Preprint of:
P. Chalin and P. R. James, "Non-null References by Default in Java: Alleviating the Nullity Annotation Burden". Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP) , Berlin , Germany , pp. 227-247, 2007.

Non-Null References by Default in Java: Alleviating the Nullity Annotation Burden

Patrice Chalin, Perry James

Dependable Software Research Group,
Dept. of Computer Science and Software Engineering,
Concordia University Montréal, Québec, Canada
chalin@cse.concordia.ca, perry@dsrg.org

Abstract. With the advent of Java 5 annotations, we note a marked increase in the availability of tools that can statically detect potential null dereferences. For such tools to be truly effective, they require that developers annotate declarations in their code with nullity modifiers and have annotated API libraries. Unfortunately, it has been our experience in specifying moderately large code bases that the use of non-null annotations is more labor intensive than it should be. Motivated by this experience, we conducted an empirical study of 5 open source projects (including the Eclipse 3.3 JDT Core) totaling over 700 KLOC. The results allow us to confirm that in Java programs, at least 2/3 of declarations of reference types are meant to be non-null, by design. Guided by these results, we propose a new non-null-by-default semantics. This new default has the advantages of better matching general practice, lightening the annotation burden of developers and being safer. We describe how we have adapted the Eclipse JDT Core to support the new semantics, including the ability to read the extensive, fully annotated API library specifications written by the Java Modeling Language (JML) community. Issues of backwards compatibility are also addressed.

1 Introduction

Null pointer exceptions (NPEs) are among the most common faults raised by components written in object-oriented languages. As a present-day illustration of this, we note that of the bug fixes applied to the Eclipse Java Development Tools (JDT) Core[1] between releases 3.2 and 3.3, five percent were directly attributed to NPEs. Developers increasingly have at their disposal tools that can detect possible null dereferences by means of static analysis (SA) of component source. A survey of such tools shows that the introduction in Java 5 of Annotations [29, §9.7] seems to have contributed to an increase in support for non-null static checking in Java.

It is well know that SA tools supporting *inter*-procedural analysis tend to yield a high proportion of false positives unless code and support libraries are supplemented with appropriate nullity annotations [19, 34]. This currently translates into more work for developers;

[1] The JDT Core includes *the* Eclipse Java compiler (incremental and batch), code assist, code search, etc.

adding annotations to new or existing code can be a formidable task: e.g. the Eclipse JDT Core contains approximately 11,000 declarations that are candidates for non-null annotation. It has been our experience in annotating moderately large code bases (including the JDT Core), that we spend most of our time constraining declarations to be non-null rather than leaving them unannotated.

Can something be done to alleviate the burden of developers? Imposing such an extra burden on developers generally translates into reduced adoption—contrast the total number of downloads, over comparable periods, of two fully automated and popular SA tools: Esc/Java2 (52,000) which requires developers to write specifications and/or annotations vs. FindBugs which doesn't (270,000)[2]. Motivated by our experiences and inspired by the success of FindBugs (built on the philosophy that simple techniques are effective too), a simple solution seemed apparent: switch the nullity interpretation of declarations to non-null by default. But since this would be contrary to the current Java default, such a switch would only be justified if significantly more than 50% of declarations are non-null in practice, and appropriate measures are taken to address backwards compatibility and migration of existing projects to the new default. We deal with both of these points in this article.

The main contribution of this paper is a carefully executed empirical study (Sections 2 and 3) confirming that:

> **In Java programs, at least 2/3 of declarations** (other than local variables) that are of reference types **are meant to be non-null, based on *design intent***.

We exclude local variables because their non-nullity can be inferred by intra-procedural analysis [2, 34]. For this study we sampled 5 open source projects totaling 722 KLOC of Java source. To our knowledge, this is the first formal empirical study of this kind—though anecdotal evidence has been mentioned elsewhere, e.g. [23, 24].

A review of languages supporting non-null annotations or types (Section 4) shows a recent trend in the adoption of non-null as a default. We believe that this, coupled with the study results, suggest that the time is ripe for non-null-by-default in Java. A second contribution of this paper is a proposal, supported by the study results, that declarations of reference types be non-null by default in Java. This new default has the advantage of better matching general practice, lightening the annotation burden of developers and being *safer* (Section 5). Our proposal also carefully addresses issues of backwards compatibility and code migration. We describe an implementation of the new default in a customized version of the Eclipse 3.3 JDT Core which supports non-null types [8, 23]. It achieves this by adopting the syntax for nullity modifiers of the Java Modeling Language (JML)—e.g. `/*@non_null*/` and `/*@nullable*/` [3]. Among other things, this choice of syntax makes it possible to support all versions of Java (not just Java 5) and non-null casts. In addition, it relieves developers from having to annotate API libraries since the tool processes the exten-

[2] To be fair, we note that FindBugs originally only supported intra-procedural analysis. It now supports inter-procedural analysis and hence, like ESC/Java2, will require developers to provide nullity annotations when this feature is enabled.

[3] For a brief period, experimental support for such annotations (i.e. inside comments) was a part of the 3.2 build.

sive collection of API specifications developed by the JML community.

Expert groups have recently been formed to look into the standardization of "Annotations for Software Defect Detection" (JSR 305) [43] and "Annotations on Java Types" (JSR 308) [16]. JSR 305 will "work to develop standard annotations (such as @NonNull) that can be applied to Java programs to assist tools that detect software defects." Making the right choice of nullity default will have an important impact on the annotation burden of developers and, we believe, can even help improve the accuracy of SA tools, particularly nullity annotation assistants (cf. Section 5.3). The next two sections cover the main contribution of the paper: the study method and study results, respectively.

2 Study

2.1 Hypothesis

The purpose of the study was to test the following hypothesis:

> In Java programs at least 2/3 of declarations (other than local variables) that are of reference types are meant to be non-null, based on design intent.

A key point of this hypothesis is that it is phrased in terms of design *intent*; i.e. whether or not the application designer intended a particular declaration to be nullable or non-null. Design intent is not something that can be reverse-engineered from the inspection of code. Tools exist which attempt to guess correct nullity annotations, but these remain guesses. As an illustration of this, consider the following interface:

```
public interface A {
  void m(Object o);
}
```

Should the parameter `o` be declared non-null? In the case of an interface, there is no code to inspect. While a tool might be able to analyze *current* implementations of the interface, if these happen to be accessible, that doesn't preclude future implementations from having different behaviors. Furthermore, let us assume that the only implementation of A has the following definition:

```
public class C implements A {
  private Object copy;
  void m(Object o) {
    copy = o.clone();
  }
}
```

Does this mean that o was meant to be non-null, or have we stumbled upon a bug in `C.m()`? Without knowledge of intent of the designers of `A`, we cannot tell. To quote Bill Pugh, FindBugs project lead, "*Static analysis tools, such as FindBugs, don't actually know what your code is supposed to do. Instead, they typically just find inconsistencies in your code*" [44].

