalling 260MLOC. For the C# and Java projects, the contract languages employed were (respectively) Microsoft Code Contracts [14] and JML [75]. Contrasting with our work, they only considered projects which actually used contracts in a meaningful way (roughly speaking, around 5% of methods had to have some kind of specification to be included). As such, the occurrence of contracts was much higher and, on average, the proportion of methods with contracts was around 40%. Regarding usage patterns, they found no strong preference for the kind of contract used (i.e. pre-/post-condition, class invariant, etc). However, they did find that preconditions, when used, tended to be larger. This contrasts with our observations that preconditions were, by far, the more frequent (recall Table 5). This difference may be explained in two ways: firstly, the contract constructs we analysed tend to favour preconditions (recall Figure 1); secondly, there was a considerable difference in the nature of projects considered, as Estler *et al.* specifically selected projects with significant contract usage. Indeed, they comment that *"In the majority of projects in our study, developers devoted a considerable part of their programming effort to writing specifications"*. Another relevant aspect of their work was an attempt to examine how contracts evolve over time and, consistent with our findings, concluded that *"the fraction of routines and classes with some specification is quite stable over time"*. They also considered a concept of "strength" similar to ours by counting the number of clauses in a contract. Again, they observed that the average strength of a contract was relatively stable over time. Finally, they also compared implementation code against contracts and, as perhaps expected, found that a method's implementation changes much more frequently that its contract. These latter findings complemented their earlier work where they identified a general trend for contracts [54]. Specifically, that they tend to change frequently in the early phases of a project, before stabilising.

Schiller *et al.* examined the use of Code Contracts across a corpus of 90 C# programs listed on Ohloh comprising around 3.5MLOC, with the goal of providing guidance for the design of contract languages [94]. Their approach was multi-pronged. Of particular relevance here is their use of an automatic analysis to categorise contract properties (i.e. clauses). Their focus was on whether contracts were checking common or simple properties (e.g. `null` checks) or richer application-specific properties and, unfortunately, found that by far the majority of contracts (around 73%) focused on `null` checks. Their conclusion was that writing nullness contracts may be consuming developer's limited time and "crowding out" other (more interesting) application-specific contracts. Their results also provide another data point on the question of pre-conditions versus post-conditions. Specifically, consistent with our findings, they observed a clear bias towards developers writing pre-conditions over post-conditions (68% vs 26%, with the rest being class invariants). Schiller *et al.* also employed a dynamic invariant synthesis tool (Daikon [52]) to infer contracts and then compared them with what the programmer wrote. They found that the tool inferred more post-conditions than pre-conditions, and concluded that *"the strong developer bias towards preconditions ... cannot be attributed to an absence of potential postconditions"*.

An earlier study on the use of contracts was conducted by Chalin [34]. His corpus consisted of 85 Eiffel projects totalling 7.9MLOC, including many freely available and open-source projects as well as a large number of proprietary projects. The study counted the lines of code used for contract elements, and categorised them according to use (e.g. pre-/post-condition, class/loop invariant, inline assertion, etc). Categorisation was simpler than in our case, since Eiffel provides explicit keywords signalling usage (e.g. `requires` for pre-condition, `ensures` for post-condition, etc). The experiment found that, roughly speaking, around 5% of measured lines were for contract elements. Of these, slightly more pre-conditions (50%) were observed than post-conditions (40%), with relatively few class invariants (7.1%). Compared with our findings and that of Schiller *et al.*, this shows a larger proportion of post-conditions and is more consistent with the findings of Estler *et al.* Chalin also found that only 35% of contract elements were `null` checks and, perhaps more surprisingly, that only 3% were for inline assertions. The latter suggests programmers find writing contracts more beneficial than checking internal invariants, perhaps because they aid interaction with others (e.g. via APIs).

Arnout and Meyer investigated the implicit contracts found in languages without linguistic support for them [16]. Their basic assumption was that, despite language limitations, programmers will still encode contracts using whatever means they have available and they refer to this as the *Closet Contract Conjecture*. This includes using exceptions to check pre- and post-conditions, but also includes mechanisms (i.e. encapsulation) for maintaining invariants over state. Their approach was to manually investigate a small number of classes from the .NET standard library (`ArrayList`, `Stack`, `Queue` and some related interfaces). Most importantly, from the perspective of this paper, they found strong evidence that exceptions were used to enforce contracts.

