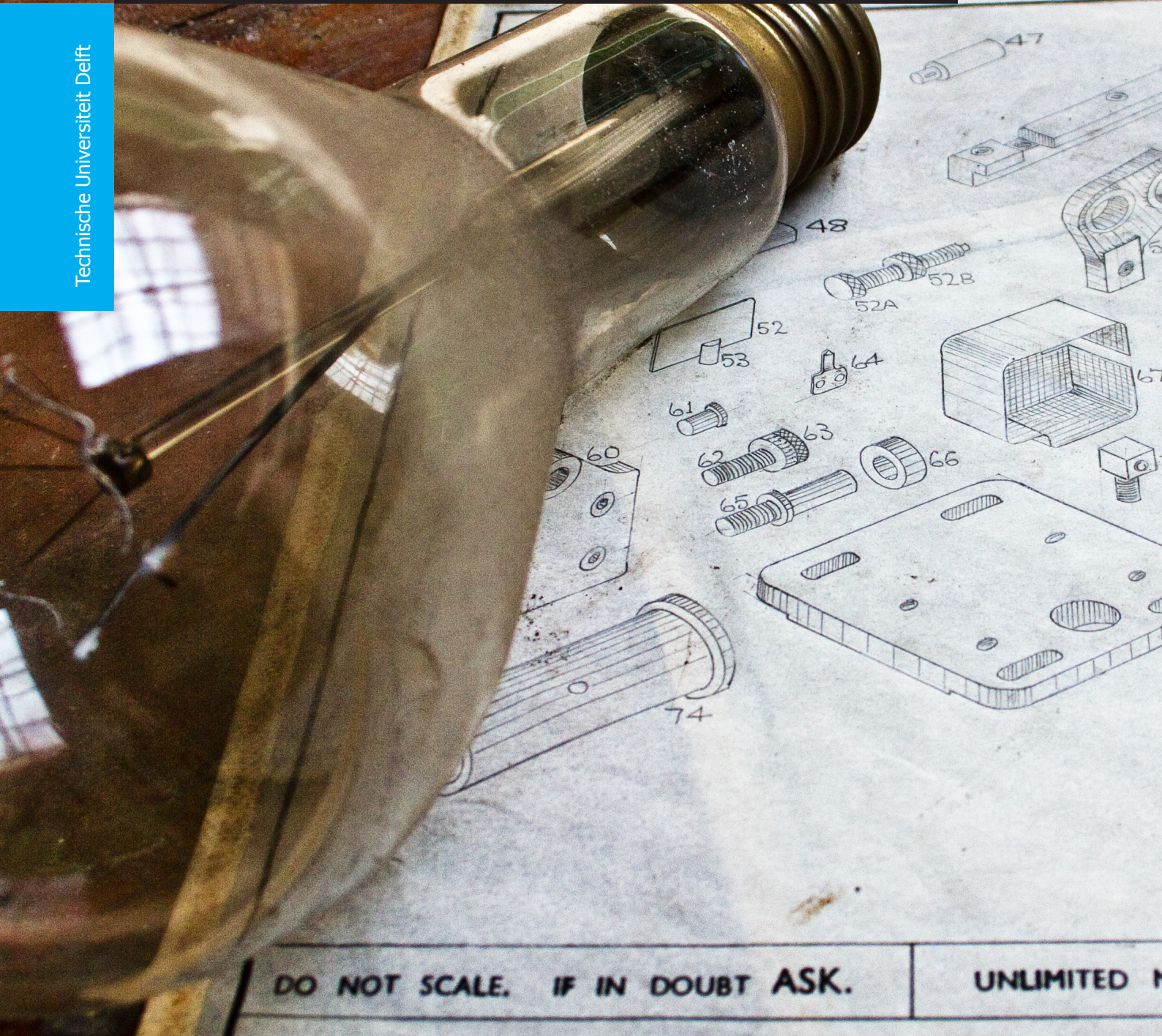


Investigating deprecation misuse a taxonomy, alternatives, controlled experiment, and experiment platform

Dereck J Bridie

Technische Universiteit Delft



Investigating deprecation misuse: a taxonomy, alternatives, controlled experiment, and experiment platform

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

by

Dereck J Bridié

to be defended publicly on Wednesday, January 8, 2019 at 17:00.



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2020 Dereck J Bridié.

Cover picture:

Tom Blackwell - "Hand-Drawn Machine Schematics". <https://flic.kr/p/9C2Xct>

Investigating deprecation misuse: a taxonomy, alternatives, controlled experiment, and experiment platform

Author: Dereck J Bridié
Student id: 4280830
Email: d.j.bridie@student.tudelft.nl

Abstract

Deprecation in Java is a language feature that allows API producers to mark program elements as obsolete. However, previous work has identified that this mechanism is co-opted to indicate another concept entirely: a misuse of deprecation. This leaves room for improvement because deprecation warning mechanisms do not fully describe the true reasoning with which API producers choose to misuse the deprecation mechanism. In this thesis, we create a taxonomy of five reasons why API producers misuse the deprecation functionality found in open-source software by analyzing 763 methods. Using this taxonomy, we create alternatives meant to help API producers avoid deprecation misuse by introducing five new annotations meant to be specific in API abnormalities. To test this proposed alternative, we conduct a user study. However, as no current experimental settings fit our needs, we create an experimental platform, RESPIRED, publishing it to improve the state-of-the-art in software engineering experiments. Finally, we test our alternatives using this platform, finding that changing only the warning text does not have a significant impact on developers.

Thesis Committee:

Chair, Supervisor: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Committee Member: Prof. Dr. A. Zaidman, Faculty EEMCS, TU Delft
Committee Member: Dr. C. Bach Poulsen, Faculty EEMCS, TU Delft
Committee Member: Dr. A. A. Sawant, UC Davis

Preface

Submitting and defending this thesis marks the end of the most formative period in my life. Since coming to Delft, I have been lucky to be able to surround myself with great friends with whom I'm proud to have spent my academic years with. We've laughed, cheered and cried, creating moments together I'll never be able to forget.

One particular moment is engraved in my soul for eternity. It was a night in June of 2016; a LAN gaming night in the Sporthal in TU Delft. An email had come, notifying me that grades had been published— the results for the final exam of Linear Algebra, a course I had struggled with, were available. I had barely failed the exam a multitude of times, and this was a deciding factor in my graduation. A small group huddled around my screen, and I didn't dare to look at the result— but an eruption of cheering and clapping propagated from this group across the hundreds seated compelled me to see the passing result. I'd like to thank Paul van der Knaap for the repeated study sessions, driving us to work hard to pass this course no matter how bleak it seemed. Not only for LA, but for Probability, Complex, and the final thesis endeavor. Paul, in the end, we made it— just like you always said we would.

It may seem strange to thank an entire section of a building, but the fourth floor of the new Computer Science building was filled with colleagues I've been lucky to meet and have many dinner sessions with. Thank you for the great food, support, and camaraderie during trying thesis times, and thank you for giving my dear (stuffed) animals a new loving home.

I'd also like to thank Anand for being my not only my supervisor, but also an encouraging friend, pushing me beyond the familiar and comfortable and guiding me with my work. Thank you, Alberto Bachelli, for your critical eye and constructive guidance during the beginning stages of the thesis. Thank you Arie van Deursen, for the confidence and guidance I needed to graduate. Thank you, Andy Zaidman and Casper Bach Poulsen, for agreeing to be on my thesis committee, grading my work, and being present at the defense.

