

Reducing the Use of Nullable Types through Non-null by Default and Monotonic Non-null*

Patrice Chalin, Perry R. James, Frédéric Rioux

Dependable Software Research Group,
Dept. of Computer Science and Software Engineering,
Concordia University, Montréal, Québec, Canada
 {chalin,perry,fred}@dsrg.org

Abstract. With Java 5 annotations, we note a marked increase in tools that can statically detect potential null dereferences. To be effective such tools require that developers annotate declarations with nullity modifiers and have annotated API libraries. Unfortunately, in our experience specifying moderately large code bases, 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 totaling 700 KLOC which confirms that on average, 75% of reference declarations are meant to be non-null, by design. Guided by these results, we propose adoption of a non-null-by-default semantics. This new default has advantages of better matching general practice, lightening developer annotation burden and being safer. We also describe the Eclipse JML JDT, a tool supporting the new semantics, including the ability to read the extensive API library specifications written in the Java Modeling Language (JML). Issues of backwards compatibility are addressed. In a second phase of the empirical study, we analyzed the uses of null and noted that over half of the nullable field references are only assigned non-null values. For this category of reference we introduce the concept of monotonic non-null type and illustrate the benefits of its use.

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) Core¹ 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

* This article is an extended version 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, LNCS 4609, pp. 227-247. Springer, 2007.

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

introduction in Java 5 of Annotations [37, §9.7] seems to have contributed to an increase in support for non-null static checking in Java.

It is well known that SA tools tend to yield a high proportion of false positives unless code and support libraries are supplemented with appropriate nullity annotations [25, 42].

This currently translates into more work for developers; 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 this burden on 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 [46] (52,000) which requires developers to write specifications and/or annotations vs. FindBugs [41] which doesn't (270,000). 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 article is a carefully executed empirical study (Sections 2 and 3) confirming 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*.

We exclude local variables because their non-nullity can be inferred by intra-procedural analysis [2, 42]. 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. [29, 31].

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 on 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 JDT—called the Eclipse JML JDT² [11]—which supports non-null types [9, 29]. It achieves this by adopting the syntax for nullity

² Also known as JML4.

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 extensive 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) [59] and “Annotations on Java Types” (JSR 308) [22]. 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 for developers and, we believe, can even help improve the accuracy of SA tools, particularly nullity annotation assistants (see Section 5.3).

In a second phase of our empirical study (Section 6) when we analyzed the use of nullable types, we noticed that over half of the nullable fields follow what we call a monotonic non-null pattern. As a third contribution (Section 7), we define monotonic non-null types and describe the benefits of their use, particularly in the context of multithreaded programs. We also describe our Eclipse JDT enhancement supporting this new concept through the introduction of the JML type modifier `/*@eventually_non-null */`.

The next two sections cover 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. For the most part, 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,

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

if these happen to be accessible, that does not 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*” [60].

Design intent is found most often in the heads of designers and *sometimes* recorded as documentation, in-lined 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` [62], that provided the final impetus to initiate the study reported in this paper: 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 two main JML verification tools—the JML checker [8] and ESC/Java2 [14]—and the tallying subsystem of Koa [47], a recently developed Dutch internet voting application⁴. 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 in-lined 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 (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 specifications⁵.

⁴ Koa was used, e.g., in the 2004 European parliamentary elections.

⁵ Particularly since projects tend to lack detailed design documentation.

Encompassing Project →	Common JML Tools	ESC Tools	SoenEA	Koa	Eclipse JDT	Total
# 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

- 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 considered “sufficiently large” [35]. 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) [57] 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 Common⁶ 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,

⁶ Formerly called the Iowa State University (ISU) tools.

```

/**
 * 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`

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. As was previously mentioned, due to the large number of source files in the JML Checker, ESC/Java2 and Eclipse JDT Core, a sampling of 35 files was made for these cases. This resulted in a total of 42.6 KLOC for all sampled files, which includes the entire SoenEA and Koa subjects.

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` as shown in Figure 1. (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. Nonetheless, the intended nullity attributes for the three formal parameters is clearly

`nonNull`.

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

```
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:

```
public 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. When evidence (either in the form of specifications, documentation or the code itself) was inconclusive, we left declarations as nullable. It is for this reason that we consider our annotation exercise to be conservative. A simple example of this would be the following method:

```
public String m(int i, String s) {  
    if (i > this.n) {  
        s = m2(s);  
    }  
    return s;  
}
```

In this case, without API specifications or evidence of the parameter `s` being dereferenced, and assuming `m2()` provides no evidence of the possible expectation that `s` be non-null, we take both the method and its parameter `s` to be declared as nullable.

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 [9, 29]. In this respect, we follow Java 5 conventions and support method

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

```

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

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

    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

2.4 Verification and Validation of Annotations

We used two complementary techniques to ensure the accuracy of the nullity annotations that we added. Firstly, we compiled each of the study subjects—using the Eclipse JML JDT to be described in Section 5.1—with runtime assertion checking (RAC) enabled and then ran it 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. on the order of 15,000 tests 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 (i.e., a version that performed runtime checks of ESC/Java2’s nullity annotations) on all files in the study samples; the increased number of checks of ESC/Java2’s nullity annotations increased our confidence in their correctness. Though testing can provide some level of assurance, coverage is inevitably partial and depends highly on the scope of the test suites.