Design intent is found most often in the heads of designers and *sometimes* recorded as

documentation, inlined code comments or machine checkable annotations and specifications. Hence, it was important for us to seek study subjects supported with design documentation, that were already annotated or for which we had access to designers who could answer our questions as we added annotations to the code.

2.2 Case Study Subjects

It was our earlier work on an ESC/Java2 case study, in the specification and verification of a small web-based enterprise application framework named `SoenEA` [45], that provided the final impetus to initiate the study reported in this paper; i.e., the burden of having to annotate (what appeared to be) *almost all* reference type declarations of an existing code base with non-null modifiers, seemed to drive home the idea that non-null should be the default. Hence, we chose to include `SoenEA` as one of our case study subjects. As our next three subjects we chose the JML checker, ESC/Java2 and the tallying subsystem of Koa, a recently developed Dutch internet voting application[4]. We chose these projects because:

- We believe that they are representative of typical designs in Java applications and that they are of a non-trivial size—numbers will be given shortly.
- The sources included some inlined design documentation and were at least partly annotated with nullity modifiers; hence we would not be starting entirely from scratch.
- We were familiar with the source code (and/or had peers that were) and hence expected that it would be easier to extend or add accurate nullity annotations. Too much effort would have been required to study and understand unfamiliar and sizeable projects in sufficient detail to be able to write correct specifications[5].
- Finally, the project sources are freely available to be reviewed by others who may want to validate our specification efforts.

All of the study subjects named so far are related to work done by the JML community and could be considered "academic" projects. Since restricting our attention to such samples might bias the study results we chose the Java Development Tools (JDT) package of Eclipse 3.3 as our final study subject. This brings in a "real" industrial grade application. Furthermore, prior to this study we were in no way involved in the development of the JDT hence there could be no bias in terms of us imposing a particular Java design style on the code base.

2.3 Procedure

2.3.1 Selection of sample files

With the study projects identified, our objective was to add nullity annotations to all of the source files, or, if there were too many, a randomly chosen sample of files. In the latter case, we fixed our sample size at 35 since sample sizes of 30 or more are generally consid-

[4] Koa was used, e.g., in the 2004 European parliamentary elections.
[5] Particularly since projects tend to lack detailed design documentation.

Encompassing Project →	JML ISU Tools	ESC Tools	SoenEA	Koa	Eclipse JDT	Total (partial)
# of files	831	455	52	459	4124	5921
LOC (K)	243	124	3	87	1018	1475
SLOC (K)	140	75	2	62	660	939
Study subject →	JML Checker	ESC/ Java2	SoenEA	Koa Tally Subsystem	Eclipse JDT Core	Total
# of files	217	216	52	29	1130	1644
LOC (K)	86	63	3	10	560	722
SLOC (K)	58	41	2	4	365	470

Table 1 General statistics of study subjects and their encompassing projects

ered "sufficiently large" [28]. Our random sampling for a given project was created by first listing the *N* project files in alphabetical order, generating 35 random numbers in the range 1..*N*, and then choosing the corresponding files.

Table 1 provides the number of files, lines-of-code (LOC) and source-lines-of-code (SLOC) [41] for our study subjects as well as the projects that they are subcomponents of. Aside from `SoenEA`, the study subjects are actually an integral (and dependant) part of a larger project. For example, the JML checker is only one of the tools provided as part of the ISU tool suite—other tools include JmlUnit and the JML run-time assertion checker compiler. The Eclipse JDT Core is one of 5 components of the Eclipse JDT, which itself is one of several subprojects of Eclipse. Overall, the source for all four projects consists of 1475 KLOC (939 KSLOC) from almost 6000 Java source files. Our study subjects account for 722 KLOC from a total population of 1644 files.

2.3.2 Annotating the sample files

We then added non-null annotations to declarations where appropriate. As an illustration of the type of situations that we faced, consider the code for the `computeCorrections()` method of the public API class `org.eclipse.jdt.core.CorrectionEngine`. (By convention, types inside packages named `internal` are not to be used by client plug-ins, while all other types are assumed to be part of the JDT Core's public API, hence `CorrectionEngine` is part of the API.) In principle, clients would only read the method's Javadoc which would allow a developer to learn that `targetUnit` and `requestor` must not be null. Nothing is said about `problem` and yet this argument is dereferenced in the method body without a test for null. Hence we have detected an inconsistency between the Javadoc and the code. Further analysis actually reveals another inconsistency: an `IllegalArgumentException` is *not* thrown when `targetUnit` is null. None-the-less, the intended nullity attributes for the three formal parameters is clearly `non_null`.

A simple example of a field declaration that we would constrain to be non-null is:

```
/**
 * Performs code correction for the given IProblem,
 * reporting results to the given correction requestor.
 *
 * Correction results are answered through a requestor.
 *
 * @param problem  the problem which describe the problem to correct.
 * @param targetUnit denote the compilation unit …. Cannot be null.
 * @param requestor the given correction requestor
 * @exception IllegalArgumentException if targetUnit or
 *            requestor is null.
 * …
 * @since 2.0
 */
public void computeCorrections(IProblem problem, … targetUnit, … requestor) throws …  {
  if (requestor == null) {
         throw new IllegalArgumentException(Messages.correction_nullUnit);
  }
  this.computeCorrections(
         targetUnit, problem.getID(),
         problem.getSourceStart(),
         problem.getSourceEnd(),
         problem.getArguments(),
         requestor);
}
```

Figure 1. Excerpt from the JDT Core API class org.eclipse.jdt.core.CorrectionEngine

```
static final String MSG1 = "abc";
```

Of course, cases in which the initialization expression is a method call require more care. Similarly we would conservatively annotate constructor and method parameters as well as method return types based on the apparent design intent. As an example of a situation where there was no supporting documentation, consider the following method:

```
String m(int paths[]) {
  String result = "";
  for(int i = 0; i < paths.length; i++) {
    result += paths [i] + ";";
  }
  return result;
}
```

In the absence of any explicit specification or documentation for such a method we would assume that the designer intended `paths` to be non-null (since there is no test for nullity and yet, e.g., the `length` field of `paths` is used). We can also deduce that the method will always return a non-null String.