Unfortunately, turning in this document is the last piece of work I will do for the TU Delft. Everything I have learned to love about The Netherlands and Delft I will have to leave behind to return to California, where a new adventure awaits. However, everything I've learned, I will bring with me. Thank you, Delft. I have been taught me an important lesson: family is the people who you surround yourself with, and not what you're born into.

Dereck J Bridié

Contents

Contents	5
List of Figures	7
1 Introduction	9
2 Related Work	13
2.1 Software reuse as an API	13
2.2 Deprecation as a Concept	13
2.3 Deprecation as a Language Feature	14
2.4 Deprecation Enhancements and Improvements	14
2.5 Deprecation’s Relation to API Consumers	15
2.6 Deprecation’s Relation to API Producers	16
3 Deprecation Misuse Cases	17
3.1 Dataset Creation	17
3.2 Usage Classification	18
3.3 Results	19
4 Deprecation Misuse Alternatives	21
4.1 Available Alternatives	21
4.2 Implementation of Misuse Alternative	22
4.3 JDK Augmentation	22
4.4 IDE Plugin	24
5 RESPIRED	27
5.1 Current State of Controlled Experiments in Software Engineering	27
5.2 Requirements	28
5.3 RESPIRED	29
5.4 Implementation	29
5.5 Features	30
5.6 Tool Distribution	34
5.7 Tool Evaluation	34
5.8 Requirements Revisited	35
6 Evaluation of Deprecation Alternative via a User Study	37
6.1 User Study Design	37

6.2	Results	46
7	Evaluation of RESPIRED	55
7.1	Value for Participants	55
7.2	Value for Researchers	55
7.3	Future Improvements	56
8	Discussion	57
8.1	Research Questions Revisited	57
8.2	Impact and Implications	59
8.3	Threats to Validity	59
9	Conclusions and Future Work	63
9.1	Contributions	63
9.2	Future Work	64
9.3	Conclusions	65
	Bibliography	67
A	Code fragments	73

List of Figures

1.1	Default deprecation warning output (OpenJDK 11.0.3)	9
1.2	Deprecation warning output with extra linting flag (OpenJDK 11.0.3)	10
1.3	Deprecation warning in IntelliJ IDEA 2019.1.3	10
3.1	Method we use to create a corpus of deprecation introductions.	18
4.1	Visual display of the warning on usages of elements annotated with the original <code>@Deprecated</code> annotation versus an alternative annotation.	25
5.1	The participant's view of the environment.	29
5.2	Systems overview of RESPIRED.	30
5.3	SurveyJS's supplied components for survey creation.	32
5.4	An example of creating a violin plot using Jupyter Notebook.	33
6.1	An example of the manner in which IntelliJ IDEA signals the introduced warnings by default.	45
6.2	Post-experiment choice tree for discovering participant sentiment.	46
6.3	Deprecation experiment overview	46
6.4	Diagrams of participant demographics	48
6.5	Task 1 deprecation flow diagram	49
6.6	Task 2 maintenance issues results	49
6.7	Distributions of time spent on each section of the experiment.	50
6.8	Participant groups avoiding the annotation in task 1 and task 2.	50
6.9	Post-survey questionnaire flow diagram	51
6.10	Flow diagram showing participant groups, definition click-through, and dissuasion	53
8.1	Taxonomy of deprecation misuse reasons.	57

Chapter 1

Introduction

Deprecation is a language feature that allows developers to mark certain functionalities as obsolete. Many valid reasons why to do so exist, including insufficient efficacy or indicating a feature that will be dropped in a future release. Oracle, the current owner of the official implementation of the Java platform, describes the following reasons for deprecating APIs in the Java Development Kit [44]:

1. The API is dangerous (for example, the `Thread.stop` method).
2. There is a simple rename (for example, AWT `Component.show/hide` replaced by `setVisible`).
3. A newer, better API can be used instead.
4. The deprecated API is going to be removed.

Deprecation is important because it gives library maintainers a method to notify library consumers about important changes in the API they supply. This information is also valuable to API consumers because it gives them time to create a migration strategy [44].

Various languages directly support the utility of the concept of deprecation, leading to the introduction of language features aimed at allowing developers to be able to mark code fragments as deprecated. For example, PHP has a special category of warnings meant for deprecation that can be emitted: `E_DEPRECATED` and `E_USER_DEPRECATED` [21]. These warnings shown at runtime and can be logged to help developers discover that they are using deprecated features [59].

In Java, deprecation is supported by the language as an optional annotation upon classes, methods, or member declarations using the `@Deprecated` fragment, or by using a `@deprecated` Javadoc comment [45]. The Java Language Specification [43] establishes rules that Java compilers must adhere to: in general, a compiler must produce a warning when a deprecated program element is overridden, invoked, or referenced by name. This often comes in the form of a text message, though the manner in which it is shown differs throughout platforms, Java compiler versions, and Java compiler command arguments. See Figure 1.1 and Figure 1.2 for an example of warning output under OpenJDK, an open source implementation of the Java Platform.

Figure 1.1 and Figure 1.2 show warning outputs when using `javac`, the command-line utility for compiling Java programs provided by the JDK. However, developers increasingly make use of an Integrated Development Environment (IDE), which assists developers in programming by offering syntax

```
1 $ javac MyProgram.java
2 Note: MyProgram.java uses or overrides a deprecated API.
3 Note: Recompile with -Xlint:deprecation for details.
```

Figure 1.1: Default deprecation warning output (OpenJDK 11.0.3)

```

1 $ javac -Xlint:deprecation MyProgram.java
2 MyProgram.java:10: warning: [deprecation] outdatedMethod() in MyProgram has
   been deprecated
3     new MyProgram().outdatedMethod();
4                               ^
5 1 warning

```

Figure 1.2: Deprecation warning output with extra linting flag (OpenJDK 11.0.3)

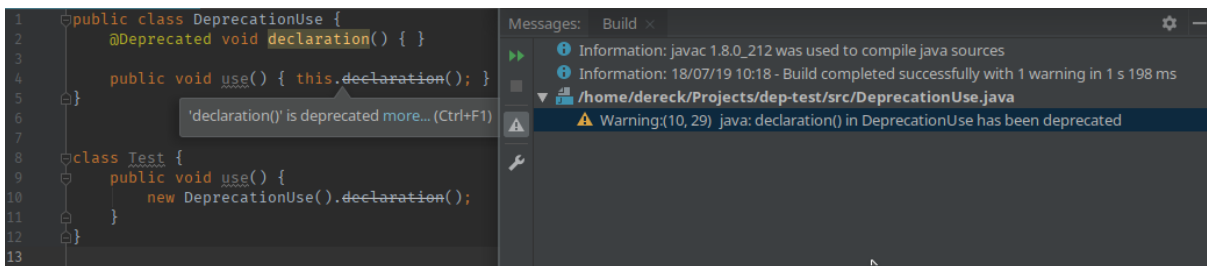


Figure 1.3: Deprecation warning in IntelliJ IDEA 2019.1.3

highlighting, content completion proposals, and indications of possible bugs [30]. Java IDEs also can show such a deprecation warning while the developer programs, amplifying the effectiveness of such a warning by displaying it along with the code that being warned upon [30]. Popular IDEs for Java can highlight usages of deprecated elements [22, 2]. For example, in IntelliJ IDEA, the declaration of a deprecated fragment is highlighted in yellow, and usages are drawn with a line through the text, as shown in Figure 1.3.