Secondly, 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 the number of declarations that are of a reference type and
- m is 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 enhanced JML-checker based statistics-gathering feature.

2.7 Threats to Validity

2.7.1 Internal validity

We see two threats to internal validity—i.e., threats caused by the manner in which we set up and conducted the experiment. 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. Secondly, 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 Source Forge (jmlspecs.sourceforge.net); ESC/Java2 and Koa are available from Joseph Kiniry’s GForge site (sort.ucd.ie), Eclipse JDT from Eclipse.org, and [SoenEA](#) is available from the authors. The Eclipse JML JDT is available from the JML Source Forge site (under the project name JML4) [49].

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 a disciplined (i.e. moderate) use of `null`, with the Null Object pattern recommended as an alternative [33]. Of course, not all Java code is written following recommended best practices; hence, if possible, our sample applications should include such “non-OO-style” code. This happened to be the case for some of the ESC/Java2 core classes which were designed quite early in the project’s history and were apparently influenced by the design of its predecessor (written in Modula-3 [16]). For example, some of the classes declare their fields as public (a practice that is discouraged [4, 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 all application domains, but we note that the [SoenEA](#) and [Koa](#) samples represent one of the most popular uses of Java—web-based enterprise applications [34].

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 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 $a \leq v < b$.

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	-
$E (\alpha=5\%)$	4.4%	6.8%	-	-	7.7%	-
$\mu_{\min} = \bar{x} - E$	85%	78%	84%	80%	71%	80%

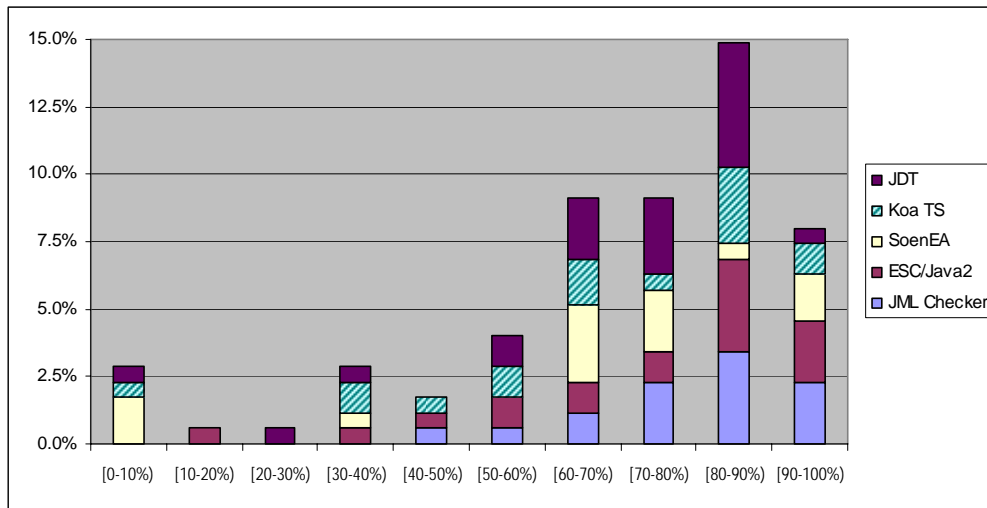


Figure 3. Distribution of the percentage of sampled files having a value for x (the proportion of non-null declarations) in the range [0-100%]. The remaining 46% of files had $x = 100\%$.

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 with 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.