2.3.3 Proper handling of overriding methods

Special care needs to be taken when annotating overriding or overridden methods. We treat non-null annotations as if defining non-null types [8, 23]. In this respect, we follow Java 5 conventions and support method

- return type covariance—as is illustrated in Figure 2;

```
public abstract class Greeting
{
  protected /*@non_null*/ String nm;

  public void set(/*@non_null*/ String nm) {
    this.nm = aNm;
  }

  public /*@non_null*/ String welcome() {
    return greeting() + " " + nm;
  }

  public abstract /*@nullable*/ String greeting();

}
```

(a) `Greeting` class

```
public class EnglishGreeting extends Greeting
{

  public void set(/*@nullable*/ String nm) // error: contravariance prohibited in Java 5
  {
    ...
  }

  public /*@non_null*/ String greeting() {  // ok: covariance supported in Java 5
  {
    return "Hello";
  }
}
```

(b) `EnglishGreeting` class

Figure 2. Illustration of nullity type variance rules for overriding methods

- parameter type invariance.

Hence, constraining a method return or parameter type to be non-null for an overriding method in one of our study sample files generally required adding annotations to the overridden method declaration(s) as well. This was particularly evident in the case of the JML checker code since the class hierarchy is up to 6 levels of inheritance for some of files that we worked on (e.g. `JmlCompilationUnit`).

2.4 Verification and Validation of Annotations

We used two complimentary techniques to ensure the accuracy of the nullity annotations that we added. Firstly, we compiled the study subjects—using the Eclipse JML JDT to be described in Section 5.1—with runtime assertion checking (RAC) enabled and then ran them against each project's standard test suite. Nullity RAC ensures that a non-null declaration is never initialized or assigned null, be it for a local variable, field, parameter or method (return) declarations. In some cases the test suites are quite large—e.g. in the order of 15,000 for the Eclipse JDT, 50,000 for JML, and 600 for ESC/Java2. While the number of tests for ESC/Java2 is lower, some of the individual tests are "big": e.g. the type checker is run on itself. In addition, we ran the RAC-enabled version of ESC/Java2 on all files in

the study samples. Though testing can provide some level of assurance, coverage is inevitably partial and depends highly on the scope of the test suites.

Hence, we also made use of the ESC/Java2 static analysis tool. In contrast to runtime checking, static analysis tools can verify the correctness of annotations for "all cases" (within the limits of the completeness of the tool); but this greater completeness comes at a price: in many cases, general method *specifications* (beyond mere nullity annotations) needed to be written in order to eliminate false warnings.

Using these techniques we were able to identify about two dozen (0.9%) incorrectly annotated declarations—excluding errors we corrected in files outside of the sample set. With these errors fixed, tests passing and ESC/Java2 not reporting any nullity warnings, we are very confident in the accuracy of the final annotations.

2.5 Metrics

Java reference types can be used in the declaration of local variables, fields, methods (return types) and parameters. In our study we considered all of these types of declaration except for local variables since they are outside of the scope of the study hypothesis. Unless specified otherwise, we shall use the term *declaration* in the remainder of this article to be a declaration other than that of a local variable.

We have two principal metrics in this study, both of which are measured on a per file basis:

- d is a measure of the number of declarations that are of a reference type and
- m is a measure of the number of declarations specified to be non-null (hence $m \leq d$).

The main statistic of interest, x, will be a measure of the proportion of reference type declarations that are non-null, i.e. m / d.

2.6 Statistics Tool

In order to gather statistics concerning non-null declarations we created a simple Eclipse JDT abstract syntax tree (AST) visitor which walks the Java AST of the study subjects and gathers the required statistics for relevant declarations. At an earlier point in the study, we made use of an enhanced version of the JML checker which both counted and inferred nullity annotations using static analysis driven by elementary heuristics. We decided instead to annotate all declarations explicitly and use a simple visitor to gather statistics. This helped us eliminate one threat to internal validity that arose due to completeness and soundness issues of the JML-checker based statistics-gathering tool.

2.7 Threats to Validity

2.7.1 Internal validity

We see two threats to internal validity. Firstly, in adding non-null constraints to the sample files we may have been excessive. As was discussed earlier, we chose to be conservative in

our annotation exercise. Furthermore, as was mentioned in Section 2.4, we ran the given project test suites with runtime checking enabled and we subjected the files to static analysis using ESC/Java2. Since ESC/Java2 is neither sound nor complete, this does not offer a guarantee of correctness, but it does increase our confidence in the accuracy of the annotations.

Finally, we note that the code samples (both before the exercise and after) are available for peer review: the JML checker is accessible from SourceForge (jmlspecs.sourceforge.net); ESC/Java2 and Koa are available from Joseph Kiniry's GForge site (sort.ucd.ie), Eclipse from Eclipse.org, and `SoenEA` (as well as the Eclipse JML JDT) are available from the authors.

2.7.2 External validity

Can we draw general conclusions from our study results? The main question is: can our sample of source files be taken as representative of typical Java applications? There are two aspects that can be considered here: the design style used in the samples, and the application domains.

Modern object-oriented programming best-practices promote e.g., a disciplined (i.e. moderate) use of `null` with the Null Object pattern recommended as an alternative [26]. Of course, not all Java code is written following recommended best practices; hence our sample applications should include such "non-OO-style" code. This is the case for some of the ESC/Java2 core classes which were designed quite early in the project history and were apparently influenced by the design of its predecessor (written in Modula-3 [12]). For example, some of the classes declare their fields as public (a practice that is discouraged [3, Item 12]) rather than using getters and setters, making it very difficult to ascertain, in the absence of supporting documentation, whether a field was intended to be non-null. Also, the class hierarchy is very flat, with some classes resembling a module in the traditional sense (i.e. a collection of static methods) more than a class.

With a five-sample set, it is impossible to claim that we have coverage in application domains, but we note that the `SoenEA` and Koa samples represent one of the most popular uses of Java—web-based enterprise applications [27].

3 Study Results

A summary of the statistics of our study samples is given in Table 2. As is usually done, the number of files in each sample is denoted by n, and the population size by N. Note that for `SoenEA`, 11 of the files did not contain any declarations of reference types, hence the population size is $41 = 52 - 11$; the reason that we exclude such files from our sample is because it is not possible to compute the proportion of non-null references for files without any declarations of reference types. We see that the total number of declarations that are of a reference type (d) across all samples is 2839. The total number of such declarations constrained to be non-null (m) is 2319. The proportion of non-null references across all files

Table 2. Distribution of the number of declarations of reference types

	JML Checker	ESC/ Java2	SoenEA	Koa TS	Eclipse JDT Core	Sum or Average
n	35	35	41	29	35	175
N	217	216	41	29	1130	1633
$\sum d_i$	420	989	231	566	633	2839
$\sum m_i$	362	872	196	424	465	2319
$\sum m_i / \sum d_i$	86%	88%	85%	75%	73%	82%
mean ($\bar{x}$)	89%	85%	84%	80%	79%	83%
std.dev.(s)	0.14	0.22	0.28	0.26	0.24	-
E (α=5%)	4.4%	6.8%	-	-	7.7%	-
$\mu_{min} = \bar{x} - E$	85%	78%	84%	80%	71%	80%

is 82%.