The work of Shrestha and Rutherford provides useful insight into the benefits of contracts with runtime assertion checking [95]. For a small set of Java classes, they measured the effectiveness of JML contracts in finding faults injected using mutation analysis, and observed a significant improvement over the null oracle (i.e. the underlying runtime system).

6.2 Contract Languages

There have been numerous attempts to add contracts to existing languages, such as Java, C# and C. Early examples include that of App [92] and Turing [67], and we now examine the more widely-used systems in detail.

Eiffel is perhaps the most influential and widely used language to support contracts [79]. Through this, Meyer promoted the idea of "Design by Contract" as a lightweight alternative to formal specification [80]. Numerous studies (some discussed above) have explored the use of contracts in Eiffel. For example, to automatically repair programs [86], to investigate strong specifications [88], to model programs in other programming languages [16] and much more.

The **Java Modelling Language** (JML) was an attempt to extend the Java language with a standard notation for expressing contracts [35, 36, 75]. The intention was that contracts in JML could be statically verified using ESC/Java [57]. Although ESC/Java was demonstrated on several real-world examples (e.g. for checking specifications for an electronic purse implementation [33]), the tool suffered many problems in practice. To help, JML also supported runtime assertion checking [38, 30, 75, 36]. Finally, work on JML continues through the OpenJML initiative [41, 93, 40]

The **Spec#** system followed ESC/Java, included a number of linguistic improvements over JML, and employed the Z3 automated theorem prover (as opposed to Simplify) [44]. Both of these meant it is capable of verifying a much wider range of programs than ESC/Java. Whilst the Spec# project has wrapped up, the authors did provide some reflections on their experiences [18]. Of particular relevance here is the following comment: *"There is a spectrum of possibilities for checking Spec# contracts. One extreme would be to verify all of them statically, another extreme would be to check them all dynamically. Either is impractically expensive."*. Here, they argued that the "runtime overhead is prohibitive" when using runtime checking.

6.3 Contracts in Component Composition and Evolution

In the context of component-based software engineering, contracts are used with a more general meaning and also include aspects such as API compatibility, quality of service attributes and more [24]. Several component frameworks have been proposed which use such contractual specifications, including Fractal [29], SOFA [87] and Treaty [47]. Dietrich and Stewart [49] looked into extracting formal contracts from Eclipse extension point documentations and found that by formalising them, they could find violations of social coding etiquette (Eclipse house rules [59]). Empirical studies on component and library evolution indicate significant potential for contract-breaking changes leading to compatibility problems [91, 48, 21, 49].

7 Conclusion

We have studied the use of contracts in a large set of widely-used, real-world Java programs. Although we found contracts being used, there is no evidence of their widespread application. If the *Closet Contract Conjecture* of Arnout and Meyer holds, then the contracts referred to are hidden deeper, where we couldn't find them. We also found no evidence that the adoption of contracts is increasing. However, when projects do use contracts, they continue to do so and expand the use of contracts as they grow and evolve, presumably because contracts are seen as beneficial. We did also find cases of incorrect use, that is, where the use of contracts does not guarantee safe substitution, neither in the context of evolution nor in the context of inheritance.

We do not have any ultimate answer as to the reasons for these findings, and can merely offer some possible explanations. Aspects that we think are of importance here include: the fragmentation of technologies and the lack of standardisation; the actual and perceived

performance overhead of enforcing contracts; the lack of tooling; and, the widespread use of testing that has a similar purpose. In summary, all of this impacts on the (actual and perceived) return on investment from using contracts. A more detailed study to explore the reasons behind our findings is an interesting and important area of future research.

Finally, an interesting question is how our results can be actioned. For **researchers** the novel insights gained into the kinds of contracts used in practice facilitates the development of new tooling. We have demonstrated, for example, that it is not difficult to extract rich semantic information regarding contracts from real-world programs. Likewise, this paper and the associated artifact (data and scripts) help to address the limited data available on real-world systems for comparison purposes. For **practitioners** the study has revealed a significant amount of technology fragmentation that hampers progress. As such, practitioners would ideally work towards better standards (e.g. for contract APIs / annotations), and create tools for processing contracts (as illustrated in this paper).