In open-source software, we find that the feature of deprecation is used for different purposes other than deprecation: a misuse of the language feature. Previous work by Sawant et al. [60] uses repository mining and issue tracker mining to find usages of deprecation, identifying legitimate usages and “misusages”: an unorthodox usage of deprecation for reasons the deprecation guidelines do not stipulate.

Sawant et al. [60] argue that inflexible communication methods between API producers and API consumer might lead to unmet communication needs, and propose future research be conducted in this area. In a follow-up work, Sawant et al. [59] elaborate on the direction such communication methods could take: introducing high-level warning mechanisms for the Java programming language.

This dissertation seeks to answer the following questions of interest. These questions guide our approach in this study.

Research Question 1: What motivations are there behind developers misusing Java’s deprecation mechanism?

To able to understand how developer’s needs can be met, we first need to understand what these needs are. Using this information, alternatives can be derived that strive to meet these needs.

Research Question 2: What are the possible ways in which misuse of the deprecation mechanism can be avoided or eliminated?

Once we have these reasons, we devise a mechanism which could replace the misuse of deprecation, giving API producers an option that more accurately captures their communication needs.

Research Question 3: How effective is the new mechanism we offer compared to the existing deprecation mechanism?

In the last part of this dissertation, we examine the properties of this new mechanism. In particular, we are interested in a comparison between the misuse of deprecation and our newly offered mechanism. The mechanism we propose should have a positive effect on API consumers, meaning that the intent of the communication was understood better than in the original case.

We start with an exact definition of misuse, a term that will guide the core of this thesis. We review the recommendations for the usage of the deprecation feature as given by Oracle [45]: In this thesis, we use the following definition of misuse, derived from the above guidelines:

A misuse of the deprecation feature is when an element is marked as deprecated when the element is NOT a transitional element, i.e. the element does not help an API consumer migrate from an existing API A to a new API B in the correct way.

This thesis explores this concept of deprecation misuse using our three research questions to guide our approach and research. **RQ1** is backed by an empirical analysis of deprecation usage in open-source projects, where we discover reasons why the deprecation mechanism is misused in actual projects.

Then, **RQ2** uses a study of existing mechanisms API producers have to communicate with API consumers to choose a new alternative to deprecation misuse. We implement this alternative and include IDE support in IntelliJ IDEA, allowing developers using our setup to see our newly devised warnings just as they would interact with any other type of warning.

Finally, we create and administer a user study to tackle **RQ3**. As we find that current solutions to conducting a user studies are dissatisfactory for our needs, and thus elect to create a new experimental platform that allows us to have complete control over the environment that a participant interacts with, creating a realistic setting for participants to develop in. We conduct a controlled experiment using this tool, requesting participants to complete various tasks related to various APIs.

Thus, the contributions of this dissertation include the following:

- A publication of a dataset of deprecation usages across 12,324 versions of open-source projects, resulting in 306,529 occurrences of deprecation method declarations (chapter 3);
- A categorization and taxonomy of misusages of the deprecation mechanism found in these open-source projects (chapter 3);
- An implementation of new annotations meant for replacing these misusages of the deprecation mechanism, which can be used in a Java Enhancement Proposal (chapter 4);
- An experiment platform, accessible over the internet for reproducible controlled experiments aimed at improving the state-of-the-art in software engineering quantitative studies (chapter 5);
- An evaluation as to the effectiveness of these new annotations, comparing their efficacy in dissuading users from its use (chapter 6).

We present the structure of this thesis, starting with chapter 2, which includes background information and related work, found by going through existing literature about deprecation and its relation to all parties involved. Then, chapter 3, we show the reasons that were found why API producers misuse the deprecation feature, along with presenting an alternative we offer and implement. In chapter 4, we describe the implementation of an alternative to deprecation misuse. Then, chapter 5 describes the online experiment platform that the user study runs on. Afterward, chapter 6 describes the user study design using the misuse alternative and the experimental platform and shows the results of this deprecation misuse user study. Then, we argue the value the experiment platform had for this study in chapter 7. In chapter 8 we take a critical view of the work that this thesis provides, discussing threats to validity. Finally, chapter 9 concludes this dissertation and provides a basis for future work.

Chapter 2

Related Work

To understand the nature of deprecation, we start with the foundations of software reuse and software evolution, laying the groundwork to understand the origins of deprecation. We examine deprecation support by programming languages, in particular, in Java, then continue to advances in improving and enhancing this support. Finally, we examine common practices between API consumers and API producers in relation to deprecation.

2.1 Software reuse as an API

Software reuse is a practice widely used in software development, allowing developers to make use of existing software components instead of recreating them each time. This reuse has many benefits, including less time spent on development, finding bugs, and a shorter time to market [13, 33, 29]. These benefits of software reuse have been recognized since the 1990s and remain a core tenet of software engineering today: entire software development communities revolve around the concept of software reuse [4, 63].

Software maintenance is an on-going effort to keep software up-to-date in face of an ever-changing environment [32]. One facet of these changes are related to the interactions between the boundaries of software systems. For this reason, APIs (Application Programming Interface) often maintain backwards compatibility by avoiding changes in external client-facing interfaces reduces software debt in API clients.

2.2 Deprecation as a Concept

However, backwards compatibility cannot be maintained at all times. Software evolution involves adding new features, changing existing features, or removing features. In particular, removing features is difficult for API users: removing a feature that a developer depends upon means that the consumer must change existing code to cope with these changes.

Deprecation To soften the immediate impact on these consumers, API producers tend to follow a deprecate-replace-remove cycle [51], releasing a version of the library in which these features are indicated to be deprecated. The concept is that a client using this version will become aware of this deprecation, and knows that this feature is slated to be removed in the future. This has multiple advantages: a client becomes aware of the future removal and has ample time to prepare for and consider the migration. If possible, messaging will suggest ways the client can approach the migration, offering alternative features or other migration paths. Finally, a version will be released in which the feature is removed.

2.3 Deprecation as a Language Feature

We investigate the history of Java’s deprecation feature in order to understand the evolution of Java’s importance in deprecation, and examine past decision-making and considerations when it comes to its implementations. Furthermore, we look at other languages to understand differences in the mechanism which can be taken into consideration when creating an alternative.

Javadoc Since the initial version of the Java Development Kit (JDK), Javadoc (at the time, capitalized as JavaDoc) has been shipped with the Java tool set [8]. Javadoc comments are a special type of comment that contain Javadoc tags. These tags are keywords that the Javadoc tool can process in order to generate human-readable documentation. In JDK version 1.1, the `@deprecated` tag was added to the Javadoc tool. Additional information may follow this tag, which is recommended to be information about a replacement API or a migration path [45].

JEP-175 Sun Microsystems, Inc. proposed a general-purpose metadata facility in JSR-175 [19], introducing the concept of annotations to the language. These annotations can be attached to definitions, giving the compiler user-defined information, and for processing at compile-time, deployment-time, and runtime processing. This feature was released in JavaSE 5.0, allowing developers to create annotations and annotate code using them. Among other annotations, the `@Deprecated` annotation was added alongside this mechanism, accompanied with compiler checks to warn on usage of deprecated fragments.