We also computed the mean, $\bar{x}$, of the proportion of non-null declarations on a per file basis ($x_i = d_i / m_i$). The mean ranges from 79% for the Eclipse JDT Core, to 89% for the JML checker. Also given are the standard deviation (s) and a measure of the maximum error (E) of our sample mean as an estimate for the population mean with a confidence level of $1 - \alpha = 95\%$. The overall average and weighted average (based on N) for μ_{min} are 80% and 74%, respectively. Hence we can conclude with 95% certainty that the population means are above $\mu_{min} = 74\%$ in all cases. As was explained earlier, we were conservative in our annotation exercise, hence is it quite possible that the actual overall population mean is greater than this.

All declarations were non-null (i.e. $x = 100\%$) for 46% of the files included in our sampling: 10% of JML, 9% of ESC/Java2, 13% of `SoenEA`, 7% of Koa and 7% of Eclipse JDT files. The distribution of the remaining 54% of files sampled is shown in Figure 3; each bar represents the proportion of sampled files having a value of x in the given range—following standard notation, $[a,b)$ represents the interval of values v in the range $a \leq v < b$. We see that the JML checker has no files with an x in the range [0-10%). On the other hand, the JDT has the largest proportion of files in the range [80-90%).

The mean of x by kind of declaration (fields, methods and parameters) for each of the study samples is given in Figure 4. The mean is highest for parameters in all cases except for the Eclipse JDT, and it is second highest for methods in all cases except for Koa. The Eclipse JDT has the highest proportion of non-null fields; we believe that this is because Eclipse developers make extensive use of named string constants declared as static final fields.

Hence the study results clearly support the hypothesis that in Java code, over 2/3 of declarations that are of reference types are meant to be non-null—in fact, it is closer to 3/4. It is for this reason that we recently adapted the Eclipse JDT Core to support nullity modifiers and to adopt non-null as the default. We describe our enhancements to the Eclipse JDT Core in Section 5.1. In the next section we explore related work.

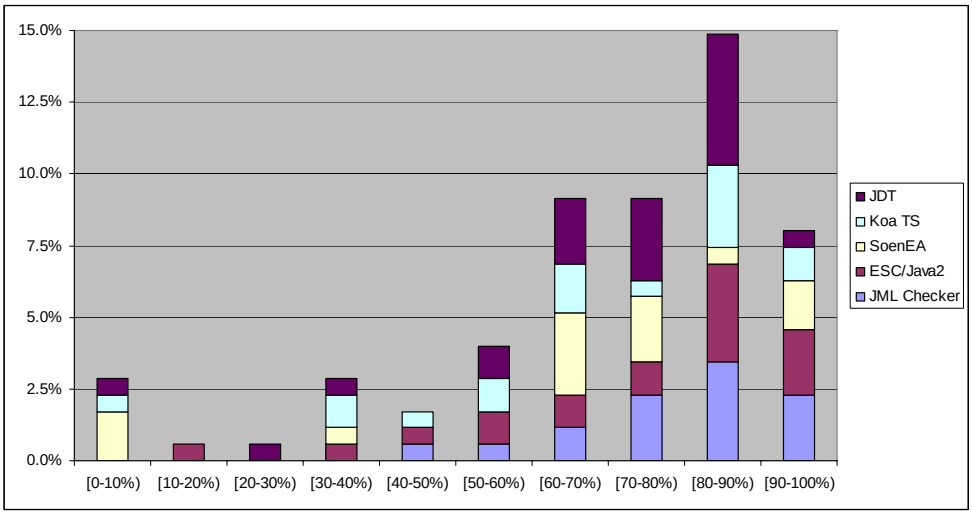

Figure 3. Distribution of the percentage of sampled files having a value for *x* (the proportion of non-null declarations) in the range [0-100%). The remaining 46% of files had *x* = 100%.

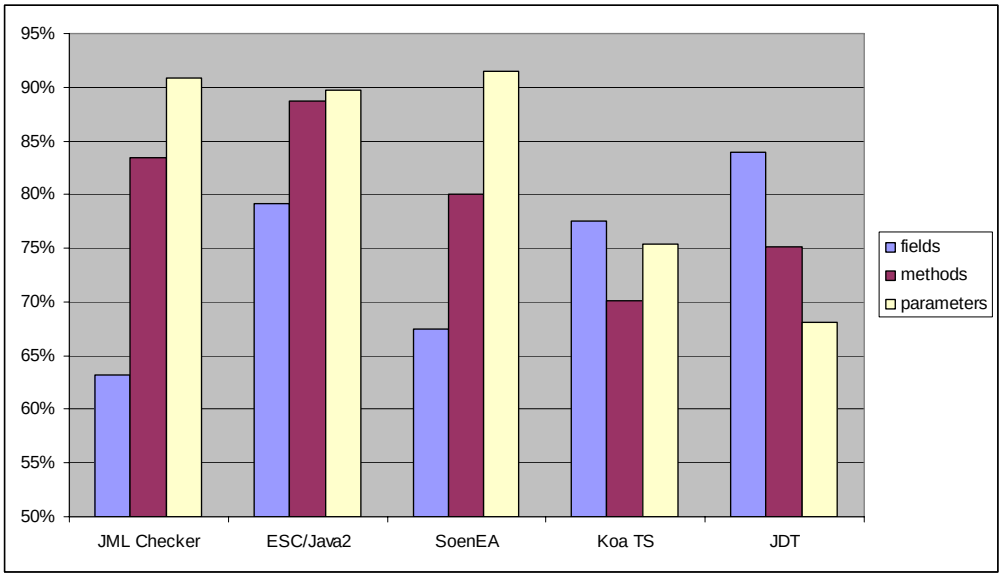

Figure 4. Mean of *x*, the proportion of non-null declarations, by kind

4 Related Work: Languages and Nullity

In this section we present a summary of the languages, language extensions and tools that offer support for non-null types or annotations. This will allow us to better assess current language design trends in the treatment of nullity, and hence better establish the context for the proposal presented in Section 5.

4.1 Languages without Pointer Types

Early promotional material for Java touted it to be an improvement over C and C++, in particular because "Java has no pointers" [25, Chapter 2], hence ridding it of "one of the most bug-prone aspects of C and C++ programming" [25, p.6]. Of course, reference types are implemented by means of pointers, though Java *disciplines* their use—e.g. the pointer arithmetic of C and C++ is prohibited in Java.