Acknowledgements. We would like the thank the anonymous readers for their helpful comments on this paper.

References

1 Apache commons lang. Accessed 12 August 2016. URL: `https://commons.apache.org`.
2 C4J - DBC for Java. Accessed 12 August 2016. URL: `https://sourceforge.net/projects/c4j/`.
3 chex4j. Accessed 12 August 2016. URL: `https://sourceforge.net/projects/chex4j/`.
4 cofoja. Accessed 12 August 2016. URL: `https://github.com/nhatminhle/cofoja`.
5 Design by contract. Accessed 12 August 2016. URL: `http://c2.com/cgi/wiki?DesignByContract`.
6 Google core libraries for java 6+. Accessed 12 August 2016. URL: `https://github.com/google/guava`.
7 Hamcrest. Accessed 12 August 2016. URL: `http://hamcrest.org/`.
8 Hibernate validator. Accessed 12 August 2016. URL: `http://hibernate.org/validator/`.
9 Java parser. Accessed 12 August 2016. URL: `http://javaparser.org/`.
10 javadbc. Accessed 12 August 2016. URL: `https://www.openhub.net/p/javadbc`.
11 Oval - object validation framework for java. Accessed 12 August 2016. URL: `http://oval.sourceforge.net/`.
12 Spring framework. Accessed 12 August 2016. URL: `http://spring.io/`.
13 Valid4j. Accessed 12 August 2016. URL: `http://www.valid4j.org/`.
14 Code contracts, 2008. URL: `https://www.microsoft.com/en-us/research/project/code-contracts/`.
15 Wladimir Araujo, Lionel C. Briand, and Yvan Labiche. On the effectiveness of contracts as test oracles in the detection and diagnosis of functional faults in concurrent object-oriented software. *IEEE Transactions on Software Engineering*, 40(10):971–992, 2014.
16 Karine Arnout and Bertrand Meyer. Finding implicit contracts in.NET components. In *Proceedings of the Formal Methods for Components and Objects (FMCO)*, volume 2852 of *LNCS*, pages 285–318. Springer-Verlag, 2002.
17 J. Barnes. *High Integrity Ada: The SPARK Approach*. Addison Wesley Longman, Inc., 1997.
18 M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Communications of the ACM*, 54(6):81–91, 2011.

19 Mike Barnett, Robert De ine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

20 Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass—Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, 2001.

21 Jaroslav Bauml and Premek Brada. Automated Versioning in OSGi: a Mechanism for Component Software Consistency Guarantee. In *Proceedings of the Conference on Software Engineering and Advanced Applications (SEAA)*, pages 428–435, 2009.

22 Emmanual Bernard. Jsr 349: Bean validation 1.1, 2013. URL: `http://beanvalidation.org/1.1/`.

23 Emmanuel Bernard and Steve Peterson. Jsr 303: Bean validation, 2009. URL: `http://beanvalidation.org/1.0/`.

24 Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.

25 Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, number 10, pages 169–190. ACM, 2006.

26 Joshua Bloch. *Effective Java*. Pearson Education, 2008.

27 J. Bowen and M. Hinchey. Ten commandments of Formal Methods ... ten years later. *IEEE Computer*, 39(1):40–48, 2006.

28 Carl Brandon and Peter Chapin. A SPARK/Ada CubeSat control program. In *Proceedings of the Conference on Reliable Software Technologies (RST)*, pages 51–64, 2013.

29 Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean Stefani. The fractal component model and its support in Java. *Software: Practice and Experience*, 36:1257–1284, 2006.

30 Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Electronic Notes in Computer Science*, 80:75–91, 2003.

31 Casey Casalnuovo, Premkumar T. Devanbu, Vladimir Filkov, and Baishakhi Ray. Replication of assert use in github projects. Technical report, 2015.

32 Casey Casalnuovo, Premkumar T. Devanbu, Abilio Oliveira, Vladimir Filkov, and Baishakhi Ray. Assert use in github projects. In *Proceedings of the International Conference of Software Engineering (ICSE)*, pages 755–766. IEEE Computer Society Press, 2015.