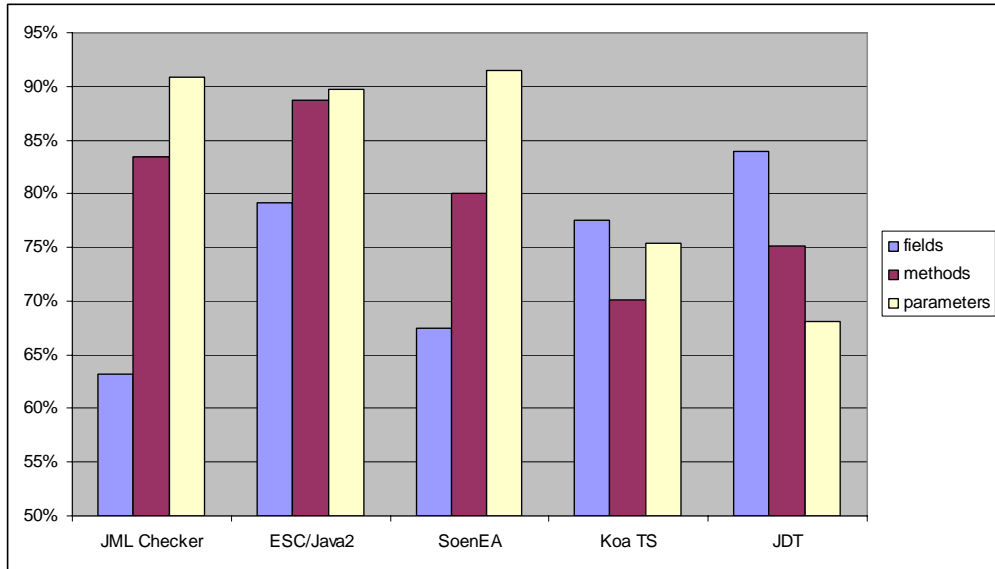


Figure 4. Mean of x , the proportion of non-null declarations, by kind

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.

Early promotional material for Java touted it to be an improvement over C and C++, in particular because “Java has no pointers” [32, Chapter 2], hence ridding it of “one of the most bug-prone aspects of C and C++ programming” [32, 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 also eliminating null. Next, we discuss such languages, i.e., languages that (regardless of the paradigm) have references but neither pointer types nor null. This is followed by a general discussion of nullity in the context of languages with pointer types, object-oriented languages with pointer types and finally, Java language extensions and tools related to nullity.

4.1 Languages without Pointer Types and without Null

Obvious examples of languages without pointer types 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 [58]. 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 [43]. Our study results confirm that Java developers, like Objective Caml or ML programmers, need non-null types more often than nullable types.

Similar remarks to those made for functional languages can be also be made of early prototypical object-oriented languages like CLU [52]. 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. Yet other languages eliminated the need for null by making use of a form of lazy instantiation. This was the case of a core language defined by Palsberg and Schwartzbach in which they offer a semantics for type substitutivity in object-oriented programs [55]. Later, though, Palsberg and Schwartzbach define a “calculus of pointers” in which they consider null (or nil, as it was referred to back then), an essential part of the calculus [56].

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 [24, 26]. 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 [23, 39]). 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 adorned with `/*@null */`. Splint does nullity checking at “interface boundaries” [26, §2]: annotations can be applied to function parameters and return values, global variables, and structure fields but not to local variables [27, 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 [45], 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* [38]. Microsoft’s Source-code Annotation Language (SAL) is a set of macros that can be used to annotate C/C++ declarations, especially buffer declarations, with metadata that can be used for static analysis and that also translates into runtime checks [54]. Annotations

constrain buffers to be non-null unless the annotation macro name ends with `opt_`. In this sense SAL does not enforce a non-null default on buffer declarations but does ensure that a naming convention makes it clear when an annotated buffer can be null (since the annotation name must have the `opt_` suffix). 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 [63, §5.24].

It appears that even early imperative programming languages from the ALGOL family supported the notion of non-null types. In fact, Mary/2 [61] offered both nullable and non-null types, each declared using its own syntax, namely, `ref T` and `protean T`, respectively. Like in ALGOL, uses could be nested: e.g., `protean ref T` represents a non-null pointer to a nullable pointer to `T`. Since there is no ambiguity in the type declaration syntax, there is no notion of a default nullity; i.e., a developer needs to make a choice and then use the appropriate syntactic construct.

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 [17]. 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 it

Table 3. Summary of support for non-null

Language / Tool	Type (T) / Annotation (A)	Default: non-null (NN) or nullable (nu)	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	field	local var	array elt	result nullity	parameter nullity			
Splint	A	NN	/*@notnull*/	/*@null*/	AS	AS	AS	S	✗	N/A	N/A	✗	N/A	✗
Cyclone	T	nu	@, e.g. T@	(std., e.g. T*)	✓	✓	(✓)	✓	✗	N/A	N/A	✗	N/A	✗
Eiffel	T	NN	!	?	✓	✓	✓	✓	✓	covariance	contravar.	✓	✗	✗
C#	T	NN	(none)	? (suffix)	✓	✓	✓	✓	AS	N/A	N/A	✗	N/A	✗
Spec#	T	nu	! (suffix)	? (suffix)	✓	✓	✓	✓	AS	invariance	invariance	(✓)	✗	✓
Nice	T	NN	!	?	AS	AS	AS	AS	✓	covariance	contravar.	✗	✗	✗
Java support														
JML	A	NN	/*@non_null*/	/*@nullable*/	✓	✓	✓	AS	AS	covariance	invariance	✓	✓	✓
IntelliJ IDEA (≥ 5.1)	A	nu	@NotNull	@Nullable	AS	AS	AS	AS	✗	covariance	contravar.	✗	✗	✗
Nully (IDEA plug-in)	A	nu	@Nonnull	@Nullable	✓	(✓)	✗	(✓)	✗	no restriction	no restriction	✗	✗	✗
FindBugs (≥ 0.8.8)	A	nu	@Nonnull	@CheckForNull	AS	AS	✗	S	✗	covariance	contravar.	✗	✗	✗
JastAdd + NonNull Extension	A	nu	[NotNull]	[MaybeNull]	AS	AS	AS	AS	✗	invariance	invariance	✗	✗	✗
Eclipse JML JDT (3.4)	T	NN	/*@non_null*/	/*@nullable*/	✓	✓	✓	✓	✓	covariance	invariance	✓	✓	✓