Other languages have pushed this discipline even further by eliminating null. Obvious examples are functional languages, including ML which also supports imperative features such as references and assignment. Another noteworthy example from the ML family is the Objective Caml object-oriented language. Though ML and Objective Caml support references, every reference is guaranteed to point to an instance of its base type, because the only way that a reference can be created is by taking the reference of a *value* of the base type [42]. Hence, references are (vacuously) non-null by default in these languages. Of course, a generic "pointer type" can be defined in ML or Objective Caml as a user-defined tagged union type

```
type 'a pointer = Null | Pointer of 'a ref;
```

Programmers need not go out of their way to define and use such a type since it is very seldom necessary [35]. Similar remarks can be made of early prototypical object-oriented languages like CLU. CLU (vacuously) supported non-null references by default since it did not have an explicit notion of pointers, nor did it have a special "null" value belonging to every reference type. Our study results confirm that Java developers, like Objective Caml programmers, need non-null types more often than nullable types.

4.2 Imperative Languages with Pointer Types

To our knowledge, the first imperative programming language, or language extension, with pointer types to adopt the non-null-by-default semantics is Splint [18, 20]. Splint is a "lightweight" static analysis tool for C that evolved out of work on LCLINT (a type checker of the behavioral interface specification language for C named Larch/C [17, 31]). Splint is sometimes promoted as "a better lint" because it is able to make use of programmer supplied annotations to detect a wider range of potential errors, and this more accurately, than lint. Annotations are provided in the form of stylized comments. In Splint, declarations having pointer types are assumed to be non-null by default, unless adored with `/*@null*/`. Splint does nullity checking at "interface boundaries" [20, §2]: annotations can be applied

to function parameters and return values, global variables, and structure fields but not to local variables [21, p.44].

While there are no other extensions to C supporting the non-null-by-default semantics, extensions for non-null annotations or types have been proposed. For example, Cyclone [37], described as a "safe dialect of C", supports the concept of never-NULL pointers, written as "*T*`@`" in contrast to the usual syntax "*T*`*`" for nullable pointers to *T* [30]. As another example, we note that the GNU `gcc` supports a form of non-null annotation for function parameters only; e.g. an annotation of the form

```
__attribute__((nonnull (1, 3)))
```

after a function signature would indicate that the first and third arguments of the function are expected to be non-null [46, §5.24].

4.3 Object-oriented Languages (Non-Java)

4.3.1 Eiffel

The recent ECMA Standard of the Eiffel programming language introduces the notions of *attached* and *detachable* types [13]. These correspond to non-null (or non-void types, as they would be called in Eiffel) and nullable types, respectively. By default, types are attached—which, to our knowledge, makes Eiffel the first non research-prototype object-oriented language to adopt this default. Eiffel supports covariance in method return types and invariance of parameter types except with respect to parameter nullity, for which is supports contravariance [13, §8.10.26, §8.14.5]—see Table 3 on page 14.

Prior to the release of the ECMA standard, types were detachable by default. Hence a migration effort for the existing Eiffel code base has been necessary. Special consideration has been given to minimizing the migration effort in the form of compiler / tool support.

4.3.2 Spec#

Spec# is an extension of the C# programming language that adds support for contracts, checked exceptions and non-null types. The Spec# compiler statically enforces non-null types and generates run-time assertion checking code for contracts [2]. The Boogie program verifier can be used to perform extended static checking of Spec# code [11]. While Spec# code cannot generally be processed by C# compilers, compatibility can be maintained by placing Spec# annotations inside stylized comments (`/*^` … `^*/`) as is done with other annotation languages like Splint and JML (which use `/*@` … `*/`).

Introduction of non-null types (vs. annotations) requires care, particularly with respect to field initialization in constructors and helper methods [22]. Open issues also remain with respect to arrays and non-null static fields, for which the Spec# compiler resorts to run-time checking to ensure type safety [1, §1.0]. For reasons of backwards compatibility, a reference type name *T* refers to possibly null references of type *T*. The notation *T*! (or `/*^ ! ^*/`, with a special shorthand of `/*!*/`) is used to represent non-null references of type *T*.

As of the February 2006 release of the Spec# compiler, it is possible to use a compiler option to enable a non-null-by-default semantics. When this is done, *T*? can be used to denote possibly null references to *T*. Note however, Spec# has no class level modifiers which would allow the default nullity to be set for a single class. We note in passing that of all the languages discussed in this section, Spec# is the only one with annotation suffixes (i.e. that appear *after* the type name rather than before). Nullity return type and parameter type variance for overriding methods in Spec# conforms to the type invariance rules of C#—i.e., types must be the same.

4.3.3 Nice

Nice is a new programming language whose syntax is superficially similar to that of Java. It can be thought of as an enriched variant of Java supporting parametric types, multi-methods, and contracts, among other features [4, 5]. Nullable types are called *option types* in Nice terminology. It is claimed that Nice programs are free of null pointer exceptions. By default, a reference type name *T* denotes non-null instances of *T*. To express the possi-

Table 3. Summary of support for non-null

Language / Tool	**Type / Anno-ta-tion**	**Default**	**Member declaration modifier (prefix) for**		**Non-null Annotation (A) and Checking** at run-time (R), or statically at compile-time (S). Abbr.: all (✓=ARS); none (×)					**Overriding method type variance w.r.t.**		**Anno. API of std libraries?**	**Class modifier?**	**Compiler option to invert default**
			non-null	**nullable**	**method**	**param-eters**	**field**	**local variables**	**array elements**	**result** nullity	**param -eter** nullity			
Splint	anno.	**non-null**	/*@notnull*/	/*@null*/	AS	AS	AS	S	×	N/A	N/A	×	N/A	×
Cyclone	**type**	nullable	@, e.g. T@	(std., e.g. T*)	✓	✓	(✓)	✓	×	N/A	N/A	×	N/A	×
Eiffel	**type**	**non-null**	!	?	✓	✓	✓	✓	✓	covariance	contravar.	✓	×	×
Spec#	**type**	nullable	! (suffix)	? (suffix)	✓	✓	✓	AS	×	invariance	invariance	(✓)	×	✓
Nice	**type**	**non-null**	!	?	AS	AS	AS	AS	✓	covariance	contravar.	×	×	×
Java support														
JML	anno.	**non-null**	/*@non_null*/	/*@nullable*/	✓	✓	✓	AS	✓	covariance	**invariance**	✓	✓	✓
IntelliJ IDEA (≥ 5.1)	anno.	nullable	@NotNull	@Nullable	AS	AS	AS	AS	×	covariance	contravar.	×	×	×
Nully (IDEA plug-in)	anno.	nullable	@NonNull	@Nullable	✓	(✓)	×	(✓)	×	no restriction	no restriction	×	×	×
FindBugs (≥ 0.8.8)	anno.	nullable	@NonNull	@CheckForNull	AS	AS	×	S	×	no restriction	no restriction	×	×	×
JastAdd + NonNull Extension	anno.	nullable	[NotNull]	[MayBeNull]	AS	AS	AS	AS	×	invariance	invariance	×	×	×
Eclipse JML JDT (3.3)	**type**	**non-null**	/*@non_null*/	/*@nullable*/	✓	✓	✓	✓	✓	covariance	**invariance**	✓	✓	✓

bility that a declaration of type T might be null, one prefixes the type name with a question mark, $?T$ [6].