33 Néstor Cataño and Marieke Huisman. Formal specification and static checking of Gemplus' electronic purse using ESC/Java. In *Proceedings of the Symposium on Formal Methods Europe (FME)*, volume 2391 of *LNCS*, pages 272–289. Springer-Verlag, 2002.

34 Patrice Chalin. Are practitioners writing contracts? In *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*, pages 100–113, 2006.

35 Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Symposium on Formal Methods for Components and Objects (FMCO)*, pages 342–363, 2005.

36 Patrice Chalin and Frédéric Rioux. JML runtime assertion checking: Improved error reporting and efficiency using strong validity. In *Proceedings of the Symposium on Formal Methods (FM)*, volume 5014 of *LNCS*, pages 246–261. Springer-Verlag, 2008.

37 Roderick Chapman and Florian Schanda. Are we there yet? 20 years of industrial theorem proving with SPARK. In *Proceedings of the Conference on Interactive Theorem Proving (ITP)*, pages 17–26, 2014.

38 Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*, pages 231–255, 2002.

39 L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM Software Engineering Notes*, 31(3):25–37, 2006.

40 David R. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In *Proceedings of the NASA Formal Methods Symposium (NFM)*, volume 6617 of *LNCS*, pages 472–479. Springer-Verlag, 2011.

41 David R. Cok. OpenJML: Software verification for Java 7 using JML, OpenJDK, and eclipse. In *Proceedings of the Workshop on Formal Integrated Development Environment (F-IDE)*, volume 149, pages 79–92, 2014.

42 David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Proceedings of the Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2005.

43 P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A Software Analysis Perspective. In *Proceedings of the Conference on Software Engineering and Formal Methods (SEFM)*, pages 233–247. 2012.

44 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

45 David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 1998.

46 L. Peter Deutsch. *An interactive program verifier*. Ph.D., 1973.

47 Jens Dietrich and Graham Jenson. Components, contracts and vocabularies-making dynamic component assemblies more predictable. *Journal of Object Technology*, 8(7):131–148, 2009.

48 Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *Proceedings of the Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 64–73. IEEE Computer Society Press, 2014.

49 Jens Dietrich and Lucia Stewart. Component contracts in Eclipse - A case study. In *Proceedings of the Symposium on Component-Based Software Engineering (CBSE)*, volume 6092, pages 150–165, 2010.

50 C. Dross, P. Efstathopoulos, D. Lesens, D. Mentre, and Y. Moy. Rail, space, security: Three case studies for SPARK 2014. In *Proceedings of the Embedded Real Time Software And Systems (ERTS)*, 2014.

51 Oliver Enseling. icontract: Design by contract in Java. Accessed 12 August 2016. URL: `http://www.javaworld.com/article/2074956/learn-java/icontract--design-by-contract-in-java.html`.

52 Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

53 H.-Christian Estler, Carlo A. Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer. Contracts in practice. In *Proceedings of the Symposium on Formal Methods (FM)*, volume 8442 of *LNCS*, pages 230–246. Springer-Verlag, 2014.

54 H.-Christian Estler, Marco Piccioni, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. How specifications change and why you should care. *Computing Research Repository (CoRR)*, abs/1211.4775, 2012.

55 Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 2103–2110. ACM, 2010.

56 J. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In *Proceedings of the European Symposium on Programming (ESOP)*, pages 125–128, 2013.

57 C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

58 R. W. Floyd. Assigning meaning to programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–31. American Mathematical Society, 1967.

59 Erich Gamma and Kent Beck. *Contributing to Eclipse: principles, patterns, and plug-ins*. Addison-Wesley Professional, 2004.

60 Olga Goloshchapova and Markus Lumpe. On the application of inequality indices in comparative software analysis. In *Proceedings of the Australasian Software Engineering Conference (ASWEC)*, pages 117–126. IEEE, 2013.

61 D. I. Good. Mechanical proofs about computer programs. In *Mathematical logic and programming languages*, pages 55–75, 1985.

62 Alwyn E. Goodloe, César Muñoz, Florent Kirchner, and Loïc Correnson. Verification of numerical programs: From real numbers to floating point numbers. In *Proceedings of the NASA Formal Methods Symposium (NFM)*, pages 441–446, 2013.