JEP-277 Since JEP-175, JEP-277 [9] has been accepted and integrated into the Java language, adding functionality to the existing `@Deprecated` annotation. In particular, users of the `@Deprecated` annotation are able to specify whether or not the fragment is marked for removal in the future. The version that the deprecation was introduced in can also be specified. The proposal of JEP-277 was motivated as follows:

Very few deprecated APIs were actually removed, leading some people to believe that nothing would ever be removed. On the other hand, other people believed that everything that was deprecated might eventually be removed, which was never the intent either. (Although it wasn’t stated explicitly in the specifications, various documents mentioned that deprecated APIs would be removed at some point.) This resulted in an unclear message being delivered to developers about the meaning of `@Deprecated`, and what, if anything, developers should do when they encountered usage of a deprecated API. Everybody was confused about what deprecation actually meant, and nobody took it seriously. This in turn has made it difficult ever to remove anything from the Java SE API.

These continual improvements in the Java Language with respect to Java show that the feature is valuable and that considerable time and development effort is being spent in improving deprecation in Java.

Deprecation in other languages Other languages also support deprecation via a language feature. C# uses a similar concept called `[ObsoleteAttribute]`, which can be attached to program fragments in a similar manner as Java’s annotations. However, in contrast to Java, developers are able to specify a message that is shown when the error is encountered. Furthermore, the fragment use can also be marked as an error, causing a build to fail when the fragment is used. This approach is more strict: where Java has mechanisms to disable warning output of deprecation use via `@SuppressWarnings`, C# offers the option to break builds. Rust also supplies such an annotation, `[#deprecated]`, allowing developers to specify a custom note and date of introduction.

2.4 Deprecation Enhancements and Improvements

As we are interested in making changes to the ecosystem of deprecation, we look at related work into improving deprecation and incorporating suggestions found in previous research.

Proposed Enhancements Sawant et al. [59] suggest three enhancements to the current deprecation mechanism, aiming to improve the communication between API producer and consumer. The first is to keep consumers aware of the future of a deprecated feature by informing them of the planned timeline. Additionally, producers should be able to specify the severity of the deprecation, such that consumers are aware of how imperative it is that action is undertaken. Finally, a generic warning mechanism is proposed that should allow other types of warnings to be raised, thereby minimizing misuse of the deprecation annotation.

JDK-6447051 The concept of adding an alternative annotation to the JDK is not entirely novel: JDK-6447051 [1] is an issue on the OpenJDK bug system, created in 2006. The enhancement request suggests that a new `@UnsupportedOperation` annotation could be created to replace cases where `@Deprecated` is used. The example given is `Collection.add()` and related operations. As the root `Collection` contains mutable operations such as adding elements and clearing the collection, immutable collections often implement these collections by throwing an exception and marking these mutable operations with `@Deprecated`. The issue explains that IDEs could highlight these methods to allow for safer coding at compile time. However, no action was taken on this issue: a lack of interest to investigate and work on this matter lead to the issue being closed in 2019.

2.5 Deprecation’s Relation to API Consumers

Deprecation has a large effect on consumers. We examine previous studies on deprecation and its effect on consumers to understand the effects of deprecation in relation to API consumers.

Foundations of deprecation impact Robbes et al. [53] investigated the effects caused by deprecation by analyzing a SmallTalk ecosystem. This work was foundational in deprecation research: case studies in API change impact in an actual ecosystem had not yet been conducted. The findings relevant to this dissertation are that many consumers were impacted by API deprecation, and that the extent of the consumers often went unnoticed by API developers. Furthermore, reactions to API changes remained undiscovered long after the introduction of the deprecation. Finally, it was found that consumers are not supplied with enough information concerning deprecation: in 40% of the cases, instructions were either absent, unclear, or the given advice was not heeded. In the case of this study, actual usage of deprecation is still fraught for consumers.

Deprecation impact in Java Sawant et al. [58] conducted a non-exact replication study of the aforementioned study, investigating reactions of more than 25,000 API consumers that made use of five different libraries in the Java ecosystem. It was found that deletion of a deprecated entity was the most common reaction to encountering deprecation, when any action was taken at all, despite API documentation intending to offer replacements for deprecated APIs. However, a vast majority do not react to deprecation, proposing two possible explanations for this phenomenon: developers do not see the importance of removing deprecated artifacts or that the benefit does not justify the effort of change. Few API clients update the API version that they use, silently increasing their technical debt by accumulating future API changes in a more current version.

Deprecation usage in the Java Standard Library Qiu et al. [50] examine 5000 open-source projects to understand how APIs supplied by the Java standard library are being used. They found that 49.5% of these projects use at least one deprecated API element from the Java standard library. Furthermore, the distribution of the Java version these methods were deprecated in is analyzed: surprisingly, methods deprecated in an old version of Java still have high usage rates.

Difficulties Linares-Vásquez et al. [34] conducted an analysis on 213,836 questions tagged with the “android” keyword in the popular questions and answers website StackOverflow [10]. They found that leaving old versions of methods in Android APIs lead to questions asked, causing doubts in developers such as in question 4904660 [5]. As Android leverages the Java ecosystem, this research shows that consideration is necessary as to the confusion any proposed alternative would bring to developers.

2.6 Deprecation’s Relation to API Producers

When examining alternatives to deprecation misuse, we should understand current practices in deprecation usage by API producers. Both correct usages of deprecation and misuses of deprecation should be considered, so that the current communication methods can be examined in detail.

Deprecation usage in practice Raemaekers et al. [51] examine deprecation best practices compared to usage in open-source libraries distributed through Maven Central. Out of 22,205 artifacts examined, only 5.4% contained a single usage of `@Deprecated` at all. As an indicator of breaking changes, `@Deprecated` is not often used: they find that 33% percent of sampled public methods are deleted without a deprecated tag.

Deprecation cycle In the interest of backward compatibility, API producers can choose to follow a deprecate-replace-remove cycle [20], giving consumers ample time before the feature is removed. However, a study by Zhou and Walker [65] shows that this cycle is not often followed: in practice, many different routines were observed, including resurrection of removed features and features being removed without prior deprecation warning. Raemaekers et al. [51] recommend a new cycle: deprecate-replace-hide-remove, adding a new step where the methods are removed from the developer’s point of view before being completely removed. This is used in the Android framework code base successfully [51].

Impact of breaking changes Xavier et al. [64] investigate breaking changes, excluding changes that are made with prior `@Deprecated` notification. They found that 62.46% of sampled libraries contain at least one breaking change. The impact on the clients, however, is relatively low: only 2.54% of clients are impacted on the median. This is conjectured to be caused by developers paying attention to usage before breaking contracts.

Facilitating adaptation Consumers that use a deprecated API are encouraged to find an alternative. However, Brito et al. [15] show that finding such an alternative is not often eased by the API producer: within 661 Java systems, 64% API elements are deprecated with replacement messages per system, leaving the remainder of elements without replacement usage documentation. They also find that no major effort to improve these messages is made over time.