4.4 Java Support for Non-Null

4.4.1 FindBugs

The FindBugs tool does static analysis of Java class files and reports *common* programming errors; hence, by design, the tool forgoes soundness and completeness for utility; an approach that is not uncommon for static analyzers [33]. In order to increase the accuracy of error reporting related to nullity and to better be able to assign blame, support for nullity annotations for return types and parameters was recently added—annotations can be applied to local variables but they are effectively ignored. The annotations are: `@NonNull`, used to indicate that the declared entity cannot be null, and `@CheckForNull`, indicating that a null value should be expected and hence, any attempted dereference should be preceded with a check [34].

Although FindBugs has been applied to production code (e.g. Eclipse 3.0 source), nullity annotations have not yet been used on such samples. Our study results suggest that when this happens, specifiers are likely to find themselves decorating most reference type declarations with `@NonNull`.

4.4.2 Nully and the IntelliJ IDEA

Nully is an IntelliJ IDEA plug-in that can be used to detect potential null dereferences at edit-time, compile-time and run-time. It can be applied to method return types and parameters as well as local variables but not fields. Nully documentation claims that it supports run-time checking of non-null constraints on local variables but this could not be confirmed. Non-null checking of parameters is only provided in the form of run-time checks [38].

There has yet to be an official release of Nully and it is not clear whether the tool is still being developed, particularly since the latest release of the IntelliJ IDEA marks the introduction of its own (proprietary) annotations `@NotNull` and `@Nullable` [36]. IDEA supports edit-time and compile-time checks, but not run-time checks of non-null. IDEA supports nullity return type covariance and parameter type contravariance. We note that this is incompatible with Java which requires invariance for parameter types.

4.4.3 JastAdd

JastAdd is an open source "aspect-oriented compiler construction system" whose architecture promises to support compiler feature development in a more modular fashion than is usually possible [32]. As a demonstration of this flexibility, support for non-null types has been defined as an "add-on" to the JastAdd based Java 1.4 compiler [14]. The implemented type system is essentially that of Fähndrich and Leino [23]. In fact, they make use of the same annotations, which makes the extension incompatible with standard Java (of course, it

should be rather easy to rename the annotations to be conformant to Java 5 annotation syntax). Like Spec#, nullity modifiers of overriding methods must match exactly, both for return and parameter types. Finally, we note that the JastAdd compiler currently only does type checking, without apparent support for code generation.

4.4.4 Java Modeling Language

The Java Modeling Language (JML) originated from Iowa State University (ISU) under the leadership of Gary Leavens. JML is currently the subject of study and use by a dozen international research teams [39]. It is a behavioral interface specification language that, in particular, brings support for Design by Contract (DBC) to Java [40]. Using JML, developers can write complete interface specifications for Java types. JML annotated Java code can be compiled with standard Java compilers because annotations are contained in stylized comments whose first character is @. JML enjoys a broad range of tool support including [7, 39]:

- Jmldoc that generates documentation in a manner that is similar to `Javadoc`, but incorporating JML specifications.
- `jmlc`, the ISU JML run-time assertion checker compiler.
- ESC/Java2, an extended static checker that provides a compiler-like interface to fully automated checking of JML specifications. Like similar tools, ESC/Java2 compromises soundness and completeness for efficacy and utility.
- LOOP tool that can be used in conjunction with PVS to perform complete verification of JML annotated Java applications.
- JmlUnit, a tool for generating JUnit test suites using JML specifications as test oracles.
- JMLKEY tool that offers support for model-driven design, principally from UML class diagrams, with JML as a design (constraint) specification language. The tool supports the complete JavaCard language.

JML has nullity modifiers (`non_null` and `nullable`) and it recently adopted a non-null-by-default semantics for reference type declarations [9].

4.5 Summary

A summary of the languages, extensions and tools covered in this section, is given in Table 3. Two key observations are that for all languages and tools *not* using Java 5 annotations there seems to be a trend in adopting

- non-null type system over non-null annotations, with
- non-null as the default.

Even well established languages like Eiffel are making the bold move of switching to the new default. The apparent trend in the evolution of languages supporting pointers would seem to indicate that the time is ripe to consider a switch in Java from nullable-by-default to non-null-by-default. A concrete proposal for this is given in the next section.

5 Non-Null by Default in Java

The study results suggest that an adoption of non-null-by-default in Java would have the advantage of

- Better matching general practice: the majority of declarations will be *correctly* constrained to be non-null, and hence,
- Lightening the annotation burden of developers; i.e. there are fewer declarations to explicitly annotate as nullable.

In addition, and possibly more importantly, the new default would be *safer*:

- Processing of null generally requires extra programming logic and code to be handled correctly. With the new default, an annotation now explicitly *alerts* developers that null values need to be considered.
- If a developer delays or forgets to annotate a declaration as nullable, this will at worst limit functionality rather than introducing unwanted behavior (e.g., null pointer exceptions)—also, limited functionality is generally easier to detect than potential null pointer exceptions.

5.1 An Implementation of Non-Null by Default: Eclipse JML JDT

Guided by our experiences in the implementation of non-null types and non-null by default in JML tools [8], we have recently completed a similar implementation as an extension to the Eclipse JDT Core—which we will call the Eclipse JML JDT Core (or JML JDT for short). Since 3.2, the Eclipse JDT Core has supported intra-procedural flow analysis for potential null problems. The JML JDT builds upon and extends this base.

One of the first and most obvious questions which we faced was: which annotation syntax should the tool support? Until the JSR 305 [43] efforts are finalized, adhering too JML-like annotations seemed advantageous since it would allow the JML JDT to

- support nullity annotations for all versions of Java, not just Java 5 (many projects, including Eclipse, are still at Java 1.4).
- support casts to non-null (these are necessary to counter false positives). Java 5 annotations cannot be used for this purpose, though JSR 308 [16] is likely to address this limitation as of Java 7.
- naturally recognize and process the extensive collection of JML API specifications which have been developed over the years by the JML community.

Using JML syntax also means that the source files will be more easily amenable to processing by the complimentary suite of JML tools (cf. Section 4.4.4). Once the standard Java 5 annotations are defined, it will be rather easy to adapt the tool to process these annotations as well.

A summary of the JML JDT Core capabilities is given at the bottom of Table 3 on page 14. In particular we note that it supports edit-time, compile-time and runtime checking of nullity annotations—see Figure 5.