supports contravariance [17, §8.10.26, §8.14.5]—see Table 3. Since a non-null type T is a subtype of T , covariance on return types means that overriding methods can declare that they return non-null even if the overridden method is declared nullable; contravariance on method parameters means that overriding methods can relax the nullity constraint on parameters declared non-null in the supertype.

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 C#, C_o

Through a 2005 extension to the .NET framework, C# [64] saw the introduction of nullable types, which essentially allows primitive types to be made nullable. Such an enhancement is the first in a series of extensions to the .NET framework which allows better native support for database and XML queries and transformations. The latest framework extension of this nature is referred to as LINQ [21] and is an integral part of the C# language extension named C_o [3].

Nullable types allow a better conceptual match with databases in which all table fields, even of primitive types, could be null. In C# and C_o a primitive type name adorned with “?” denotes the nullable variant of the base type. Of course, an unadorned primitive type continues to represent the non-null variant of the base type. Since reference types are nullable by default in these languages, C# disallows the use of “?” with reference types.

4.3.3 Spec#

Spec# is an extension of the C# programming language that adds support for contracts, checked exceptions and non-null (reference) 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 [15]. 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 [28]. 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 de-

note possibly null references to T . Note, however, that 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.

4.3.4 Nice

Nice is a 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 [5, 6]. 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 possibility that a declaration of type T might be null, one prefixes the type name with a question mark, $?T$ [7].

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 [41]. In order to increase the accuracy of error reporting related to nullity and to better 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 [42].

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 and assignments of null to non-null declarations 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 [48].

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` [44]. 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 differs from 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 [18, 40]. 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 [19]. The implemented type system is essentially that of Fähndrich and Leino [29]. 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. To provide support for its non-null type system, JastAdd was extended with the ability to infer type nullity and rawness for APIs and other legacy code. After analyzing 100 KLOC from the Java 1.4 API, 24% of reference return types were inferred to be non-null and 70% of dereferences were found to be safe [19].

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 [49]. It is a behavioral interface specification language that, in particular, brings support for Design by Contract (DBC) to Java [50]. 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 documentation and test generation, as well as runtime assertion checking, extended static checking and full static program verification [8]. JML has nullity modifiers (`non_null` and `nullable`) and it recently adopted a non-null-by-default semantics for reference type declarations [10].

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 [9], with
- non-null as the default.

Even well established languages like Eiffel are making the bold move of switching to the new default [53]. 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 adopting 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*:

- Null references generally require 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 with implementation of non-null types and non-null by default in JML tools [9], 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 JML4 for short). Since 3.2, the Eclipse JDT Core has supported intra-procedural flow analysis for potential null problems. The Eclipse 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 [59] efforts are finalized, adhering to JML-like annotations seemed advantageous since it would allow the Eclipse JML JDT to:

- support nullity annotations for all versions of Java, not just Java 5 and later (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 [22] 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.