Reasons for deprecation Sawant et al. [60] give a solid foundation to this work, investigating reasons behind feature deprecation. They categorized 12 high-level reasons for introducing deprecation, of which 10 are “orthodox”, i.e. in line with the common understanding of deprecation, and 2 are not. These two mark an incomplete implementation of an interface and to mark a temporary feature which could be removed in the future. We use this work as a foundation for this dissertation, expanding upon these reasons for introducing deprecation in the context of misuse.

Chapter 3

Deprecation Misuse Cases

This chapter seeks to answer the first research question: what reasons do API producers have for misusing the deprecation mechanism in Java? First, we start by elaborating on the dataset we use to find usages of deprecation in APIs. Then, we describe the manner in which the reasons are found and categorized. Finally, we present the results of this categorisation, listing the reasons that we found why API producers misuse the deprecation mechanism.

3.1 Dataset Creation

In order to find the reasons why the deprecation mechanism is misused in Java, we need a dataset of deprecation usage. We look at open-source projects to create a corpus of Java source code, finding deprecation usages within this corpus. Finally, we narrow down this dataset to find usages of the deprecation mechanism. This happens in the following manner:

1. We use a dataset for API usage by Sawant and Bacchelli [57] as our dataset of projects to analyze. This dataset contains Java projects located on GitHub that use Maven as their build automation tool, using dependency information of ca. 42,000 projects to build a list of most popular APIs based on how many GitHub projects depend on them. This gives us a view of which APIs are likely to be consumed in software projects.
2. We use this dataset to derive a list of projects of which we will find usages of the deprecation feature. We select an arbitrary number of projects from this dataset: APIs which see at least 2000 dependant projects, resulting in 231 API producers.
3. From these API producers, we retrieve all available source code archives of each project from Maven Central. When such a version is not provided by Maven Central, it is excluded from our considerations. We end with 12,324 versions of projects.
4. We use SPOON [46], a library that facilitates processing Java source code and parsing it into an Abstract Syntax Tree. We analyse this tree to find usages of the deprecation feature. We include methods that are annotated with `@Deprecated`, `java.lang.Deprecated`, use the `@deprecated` Javadoc tag, or mention the text `deprecated` in its comments.
5. For each method that we find, we record contextual information about this method, including the declaration, implementation body, and documentation attached to the element. Table 3.1 shows an example of the data that is recorded for each deprecation usage. This dataset has been uploaded as an artifact [14], providing a starting point for other research in the field of deprecation.
6. We consider only the first occurrence of each fully qualified name, in chronological order by artifact publication date. A method marked as deprecated may remain for multiple versions. Our intuition says that the reason for deprecation does not change after introduction of the deprecated element.

Artifact	mysql:mysql-connector-java
Version	5.1.33
FQN	com.mysql.jdbc.ReplicationConnection.getServerCharacterEncoding
Declaration	return getServerCharset();
Javadoc	@deprecated replaced by <code>getServerCharset()</code>
Annotations	@java.lang.Deprecated
Artifact	com.google.android:android
Version	4.1.1.4
FQN	android.telephony.gsm.SmsMessage.getEmailFrom
Declaration	throw new java.lang.RuntimeException("Stub!");
Javadoc	<i>None</i>
Annotations	@java.lang.Deprecated

Table 3.1: Example rows in deprecation dataset.

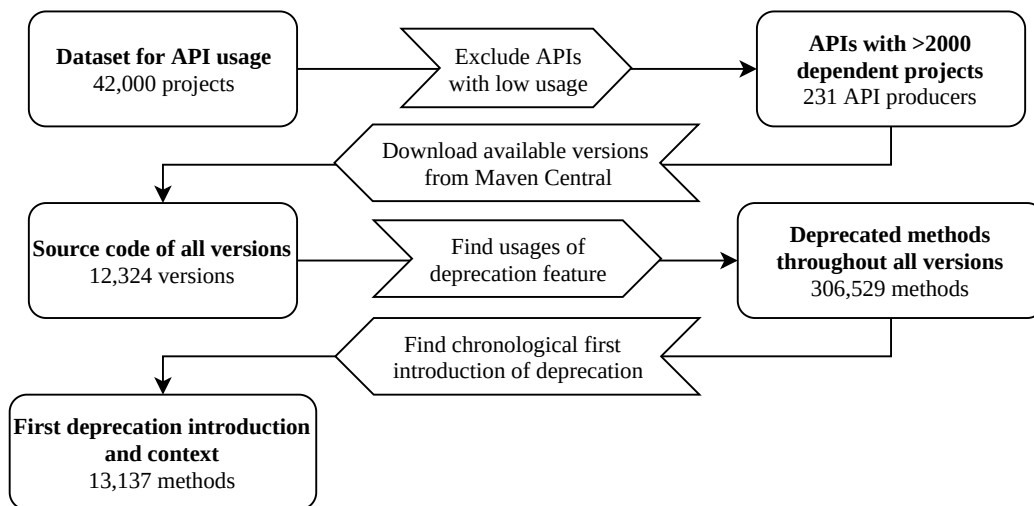


Figure 3.1: Method we use to create a corpus of deprecation introductions.

3.2 Usage Classification

Now that a dataset for deprecated methods has been made, we create a manual classification of reasons why API producers deprecate features. In this classification, we examine both correct usages and misuses, following the formal definition of deprecation as given in chapter 1. We use an exploratory approach to identify cases of misuse, then use work by Sawant et al. [60] to validate the reasons we found.

Categorizing the reason why deprecation was introduced for each of the 13,137 deprecated methods in the dataset is a considerable amount of work, so we use a sampling strategy to reduce the categorization work. We understand and are critical of the bias introduced by sampling: these considerations are explored in subsection 8.3.1. We sampled the projects in the following manner:

1. We group APIs under the management of the Spring Framework¹ together, assuming that this coding community will abide by the same deprecation guidelines.

¹<https://spring.io/>

2. From the top 20 projects with the most usages, we take five to analyze, categorizing 100 deprecation usages each. These five are randomly chosen with equal probability. From the remainder, we categorize 2 methods each (if there are less than two methods, we use all of the methods).
3. These methods are selected using a stratified random sampling method, whereby the strata are formed based on project version, that is, we attempt to diversify the versions per project we examine.

This manner of sampling leads to a total of 763² methods for consideration for manual analysis.

Finally, we conduct the manual analysis. From the sampled deprecation usage dataset, we examine each method call sequentially. We consider all data that was record for this method, and attempt to judge the reason why deprecation was introduced based on the method’s declaration, documentation, and implementation. The author of this thesis conducted this analysis, being conservative in coding a misuse: for cases in which not enough information was available to make a judgement or reasonable doubt existed, intended use of deprecation was assumed. This resulted in six categories, five of which representing misuse of the deprecation mechanism. When this was completed, the found misuse categories were discussed with the supervisors of this dissertation, in which each method declaration within a misuse category was critically observed.

3.3 Results

We present the results of the manual classification of the reasons why API producers use the deprecation mechanism in Java. From the sampled subset, we identify six categories of deprecation usages, listed in Table 3.2.

Category	Description of Misuse	Occurrences
IU	Intended Usage of the deprecation annotation.	743
BETA	This element is subject to possible changes.	8
UNSUPP	This method will always throw an <code>UnsupportedOperationException</code> .	8
DONTCALL	This element should never be called under any circumstances.	2
TEMP	This element was not intended for long-term use.	1
INTERNAL	This element should never be used by an API consumer.	1

Table 3.2: Identified reasons for deprecation usage.