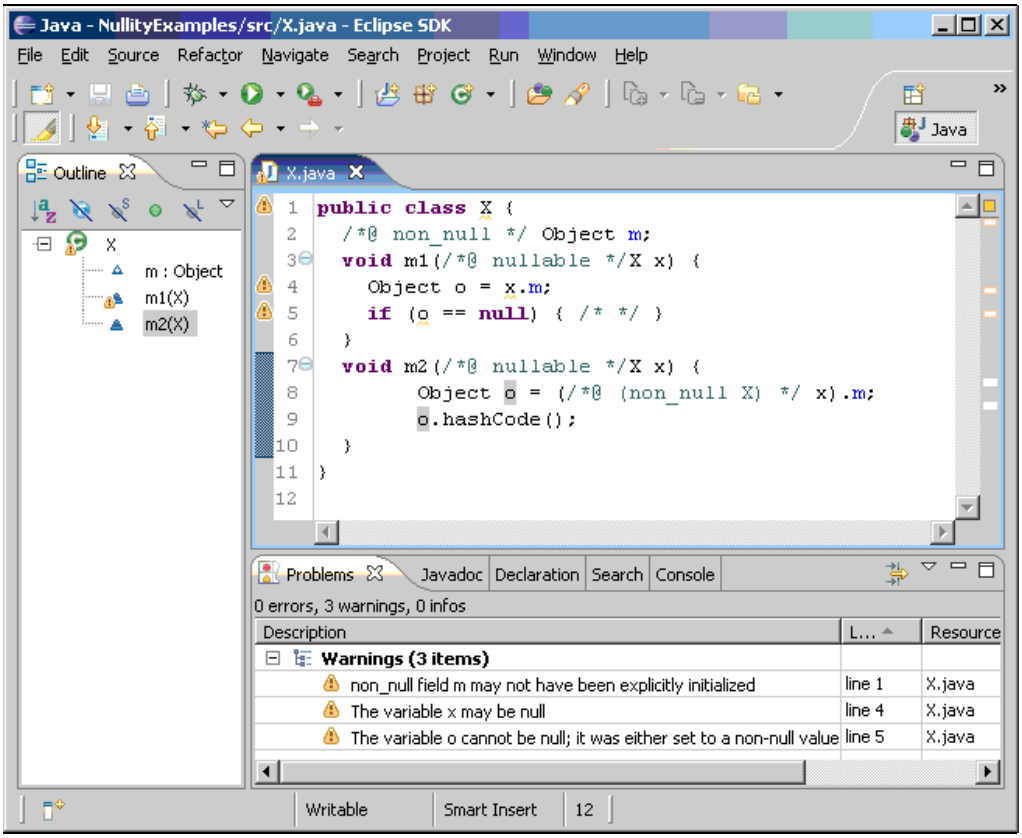

Figure 5. Screenshot of the Eclipse JML JDT

5.2 Backwards Compatibility, and Migration to the New Default

The JML JDT supports tool-wide and project specific settings for the choice of nullity default. A finer grain of control is provided in the form of type (i.e., class or interface) scoped declaration modifiers named `nullable_by_default`, and `non_null_by_default`. Applying the first of these modifiers to a type enables developers to recover the nullable-by-default semantics; i.e., all reference type declarations in the class or interface that are not explicitly declared non-null are interpreted as nullable. Note that the scope of the `*_by_default` modifiers is strictly the type to which it is applied; hence, it is not inherited by subtypes.

In addition to these class- and interface-scoped modifiers, a script is available that enables developers to add the `nullable_by_default` modifier to all classes and interfaces in a given project. This allows the global project default to be non-null while, gradually, and as needed, files can be reviewed and updated one-by-one to conform to the new default by

- adding `nullable` modifiers,
- optionally removing explicit `non_null` modifiers (if there are any), and finally,
- removing the `nullable_by_default` modifier.

(This is the process which we have been following in our gradual migration of the thousands of JML-annotated source files which are part of our tool and case study repositories.) Of course, such a porting effort also drives home the importance of adopting the right default semantics as early as possible.

5.3 Helping Automated Annotation Assistants Too

The best time to add nullity annotations is when code is being created since at that time the original author is available to record his or her design intents. Adoption of non-null by default means that developers will have fewer declarations to annotate in the new code that they write.

What can be done about existing unannotated code? There exist a few fully automatic static analysis tools, called annotation assistants, which can help in adding nullity annotations (among other specification constructs) to source files. Does the existence of such tools eliminate the need to change nullity defaults? We believe not. For the most part, these annotation assistants are research prototypes and would be unable to cope with large code bases. In a recent study, Engelen lists three non-null annotation assistants [15]: the JastAdd Nullness Inferrer [14], Houdini [24] and the more recent CANAPA [10]. Of these tools, only Houdini has published performance results. Houdini makes use of ESC/Java2 to test its annotation guesses, and it is capable of processing less than 500 lines per hour (though admittedly it infers more than non-null annotations). On the other hand, the accuracy of its non-null annotations is reported at 79% [24].

Our study results have show that we can get comparable accuracy simply by assuming that declarations are non-null. We believe that the switch to non-null by default can actually be an aid to annotation assistants. Tools can assume that declarations are non-null (and hence get a majority of annotations correct) and only opt for nullable if there is clear evidence that the declaration can be assigned null.

6 Conclusion

In this paper, we report on a novel study of five open projects (totaling over 722 KLOC) taken from various application domains The study results have shown that on average, one can expect 74% of reference type declarations to be non-null by design in Java. We believe that this report is timely, as we are witnessing the increasing emergence of static analysis (SA) tools using non-null annotations to detect potential null pointer exceptions. Before too much code is written under the current nullable-by-default semantics, it would be preferable that Java be adapted, or at least a *standard* non-null annotation-based extension be defined, in which declarations are interpreted as non-null by default. This would be the first step in the direction of an apparent trend in the modern design of languages (with pointer

types), which is to support non-null types and non-null by default.

One might question whether a language as widely deployed as Java can switch nullity defaults. If the successful transition of Eiffel is any indication, it would seem that the switch can be achieved if suitable facilities are provided to ease the transition. We believe that our Eclipse JML JDT offers such facilities in the form of support for project-specific as well as fine-grained control over nullity defaults (via type-scope annotations). Until standard Java 5 nullity annotations are adopted via JSR 305, we have designed the JML JDT to recognize JML style nullity modifiers, hence allowing the tool to reuse the comprehensive set of JML API specifications (among other advantages). Adding nullity annotations is time consuming. By adopting JML-style nullity modifiers we also offer developers potentially increased payback, in that all other JML tools will be able to process the annotations as well—including the SA tool ESC/Java2 and JmlUnit, which generates JUnit test suites using JML specifications and annotations as test oracles.