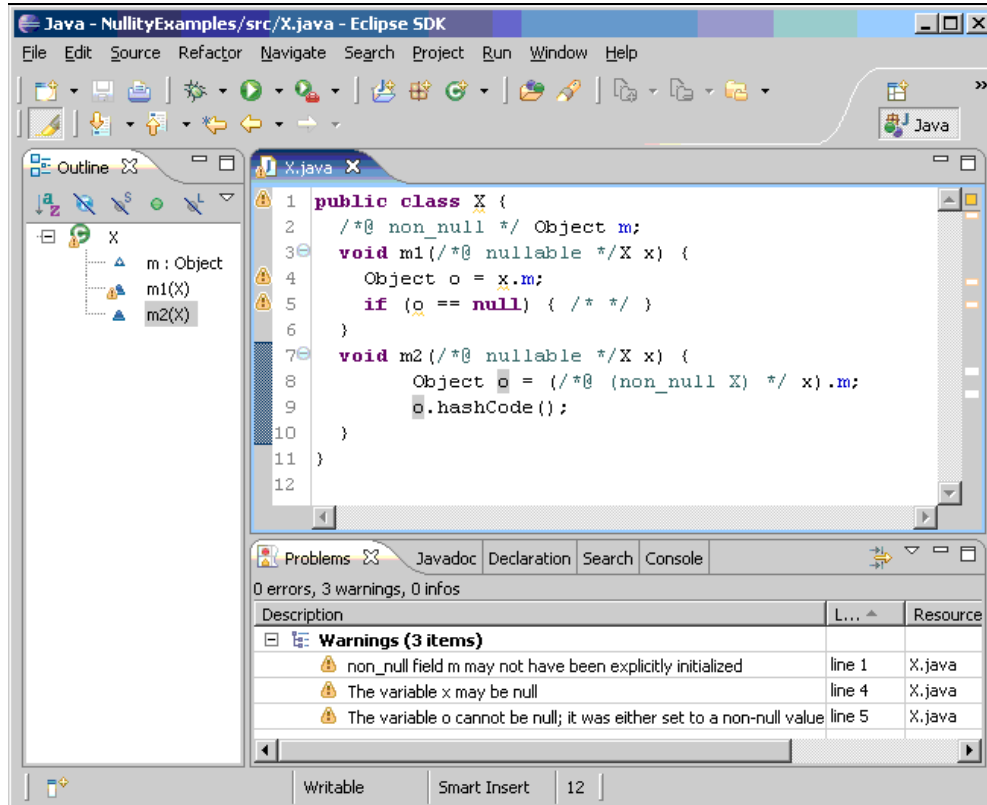


Figure 5. Screenshot of the Eclipse JML JDT (JML4)

Using JML syntax also means that the source files will be more easily amenable to processing by the complementary suite of JML tools (see Section 4.4.4). Once the standard (Java 5) annotations of JSR 305 are adopted, it will be rather easy to adapt the tool to process these annotations as well.

A summary of the Eclipse JML JDT Core capabilities is given at the bottom of Table 3. In particular we note that it supports edit-time, compile-time and runtime checking of nullity annotations—among several other features useful for Java program verification [12]. Figure 5 shows a screenshot of the tool in which it reports 3 violations of the non-null type system for a simple class.

5.2 Backwards Compatibility, and Migration to the New Default

The Eclipse 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 `nonnull_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 `nonnull` 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 intentions. 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 [20]: the JastAdd Nullness Inferer [19], Houdini [31] and the more recent CANAPA [13]. 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). However, the accuracy of its non-null annotations is reported at 79% [31].

Our study results have shown 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 Usage of Null

In this section we describe a second phase of analysis that was performed on the subjects of our case study. One of our goals was to see if we could further reduce the use of nullable types. While our initial investigation allowed us to conclude that 75% of declarations of reference types are non-null, a natural follow-up question was: Can the remaining uses of nullable be categorized and if so, which more semantically meaningful categories exist aside from nullable? In the subsections that follow we share our insights on the uses of nullable for fields, method return types and method parameters.

6.1 Fields

One of the first categories that we noticed, mainly because of its prevalence, was that some fields are not initialized during construction, but later. Once these fields are set to a non-null value, they remain non-null. This happens, for example, when fields are lazily initialized as is the case for `theUniqueInstance` field of the Singleton pattern [36]. We call this group *monotonic non-null* since their non-null status is monotonic.

Another use of nullable references is at the end of an object’s life cycle. References to “large” objects (or objects that refer to a large number of other objects, such as a tree) are often declared nullable so that the reference can be set to null when the object is no longer needed. This is necessary in Java to reduce memory usage and avoid what is often referred to as “memory leaks” in Java [4, Item 5].

Generally though, the use of nullable fell into two groups, those that are monotonic non-null (i.e., those that are non-null once initialized, but that are not initialized until after object construction) and those that freely move between holding null and non-null values during their lifetimes. The latter are similar to ML’s `option` type [58], which is defined as

```
datatype 'a option = None | Some 'a
```

It is interesting to note that the Nice language used the keyword `option` instead of `nullable` (see Section 4.3.4).

6.2 Methods

Methods return null either because it is a valid return value or to indicate that a valid object could not be returned for a number of reasons:

- initialization of an object has not been completed, hence some of its methods return null,
- an error occurred,

- no value corresponding to the parameters was found, e.g., in a database query or search of a data structure, or
- the end of a recursive data structure was reached, e.g., in a linked list.

All of these can be considered as cases of methods having an Option type. In the first two cases, methods could throw an exception instead of returning null. In other cases, the *Null Object* pattern [33] could be applied.

6.3 Parameters

Method parameters were the only category of reference in which it was not possible to determine a meaningful refinement in the use of nullable types. That is, in all cases a null value either was a valid value for a setter method (for the purpose of resetting a nullable field) or indicated a “don’t care” or “not applicable” parameter. In the latter case, overloading the method to only require parameters of interest would eliminate the need for a nullable parameter.

6.4 Statistics

Most of the time, nullable references can be avoided using the Null Object pattern, Java’s exception mechanism or overloaded functions. Nonetheless, our empirical study shows that, on average, developers would use them for approximately one reference out of four.