We elaborate upon observed cases below:

- **IU**: In this case, an existing feature is planned to be removed in the future for a reason that is congruent with deprecation guidelines.
- **BETA**: Misuse of deprecation to indicate a method in beta stages. The specifics of the method are not completely finalized, and are subject to changes when the API leaves its beta state. This category does not supercede an existing API.

²Attentive readers may notice that no odd numbers were mentioned, making it difficult to end up with an odd number using a sum of even numbers. However, projects may have only a single deprecated declaration, leading to only a single selectable method from a given stratum.

- **UNSUPP**: We observe this misuse in classes that choose to implement an interface partially. A classical example is related to immutable collections, as in the case of `ImmutableList`³, where modifying such a collection will cause an exception at runtime⁴.
- **DONTCALL**: A method annotated in this category is meant to never be called under any circumstances. An example of such a definition can be found in Listing 3.1.
- **TEMP**: These elements are misuses of deprecation when the element does not supercede an existing API or help consumers migrate between APIs. Such a method was added to the API in the understanding that it was not intended for long-term use.
- **INTERNAL**: Misusage of deprecation feature to indicate that a method should only be used within the API producer internally, as opposed to being available for use by the API consumer.

Listing 3.1: **DONTCALL** example in `org.hamcrest:hamcrest`⁵.

```

1 /**
2  * This method simply acts a friendly reminder not to implement Matcher
3  * directly and
4  * instead extend BaseMatcher. It's easy to ignore JavaDoc, but a bit harder
5  * to ignore
6  * compile errors.
7  *
8  * @see Matcher for reasons why.
9  * @see BaseMatcher
10 * @deprecated to make
11 */
12 @Deprecated
13 void _dont_implement_Matcher___instead_extend_BaseMatcher_();

```

³<https://github.com/google/guava/blob/ad529ca5425cb17bd9e26a7f7fc06e80bef0d692/guava/src/com/google/common/collect/ImmutableList.java#L521>

⁴In Java, the top-level interface for `Collection` has mutable methods, therefore all list implementations must provide mutable methods as mandated by the contract of `Collection`.

⁵<https://github.com/hamcrest/JavaHamcrest/blob/0da90880b3aa7364afba05b7f1a853a25c190a44/hamcrest/src/main/java/org/hamcrest/Matcher.java>

Chapter 4

Deprecation Misuse Alternatives

Now that the reasons for misuse of the deprecation mechanism in Java have been identified, we continue work on the second research question: in what ways can we help producers avoid or eliminate misuse of the deprecation mechanism? We start with a study of existing mechanisms used by API producers to communicate to clients in source code. Then, by examining advantages and disadvantages of each method, we select one that we will use in this dissertation. We give an implementation of this method, ensuring that it can be used by Java developers such that it can be evaluated against deprecation misuse.

4.1 Available Alternatives

Now that the misuse categories have been found, we devise alternative solutions that support API producers in communicating their needs concerning their APIs, where deprecation was used instead. In these cases, we identify that the intent with which these `@Deprecated` annotations were placed, taking as goal that any new alternative to the `@Deprecated` annotation should communicate an API producer's needs more clearly.

We examine existing solutions for messaging API consumers, weighing the advantages and disadvantages of each:

Add alternative annotations to JDK We take inspiration from the JDK and introduce new annotations to the JDK alongside the original `@Deprecated` annotation. This method will allow developers to use the new annotations as they are used to using `@Deprecated`, easing the migration towards the new annotations. Java developers would only need to update their Java version to receive these alternative annotations, a process that should be done when support is dropped for their current version regardless. However, an annotation has no functionality without a mechanism to read this added information: the compiler must be adjusted to warn for these new annotations appropriately.

Also, this approach would require submitting a JDK Enhancement Proposal (JEP) [11], a process that allows non-trivial changes to be introduced into the JDK. Getting a JEP approved is a multi-step involved process, requiring many layers of review before a change can reach normal developers. However, developers do not often update to the newest JDK version according to a survey conducted in October 2018 with more than 10,200 respondents: 79% continued to use version 8, even though 9 had been available for over a year [4]. Java 8 is a Long Term Support version, which could explain developers' complacency to remain on this version.

Create library with annotations We examine a library that solely provides annotations: JetBrains' `java-annotations`¹, which provides `@Nullable` and `@NotNull`, among others. These annotations allow an IDE to warn for potential developer error by adding constraints on parameters and fields: in this case, that the value may be `null` or `never null`, respectively. A similar approach can be taken in this

¹<https://github.com/JetBrains/java-annotations>

study by creating a library and IDE support for this library: however, API producers will need to include this library in their dependencies, and API consumers must use IDEs for which this support is created. Furthermore, such a plugin must then be made for each IDE in use, including support and maintenance.

Introduce Javadoc tag Our final proposed approach is to add a new Javadoc tag to the Javadoc specification. This new tag could be more specific in its messaging, able to inform users about the situation with a developer-provided message. This gives the API producer some freedom to motivate why the tag was placed or give the consumer a timeframe in which changes to this element will be made. However, this freedom is not currently used often in practice, in the case of the `@deprecated` Javadoc tag [15]. Additionally, Java binary files meant for distribution do not contain Javadoc information, as this information is removed during compilation, whereas annotation information remains.

4.2 Implementation of Misuse Alternative

In this dissertation, we limit our scope to choosing a single deprecation misuse alternative. We choose to take an ambitious approach and extend the Java Standard Library in the JDK with new annotations. We choose this because this allows API producers to annotate their APIs with a method that is idiomatic: annotations are an existing method of conveying information used by Java developers. Furthermore, inclusion in the JDK allows this work to be adapted and proposed as a Java Enhancement Proposal with minimal effort, allowing these annotations to be introduced into the language and become available to API producers. Should the work be rejected by the Java Language developers, it can still be adapted into an external library with hardly any effort.

4.3 JDK Augmentation

To add our annotations into the standard library, we need to extend a JDK that consumers use to build their libraries. We use OpenJDK as the basis for our implementation. OpenJDK is a free and open-source implementation of the JDK, allowing contributors to change the source code of the compiler and standard library, and is the most commonly used open source implementation available [4].

Creating an annotation is done with the `@interface` keyword, and is defined similarly to a Java interface. This means that an annotation can specify interface members, along with a default value. Listing 4.1 shows an example of how a new annotation can be created named `@ClassPreamble`. This also demonstrates optional and required interface members, signified by the `default` keyword.

Listing 4.1: Defining an annotation according to Oracle documentation².

```
1 @interface ClassPreamble {
2     String author();
3     String date();
4     int currentRevision() default 1;
5     String lastModified() default "N/A";
6     String lastModifiedBy() default "N/A";
7     // Note use of array
8     String[] reviewers();
9 }
```

We consider two methods of alternative communication: one generic annotation that uses a required interface member that justifies the use of the annotation, and a group of annotations each targeted for specific use for the misuse reasons found in section 3.3. We consider work by Sawant et al. [59] in which

²<https://docs.oracle.com/javase/tutorial/java/annotations/declaring.html>

two Java language contributors were interviewed, referred to as J1 and J2. J1 and J2 did not voice support for a generic warning mechanism, where J1 considered such a change to be too “extreme”. J2 felt that “[by attempting] to eliminate every type of misuse, were only going to open the opportunity for more types of misuse, and were going to make it harder for people who are going to use it in a sensible way or in an imaginative way” [59]. J2, however, did voice support for more specific warnings.