As a natural continuation of our work, we have begun enhancements to the Eclipse JML JDT to allow runtime and compile-time (SA) support for Design by Contract via a core of JML's syntax.

Acknowledgments

We are grateful to Joseph Kiniry for providing access to, and assistance on, the Koa source, and for discussions about Eiffel. We thank Frederic Rioux for his contribution to the initial study efforts. Research support was provided by NSERC of Canada (261573-03) and the Quebec FQRNT (100221).

References

[1] M. Barnett, R. DeLine, B. Jacobs, M. Faehndrich, K. R. M. Leino, W. Schulte, and H. Venter, "The Spec# Programming System: Challenges and Directions". *International Conference on Verified Software: Theories, Tools, Experiments*, Zürich, Switzerland, 2005.

[2] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview". In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean editors, *Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, Marseille, France, March 10-14, vol. 3362 of *LNCS*. Springer, 2004.

[3] J. Bloch, *Effective Java Programming Language Guide*. Addison-Wesley, 2001.

[4] D. Bonniot, "Using kinds to type partially-polymorphic methods", *Electronic Notes in Theoretical Computer Science*, 75:1-20, 2003.

[5] D. Bonniot, "The Nice programming language", http://nice.sourceforge.net/, 2005.

[6] D. Bonniot, "Type safety in Nice: Why programs written in Nice have less bugs", 2005.

[7] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An Overview of JML Tools and Applications", *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212-232, 2005.

[8] P. Chalin, “Towards Support for Non-null Types and Non-null-by-default in Java”. *Proceedings of the 8th Workshop on Formal Techniques for Java-like Programs (FTfJP'06)*, Nantes, France, July, 2006.
[9] P. Chalin and F. Rioux, “Non-null References by Default in the Java Modeling Language”. *Proceedings of the Workshop on the Specification and Verification of Component-Based Systems (SAVCBS)*, Lisbon, Portugal, Sept. ACM Press, 2005.
[10] M. Cielecki, J. Fulara, K. Jakubczyk, and L. Jancewicz, “Propagation of JML non-null annotations in Java programs”. *Proceedings of the International Conference on Principles and Practices of Programming In Java (PPPJ'06)*, Mannheim, Germany, 2006.
[11] R. DeLine and K. R. M. Leino, “BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs”, Microsoft Research, Technical Report, 2005.
[12] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe, “Extended Static Checking”, Compaq Systems Research Center, Research Report 159. December, 1998.
[13] ECMA International, “Eiffel Analysis, Design and Programming Language”, ECMA-367. June 2005.
[14] T. Ekman and G. Hedin, “Modular implementation of non-null types for Java”, Lund University, (submitted for publication), 2005.
[15] A. F. M. Engelen, “Nullness Analysis of Java Source Code”. Master's thesis. Nijmegen Institute for Computing and Information Sciences, Radboud University Nijmegen, Netherlands, 2006.
[16] M. Ernst and D. Coward, “Annotations on Java Types”, JCP.org, JSR 308, 2006.
[17] D. Evans, “Using Specifications to Check Source Code”, MIT, MIT/LCS/TR 628. June, 1994.
[18] D. Evans, “Static Detection of Dynamic Memory Errors”. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia, Pennsylvania, United States. ACM Press, 1996.
[19] D. Evans, “Annotation-Assisted Lightweight Static Checking”. *First International Workshop on Automated Program Analysis, Testing and Verification*, February, 2000.
[20] D. Evans, “Splint User Manual”, Secure Programming Group, University of Virginia. June 5, 2003.
[21] D. Evans and D. Larochelle, “Improving security using extensible lightweight static analysis”, *IEEE Software*, 19(1):42-51, 2002.
[22] M. Fähndrich and K. R. M. Leino, “Non-Null Types in an Object-Oriented Language”. *Proceedings of the Workshop on Formal Techniques for Java-like Languages*, Malaga, Spain, 2002.
[23] M. Fähndrich and K. R. M. Leino, “Declaring and Checking Non-null Types in an Object-Oriented Language”, in *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA'03*: ACM Press, pp. 302-312, 2003.
[24] C. Flanagan and K. R. M. Leino, “Houdini, an Annotation Assistant for ESC/Java”. *Proceedings of the International Symposium of Formal Methods Europe*, Berlin, Germany, vol. 2021, pp. 500-517. Springer, 2001.
[25] D. Flanagan, *Java in a Nutshell: A Desktop Quick Reference*. O'Reilly, 1996.
[26] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
[27] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
[28] J. E. Freund and R. E. Walphole, *Mathematical Statistics*. Prentice-Hall, 1980.
[29] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Addison-Wesley Professional, 2005.
[30] D. Grossman, M. Hicks, T. Jim, and G. Morrisett, “Cyclone: a Type-safe Dialect of C”, *C/C++ Users Journal*, 23(1), 2005.

[31] J. V. Guttag and J. J. Horning, *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
[32] G. Hedin and E. Magnusson, "JastAdd--an aspect-oriented compiler construction system", *Science of Computer Programming*, 47(1):37-58, 2003.
[33] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy", *ACM SIGPLAN Notices*, 39(12):92-106, 2004.
[34] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs", *SIGSOFT Software Engineering Notes*, 31(1):13-19, 2006.
[35] INRIA, "Pointers in Caml", in *Caml Documentation, Specific Guides*, http://caml.inria.fr/resources/doc/, 2006.
[36] JetBrains, "Nullable How-To", in *IntelliJ IDEA 5.x Developer Documentation*: JetBrains, 2006.
[37] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C". *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June, pp. 275-288, 2002.
[38] K. Lea, "Nully": https://nully.dev.java.net/, https://nully.dev.java.net/, 2005.
[39] G. T. Leavens, "The Java Modeling Language (JML)": http://www.jmlspecs.org, 2006.
[40] G. T. Leavens and Y. Cheon, "Design by Contract with JML", Draft paper, 2005.
[41] R. Park, "Software Size Measurement: A Framework for Counting Source Statements", CMU, Software Engineering Institute, Pittsburgh CMU/SEI-92-TR-20, 1992.
[42] L. C. Paulson, *ML for the Working Programmer*. Cambridge University Press, 1991.
[43] W. Pugh, "Annotations for Software Defect Detection", JCP.org, JSR 305, 2006.
[44] W. Pugh, "How do you fix an obvious bug?" http://findbugs.blogspot.com/, 2006.
[45] F. Rioux and P. Chalin, "Improving the Quality of Web-based Enterprise Applications with Extended Static Checking: A Case Study", *Electronic Notes in Theoretical Computer Science*, 157(2):119-132, 2006.
[46] R. Stallman, "Using the GNU Compiler Collection (GCC): GCC Version 4.1.0", Free Software Foundation, 2005.