The most interesting usage of null that we identified was the monotonic non-null use for fields. Nullable fields are best avoided because it is not possible, in general, to reason statically about their nullity status in the context of multithreading. The problem is illustrated in Figure 6: while a nullable field can be tested for null in one thread, another thread can change a field’s value between a test against null and a dereference. A Java programming idiom that allows one to safely test and then use the non-null value of a field is illustrated in Figure 7. This idiom is so common that a special syntax was introduced in Eiffel to address it; an Eiffel program fragment equivalent to the one of Figure 7 is given in Figure 8. Notice how the syntactic shorthand allows the local variable declaration to be embedded inside the `if`’s condition.

It is in the light of these examples that one can see the benefit of declaring fields as monotonic non-null; i.e., for monotonic non-null fields, the test in Figure 6 would be sufficient to guard against potential null dereferences in the `then` part of the `if` statement. In fact, from the point of view of flow analysis, monotonic non-null fields can be treated like local variables, for which the nullity status can always be determined. In reviewing our case study subjects a second time, we noted that approximately 60% of nullable fields were monotonic non-null. Details, per project of our study, are shown in Table 4.

```

if (this.f != null) {
    this.f.g ... // possible NPE; field can be changed by another thread
}

```

Figure 6. Testing of a field against null is useless in multithreaded programs

```

F f0;
if ((f0 = this.f) != null) {
    f0.g ... // safe
}

```

Figure 7. An idiom for testing fields against null (that is thread safe)

```

if {f0: F} Current.f then
    f0.g ... // safe
end

```

Figure 8. An “object test” in Eiffel

The concept of *monotonic non-null* applies not only to fields but also to method return values to a certain extent. An obvious group of methods that return a monotonic non-null value is the getter methods for monotonic non-null fields. This is just one case of pure methods that may initially return null but after sufficient initialization always return a non-null value.

7 Monotonic Non-null

In the previous section, we introduced a group of nullable references we called *monotonic non-null*. This concept has been implemented, through the use of the `eventually_non_null` keyword in the Eclipse JML JDT [11, 12]. Field declarations annotated with this modifier are not guaranteed to be initialized to a non-null value by their declaring class’s constructors, but like `nonnull` fields, they are not allowed to be assigned a nullable value. Monotonic non-null references behave like non-null references once they have been assigned to. Because of this, we are able to treat them as non-null after a simple test of non-null.

While monotonic non-null fields share some similarities with non-null fields of a Raw or existentially Delayed object, as described by Fähndrich *et al.* [29, 30], they are also a

Table 4. Proportion of nullable fields that are monotonic non-null

	JML Checker	ESC/ Java2	SoenEA	Koa TS	Eclipse JDT Core	Sum or Average
Monotonic non-null	11	14	9	9	15	58
Option type	3	13	4	12	7	39
Total	14	27	13	21	22	97
% Monotonic non-null	78.6%	51.2%	69.2%	42.9%	68.2%	59.8%

```

/*#@ eventual | y_non_null */ Object f;
/*@ ensures \result == this.f; */
/*@ pure */ /*#@ eventual | y_non_null */ Object m() {
    return this.f;
}

```

Figure 9. A monotonic non-null method

```

/*@ nullable */ Object f;
/*@ constraint \old(f) != null ==> f != null;

/*@ constraint \old(m()) != null ==> m() != null;

/*@ ensures \result == this.f;
/*@ pure nullable */ Object m(){
    return this.f;
}

```

Figure 10. Same code desugared to standard JML

more general and flexible concept. Like our monotonic non-null fields, non-null fields of an instance of a raw or delayed type can be null until they are assigned to but, in contrast, an instance of a raw or delayed type must become “cooked” before the end of the instance constructor body is reached. An instance is cooked if *all* of its non-null fields have been assigned *non-null* values. Hence, a main difference is that Raw or Delayed is a type qualifier that applies to a class type and it influences the semantics of all non-null fields of that type. Monotonic non-null is a per-field qualifier. Thus, monotonic non-null fields can be null beyond the end of an object’s constructor’s body, and if the object has more than one monotonic non-null field then the nullity status of each can change independently. While in the general setting of multithreaded programs this construct can help to provide safety from `NullPointerException`, it does not eliminate the possibility of race conditions.

In addition to field declarations, methods return types can also be annotated as `eventual | y_non_null`. Once such a method returns a non-null value, it is guaranteed to never again return `null`. As we mentioned earlier, getter methods for `eventual | y_non_null` fields are obvious candidates; Figure 9 shows such a method⁷. Monotonic non-null methods can be desugared using, in particular, JML’s constraint clause as shown in Figure 10. A constraint clause, also called a history constraint, expresses properties that must hold between any visible state (whose values are captured via the `\old()` operator) and all visible states that follow it [51, §8.3]. Since `eventual | y_non_null` is a type modifier, desugaring it into a constraint on the field is not strong enough, i.e., it is only an approximation of its true meaning.