63 C. A. R. Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, 2003.

64 C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

65 Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

66 Ashlie B. Hocking, John C. Knight, M. Anthony Aiello, and Shinichi Shiraishi. Arguing software compliance with ISO 26262. In *Proceedings of the Symposium on Software Reliability Engineering (ISSRE)*, pages 226–231. IEEE Computer Society, 2014.

67 Richard C. Holt, Philip A. Matthews, J. Alan Rosselet, and James R. Cordy. *The Turing Programming Language. Design and Definition*. Prentice Hall, 1988.

68 B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of the NASA Formal Methods Symposium (NFM)*, pages 41–55, 2011.

69 B. Jacobs, J. Smans, and F. Piessens. A quick tour of the verifast program verifier. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, pages 304–311, 2010.

70 T. J. Jennings and B. A. Carré. A subset of Ada for formal verification (SPARK). *Ada User*, 9(Supplement):121–126, 1989.

71 Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: A reflective Java library to support design by contract. In *Proc. REFLECTION*, pages 175–196, 1999.

72 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

73 Michael Kimberlin. Reducing boilerplate code with project lombok. URL: `http://jnb.ociweb.com/jnb/jnbJan2010.html`.

74 S. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, 1969.

75 G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, March 2005.

76 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6355 of *LNCS*, pages 348–370. Springer-Verlag, 2010.

77 K. Rustan M. Leino. Developing verified programs with Dafny. In *Proceedings of the Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 7152 of *LNCS*, pages 82–82. Springer-Verlag, 2012.

78 D. Luckham, SM German, F. von Henke, R. Karp, P. Milne, D. Oppen, W. Polak, and W. Scherlis. Stanford Pascal Verifier user manual. Technical Report CS-TR-79-731, Stanford University, Department of Computer Science, 1979.

79 B. Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

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

81 Emerson Murphy-Hill and Dan Grossman. How programming languages will co-evolve with software engineering: a bright decade ahead. In *Proceedings of the on Future of Software Engineering (FOSE)*. ACM, 2014.

82 Peter Naur. Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6, 1966.

83 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 201–212, 2008.

84 D. J. Pearce and L. Groves. Whiley: a platform for research in software verification. In *Proceedings of the Conference on Software Language Engineering (SLE)*, pages 238–248, 2013.

85 D. J. Pearce and L. Groves. Designing a verifying compiler: Lessons learned from developing whiley. *Science of Computer Programming*, 113(2):191–220, 2015.

86 Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering*, 40(5):427–449, 2014.

87 Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE transactions on Software Engineering*, 28(11):1056–1076, 2002.

88 N. Polikarpova, C. Furia, Y. Pei, Y. Wei, and B. Meyer. What good are strong specifications? In *Proceedings of the International Conference of Software Engineering (ICSE)*, pages 262–271, 2013.

89 Tom Preston-Werner. Semantic versioning 2.0.0. Accessed 12 August 2016. URL: `http://semver.org/`.

90 William Pugh. Jsr305: Annotations for software defect detection, 2013. URL: `https://jcp.org/en/jsr/detail?id=305`.

91 S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning versus breaking changes: A study of the Maven repository. In *Proceedings of the Working Conference on Source Code Analysis & Manipulation*, pages 215–224. IEEE Computer Society Press, 2014.

92 David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

93 José Sánchez and Gary T. Leavens. Static verification of PtolemyRely programs using OpenJML. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, pages 13–18. ACM Press, 2014.

94 Todd W. Schiller, Kellen Donohue, Forrest Coward, and Michael D. Ernst. Case studies and tools for contract specifications. In *Proceedings of the International Conference of Software Engineering (ICSE)*, pages 596–607, 2014.

95 Kavir Shrestha and Matthew J. Rutherford. An empirical evaluation of assertions as oracles. In *ICST*, pages 110–119. IEEE Computer Society Press, 2011.

96 Clemens Szyperski. *Component Software, Second Edition.* ACM Press, Addison-Wesley, 2002.

97 Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Asia Pacific Software Engineering Conference (APSEC)*, pages 336–345. IEEE, 2010.

98 The OSGi Alliance. OSGi service platform, 2012. Release 4.3.

99 Jeffrey M. Voas and Keith W. Miller. Putting assertions in their place. In *Proceedings of the Symposium on Software Reliability Engineering (ISSRE)*, pages 152–157, 1994.