Guided by these Java Language contributors, we choose to create a new annotation for each of the misuse reasons found in section 3.3. First, we create an annotation for each of the misuse cases, resulting in five annotations as shown in Table 4.1.

Category	Annotation
BETA	@Beta
UNSUPP	@AlwaysThrowsException
DONTCALL	@NeverUse
TEMP	@Temporary
INTERNAL	@Internal

Table 4.1: Identified deprecation misuses against the proposed new annotation.

Then, we add these annotations to the standard library, using @Deprecated as the basis for the implementation. We adjust naming and documentation as to reflect the meaning of these annotations, but otherwise add no new functionality. Listing 4.2 shows an example of the adaptation, displaying the source code of the newly-created @Beta annotation.

Finally, we build the JDK, resulting in binaries that can be used to compile Java code files that make use of these new annotations. Any user of this newly-built JDK can use the alternative annotations as easily as they would use @Deprecated as a drop-in replacement.

Listing 4.2: The implementation of @Beta we add to the JDK.

```

1 package java.lang;
2
3 import java.lang.annotation.Documented;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 import static java.lang.annotation.ElementType.*;
9
10 /**
11  * A program element annotated {@code @Beta} is one that programmers are
12  * discouraged from using. This element is marked to be in beta, meaning
13  * that the method may be changed in future API versions.
14  */
15 @Documented
16 @Retention(RetentionPolicy.RUNTIME)
17 @Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, MODULE,
18             PARAMETER, TYPE})
19 public @interface Beta {}

```

4.4 IDE Plugin

Creating an annotation on its own has little use: this added information is easy to miss by an API consumer as the annotation has no functionality. No warnings will be shown when using this annotation, and can only be seen if a developer browses the definition of an element that is annotated. In the case of the `@Deprecated` annotation, signals in the IDE for deprecation are wholly implemented by the IDE.

However, IDEs have no unified platform for such behavior: for example, Eclipse uses a different platform than IntelliJ IDEA. Considering IDE usage statistics and a cursory exploration of the development effort involved, we choose to base our implementation on IntelliJ IDEA's platform. Although we are convinced that the choice of platform should not have a significant impact on the effectiveness of the alternative, we are aware that developers may be used to a different IDE than IntelliJ IDEA.

In IntelliJ IDEA, functionality added to the base platform is created in plugins. We create an IntelliJ IDEA plugin that has similar behavior to the existing functionality for signaling usages of fragments annotated with `@Deprecated`. However, this plugin shows a different message when signaling a usage of an alternative annotation. Table 4.2 lists the messages that we show for each annotation, where `@Deprecated` is the default message provided in the default settings of IntelliJ IDEA.

Annotation	IntelliJ IDEA reported message
<code>@Deprecated</code>	"O" is deprecated
<code>@Beta</code>	This method is not yet stable, and should be avoided where possible.
<code>@AlwaysThrowsException</code>	This method will always throw an exception.
<code>@NeverUse</code>	This method should never be used.
<code>@Temporary</code>	This method will be removed in the future.
<code>@Internal</code>	This method should not be called by API consumers.

Table 4.2: Annotations and message shown upon use

To create a plugin, we make use of the IntelliJ IDEA SDK. Our plugin analyzes Java files, creating an Abstract Syntax Tree representation of the code under analysis. We resolve every method usage under analysis, finding the declaration that this usage refers to. Then, we inspect all annotations that this declaration is annotated with: should one of our alternative annotations be found, then the inspection will show a warning on this usage. Figure 4.1a shows the signal shown by IntelliJ IDEA when hovering upon a usage of a fragment annotated with `@Deprecated`. Figure 4.1b shows an example of message a developer would see when encountering a usage of a custom annotation.

Though the SDK offers a multitude of problem highlight types (`ERROR`, `WARNING`, `INFORMATION`, etc.), we choose to mark this inspection with `LIKE_DEPRECATED`. This is to lower the number of variables in our evaluation: an `ERROR` problem highlight type using a red underline might motivate participants to avoid problematic fragments more effectively than when compared to a normal strikethrough. For this reason, we keep the situations the same, varying only the text that is shown upon such a usage.

Our hypothesis then, is that this implementation with a more targeted warning description is more effective at dissuading users from using this annotated element than a misuse of the deprecation feature. For example, a warning text informing a user that calling this method will always throw an exception could be more effective than a generic deprecation warning.

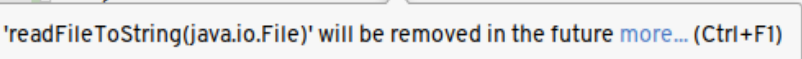
```
1902
1903 public static String temporaryUsage(File file) throws IOException {
1904     return readFileToString(file);
1905 }
1906
1907 * Reads the contents of a file into a byte array.
1908 The file is always closed.
```



'readFileToString(java.io.File)' is deprecated [more...](#) (Ctrl+F1)

(a) The signal shown by IntelliJ upon usage of a method annotated with the original annotation `@Deprecated`.

```
1903
1904 public static String temporaryUsage(File file) throws IOException {
1905     return readFileToString(file);
1906 }
1907
1908 * Reads the contents of a file into a byte array.
1909 The file is always closed.
```



'readFileToString(java.io.File)' will be removed in the future [more...](#) (Ctrl+F1)

(b) The signal shown by IntelliJ upon usage of a method annotated with the alternative annotation `@Temporary`.

Figure 4.1: Visual display of the warning on usages of elements annotated with the original `@Deprecated` annotation versus an alternative annotation.

Chapter 5

RESPIRED

To answer **RQ3**, aimed at evaluating the effectiveness between our deprecation misuse and our proposed alternative, we need a user study to evaluate differences in behaviors in each of the two groups. However, we find that current support for creating and conducting experiments like these with the requirements we have is inadequate. For this reason, we create a new experiment platform that suits our needs, and make it available for the software engineering research community as a whole.

This chapter details RESPIRED, a platform developed during this thesis to improve the state of the art in software engineering experiments. We start with a literature study in current methods in controlled experiments in software engineering, observing disadvantages in the approaches used in current practices and remarking what can still be improved. Afterward, we list our requirements. Then, we introduce the platform, detailing its features and showing how the platform improves current practices. Finally, we detail a security evaluation, user evaluation, and finish with how the platform can be used in other studies by other researchers.

5.1 Current State of Controlled Experiments in Software Engineering

In this section, we explain the motivation for the creation of the platform by examining recent literature into developer studies.

“Glacier: Transitive class immutability for Java” [17] uses a user study to test the effectiveness of a type annotation system for Java. Coblenz et al. use a controlled experiment as an evaluation strategy, soliciting participants and inviting them for the experiment in a physical location. In this way, they could offer the participant an environment already fitted with video recording software, the tasks to be done, and the development environment in which to work in. However, this leads to some room for improvement: as only a single computer was configured with the software, only one subject could be tested at a time. Furthermore, the participant must have also been able to physically be present for the experiment, making the pool of possible candidates smaller.