Parameters are better behaved than fields in that their initial values are fixed at the point of call and flow analysis can track their nullity status within the method body. As a result, monotonic non-null is not a useful modifier for formal parameters. The type

⁷ The # before the @ in the first line indicates JML4-specific annotation.

parameters of generic types can also take nullity attributes, but we have not come across a case in which it would be useful to have these be `eventually_non_null` instead of `nullable`.

Arrays whose elements are marked as `nonnull` but whose declaration does not have an initializer are almost always meant to have `eventually_non_null` Elements. This is one possible solution to the problem of determining an ending point for their initialization [29].

We have added compile-time and runtime checking of `eventually_non_null` fields and array elements to the JML4 compiler [11]. Static checking is accomplished by simply disallowing an assignment of a value “not known to be non-null”⁸ to such a field. At runtime, a contract violation error is thrown when the right-hand side of an assignment to a field declared to be `eventually_non_null` evaluates to null. Checking that the value returned by a method is `eventually_non_null` is beyond the abilities of the type system, and would have to be performed, e.g., by extended static checking using the desugaring that was given earlier. Runtime checking would require keeping an extra Boolean field, initialized to false, that to indicate whether the method has returned non-null. When an `eventually_non_null` method terminates, if the value to be returned is null but the Boolean field indicates that a non-null value has already been returned then a contract violation error is thrown. If the value to be returned is non-null then the Boolean field is set to true.

8 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 approximately 75% 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—i.e., JML4—offers such facilities in the form of support for project-specific as well as fine-grained control over nullity defaults (via type-scoped annotations). Until standard (Java 5) nullity annotations are adopted via JSR 305, we have

⁸ Note that this is a distinct category from values known to be null.

designed the Eclipse 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.

Based on an analysis of the usage of nullable types, we discovered the prevalence of monotonic non-null: almost 60% of the nullable fields in our study were of this type. We demonstrated how the use of monotonic non-null types could be beneficial, particularly in the context of multithreaded programs. Monotonic non-null types have been implemented in the Eclipse JML JDT and are available through the use of the `eventually_non_null` type modifier.

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 would like to thank the anonymous reviewers for their helpful comments as well as Markku Sakkinen and Peter Grogono for exchanges on nullity and references to Mary 2. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT).

References

- [1] M. Barnett, R. DeLine, B. Jacobs, M. Fähndrich, K. R. M. Leino, W. Schulte, and H. Venter, “The Spec# Programming System: Challenges and Directions”. *International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, 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)*, Marseille, France, March 10-14, vol. 3362 of *LNCS*. Springer, 2004.
- [3] G. Bierman, E. Meijer, and W. Schulte, “The Essence of Data Access in Co”. *Proceedings of the European Conference Object-Oriented Programming (ECOOP)*, Glasgow, UK, 2005.
- [4] J. Bloch, *Effective Java Programming Language Guide*. Addison-Wesley, 2001.
- [5] D. Bonniot, “Using Kinds to Type Partially-Polymorphic Methods”, *Electronic Notes in Theoretical Computer Science*, 75:1-20, 2003.
- [6] D. Bonniot, “The Nice Programming Language”, <http://nice.sourceforge.net>, 2005.
- [7] D. Bonniot, “Type Safety in Nice: Why Programs Written in Nice Have Less Bugs”, 2005.

- [8] 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.
- [9] 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)*, Nantes, France, July, 2006.
- [10] 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, July-August, vol. 4609 of LNCS, pp. 227-247. Springer, 2007.
- [11] P. Chalin, P. R. James, and G. Karabotsos, “An Integrated Verification Environment for JML: Architecture and Early Results”. *Proceedings of the Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, Cavtat, Croatia, Sept. 3-4, pp. 47-53. ACM, 2007.
- [12] P. Chalin, P. R. James, and G. Karabotsos, “JML4: Towards an Industrial Grade IVE for Java and Next Generation Research Platform for JML”, Concordia University, Dependable Software Research Group Technical Report, 2008.
- [13] 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)*, Mannheim, Germany, 2006.
- [14] D. R. Cok and J. R. Kiniry, “ESC/Java2: Uniting ESC/Java and JML”. 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, pp. 108-128. Springer, 2004.
- [15] R. DeLine and K. R. M. Leino, “BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs”, Microsoft Research, Technical Report, 2005.
- [16] 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.
- [17] ECMA International, “Eiffel Analysis, Design and Programming Language”, ECMA-367. June 2005.
- [18] T. Ekman, “Extensible Compiler Construction”. Ph.D. thesis. CS Dept., Lund University, 2006.
- [19] T. Ekman and G. Hedin, “Pluggable non-null types for Java”, Dept. of CS, Lund University, Technical Report (unpublished), 2006.
- [20] 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.
- [21] M. Erik, B. Brian, and B. Gavin, “LINQ: Reconciling Object, Relations and XML in the .NET Framework”. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL, USA. ACM, 2006.
- [22] M. Ernst and D. Coward, “Annotations on Java Types”, JCP.org, JSR 308. October 17, 2006.
- [23] D. Evans, “Using Specifications to Check Source Code”, MIT, MIT/LCS/TR 628. June, 1994.
- [24] 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.
- [25] D. Evans, “Annotation-Assisted Lightweight Static Checking”. *First International Workshop on Automated Program Analysis, Testing and Verification*, February, 2000.

- [26] D. Evans, “Splint User Manual”, Secure Programming Group, University of Virginia. June 5, 2003.
- [27] D. Evans and D. Larochelle, “Improving security using extensible lightweight static analysis”, *IEEE Software*, 19(1):42-51, 2002.
- [28] 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.
- [29] 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.
- [30] M. Fähndrich and S. Xia, “Establishing Object Invariants with Delayed Types”. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, and Applications*, Montreal, 2007.
- [31] 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.
- [32] D. Flanagan, *Java in a Nutshell: A Desktop Quick Reference*. O'Reilly, 1996.
- [33] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [34] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- [35] J. E. Freund and R. E. Walphole, *Mathematical Statistics*. Prentice-Hall, 1980.
- [36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [37] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Addison-Wesley Professional, 2005.
- [38] D. Grossman, M. Hicks, T. Jim, and G. Morrisett, “Cyclone: a Type-safe Dialect of C”, *C/C++ Users Journal*, 23(1), 2005.
- [39] J. V. Guttag and J. J. Horning, *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
- [40] G. Hedin and E. Magnusson, “JastAdd—An Aspect-Oriented Compiler Construction System”, *Science of Computer Programming*, 47(1):37-58, 2003.
- [41] D. Hovemeyer and W. Pugh, “Finding Bugs is Easy”, *ACM SIGPLAN Notices*, 39(12):92-106, 2004.
- [42] 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.
- [43] INRIA, “Pointers in Caml”, in *Caml Documentation, Specific Guides*, <http://caml.inria.fr/resources/doc/>, 2006.
- [44] JetBrains, “Nullable How-To”, in *IntelliJ IDEA 5.x Developer Documentation*: JetBrains, 2006.
- [45] 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.
- [46] J. R. Kiniry, “ESC/Java2”, <http://secure.ucd.ie/products/opensource/ESCJava2>, 2005.
- [47] J. R. Kiniry, A. E. Morkan, F. Fairmichael, D. Cochran, P. Chalin, M. Oostdijk, and E. Hubbers, “The KOA Remote Voting System: A Summary of Work To-Date”. *Proceedings of the Symposium on Trustworthy Global Computing (TGC)*, Lucca, Italy, November 7-9, pp. 244-262, 2006.
- [48] K. Lea, “Nully”, <https://nully.dev.java.net>, 2005.

- [49] G. T. Leavens, “The Java Modeling Language (JML)”: <http://www.jmlspecs.org>, 2008.
- [50] G. T. Leavens and Y. Cheon, “Design by Contract with JML”, Draft paper, 2005.
- [51] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin, “JML Reference Manual”, <http://www.jmlspecs.org>, 2008.
- [52] B. Liskov, E. Moss, A. Snyder, R. Atkinson, J. C. Schaffert, T. Bloom, and R. Scheifler, *CLU Reference Manual*. Springer, New York, 1984.
- [53] B. Meyer, “Attached Types and Their Application to Three Open Problems of Object-Oriented Programming”. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Glasgow, UK, July 25-29, vol. 3586 of *LNCS*, pp. 1-32. Springer, 2005.
- [54] Microsoft Developer Network, “C Run-Time Library: SAL Annotations”, in *MSDN Library, Visual Studio 2008, Visual C++ Reference*, 2008.
- [55] J. Palsberg and M. I. Schwartzbach, “Type substitution for object-oriented programming”. *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, Ottawa, Canada. ACM, 1990.
- [56] J. Palsberg and M. I. Schwartzbach, “Static Typing for Object-Oriented Programming”, *Science of Computer Programming*, 23(1):19-53, 1994.
- [57] R. Park, “Software Size Measurement: A Framework for Counting Source Statements”, CMU, Software Engineering Institute, Pittsburgh CMU/SEI-92-TR-20, 1992.
- [58] L. C. Paulson, *ML for the Working Programmer*. Cambridge University Press, 1991.
- [59] W. Pugh, “Annotations for Software Defect Detection”, JCP.org, JSR 305, 2006.
- [60] W. Pugh, “How do you fix an obvious bug?” <http://findbugs.blogspot.com>, 2006.
- [61] M. Rain, “The Structure of the MARY/2 Compiler”, *Software: Practice and Experience*, 11(3):225-235, 1981.
- [62] 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.
- [63] R. Stallman, “Using the GNU Compiler Collection (GCC): GCC Version 4.1.0”, Free Software Foundation, 2005.
- [64] M. Williams, *Microsoft Visual C#.NET*. Microsoft Press, 2002.