Similarly, a study by Uesbeck et al. [61] set up a room in which many computers were prepared for participants to work on. These computers were outfitted with virtualization software and a virtual machine containing the experimental setup. For participants who could not physically be present, this virtual machine image was transferred to the participant. Such images can be quite sizable; the most recent distribution of Ubuntu packaged by OSBoxes is 1.6GB¹. On top of this, the experiment must still be included in this size. Furthermore, the burden of installing virtualization software and getting the environment working correctly is put on the participant, complicating the process of taking part in the experiment.

¹<https://www.osboxes.org/ubuntu/#ubuntu-19-04-vmware>

Yet another example is “A Metric for Software Readability [16]”. Buse and Weimer [16] use an online survey tool to test the readability of software snippets. Although this allows participants to participate in the experiment at their own leisure, this does not model a real-world scenario: programmers often use IDEs when writing code, offering a wide suite of tools to aid program comprehension such as code highlighting and variable reference matching. Buse and Weimer [16] note this in their discussion: *“integrated development environments (IDEs) and specialized static analysis tools designed to aid in software inspections may constitute a better approach to the goal of enhancing program understanding.”*

5.2 Requirements

In chapter 4, we chose to create a custom JDK along with an IntelliJ IDEA plugin. This means that our experiment’s participants will, at the minimum, need to work with an installation of our JDK, IntelliJ IDEA, and our custom plugin. The length of an experiment is correlated with participant drop-out rate [25], so a lengthy setup time is detrimental to an experiment’s success. Though experiments often provide virtual machine images [61], these also require an installation process.

Furthermore, we also need to be able to observe how our participant interacts with the environment. At the minimum, we should be able to record participants’ answers to our programming tasks. However, it is also of interest to know what other features of the IDE are being used with respect to the warnings we display, e.g. is the warning being displayed at all. For these reasons, it is important that we can observe the participants’ actions, most preferably with a screen recording.

We are also interested in participant demographics and their sentiments, so we also need to be able to ask questions and receive answers in various ways. This means that we also need a survey mechanism that can ask participants questions and allow them to input their answers.

Finally, we identify common drawbacks in the setup of recent experiments in the literature, and attempt to create a experimental platform that aims to improve upon these flaws. In an ideal platform for creating empirical software development experiments, we look for the following qualities:

- **[REPRO]**: Replication studies are important to show the validity of an original study. In order to facilitate the replication of any study conducted with this tool, we aim to make it easy for a researcher to publish their experimental setup such that it can be run again.
- **[REUSE]**: In order to be able to offer this platform to the software engineering research community, we should make the platform generic enough such that other types of experiments can be conducted, e.g. supporting other languages or IDEs.
- **[FAM]**: We aim to make components within the environment familiar to the participant, to help mitigate threats of validity which are related to the experiment being conducted in a foreign environment.
- **[POWER]**: The environment should offer more than a current online survey platform does, adding the power of an IDE together with an online survey platform.
- **[REMOTE]**: Participants should be able to access the environment from across the world, using hardware that is familiar to them.
- **[DATA]**: The tool should facilitate the processing of experimental data, allowing researchers to easily handle their obtained results.
- **[SUPPORT]**: The tool should be aware of common user study patterns and offer support for studies using these patterns.

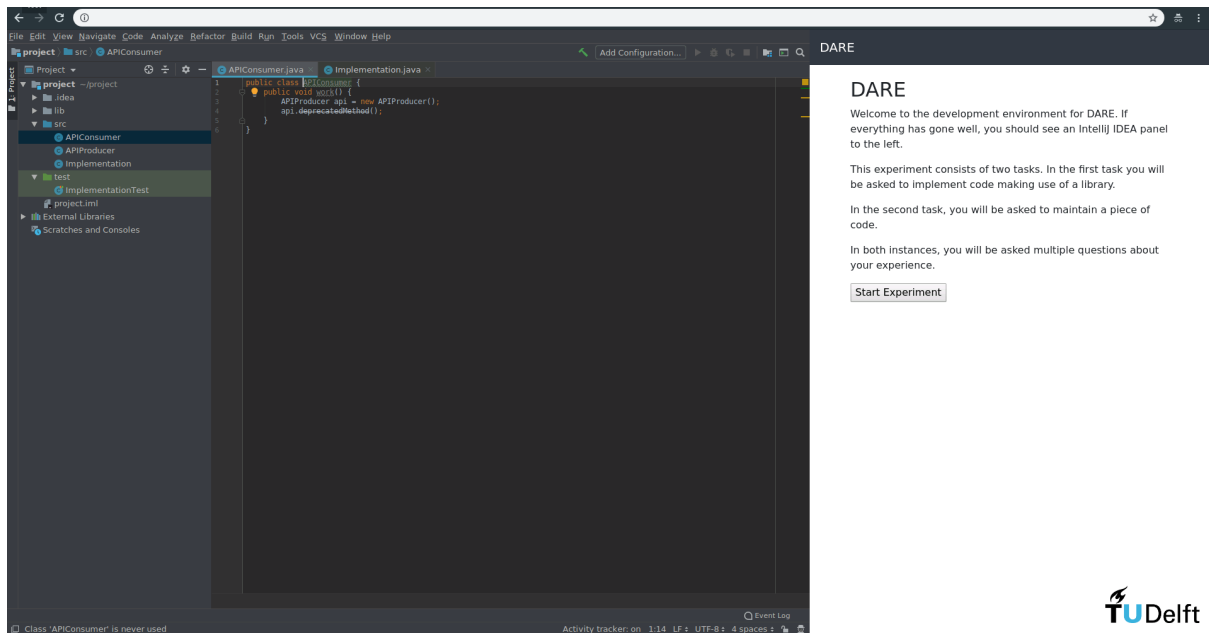


Figure 5.1: The participant’s view of the environment. Left: fully-featured IntelliJ IDEA environment. Right: Information and survey pane.

5.3 RESPIRED

During the course of this dissertation, we develop a web-based remote experimental environment. We aim to make this tool reusable and extensible so that researchers will be able to make use of work in this thesis in future experimental studies. This tool is named RESPIRED, a **R**emote **E**xtensible **S**urvey **P**latform using **I**DEs in **R**esearch for **E**mpirical **D**evelopment.

We take an ambitious approach to this platform, developing it not only for the purposes of this experiment, but with future work in mind. Though it has been used in this Java deprecation experiment, the components can be swapped out in order to create an environment for any arbitrary Linux-compatible controlled experiment [REUSE].

Participants are given an URL which gives them remote access to the experimental environment. The participant is then able to interact with the windows within the environment as if they were programs running on their own machine through their web browser [REMOTE]. Figure 5.1 shows an example of what a participant might see in their browser.

5.4 Implementation

The platform leverages Docker to create isolated reproducible system images. The main component is the orchestrator, responsible for booting, monitoring, and ending experimental instances. This component is also the user-facing system, showing the user the experiment information, asking for experimental informed consent, conducting the entry survey, and hosting the remote desktop interface.

See Figure 5.2 for a systems overview of RESPIRED. The tool uses Docker to create system containers. The containers provided by the tool have many utilities geared towards user studies, including screen recording, periodic snapshots of project file contents, an IDE, and the survey. Multiple images can be run at the same time, allowing multiple participants to undergo the same experiment at once.

The default experimental image provided by RESPIRED builds upon `ubuntu:18.042`, a minimal

²https://hub.docker.com/_/ubuntu/

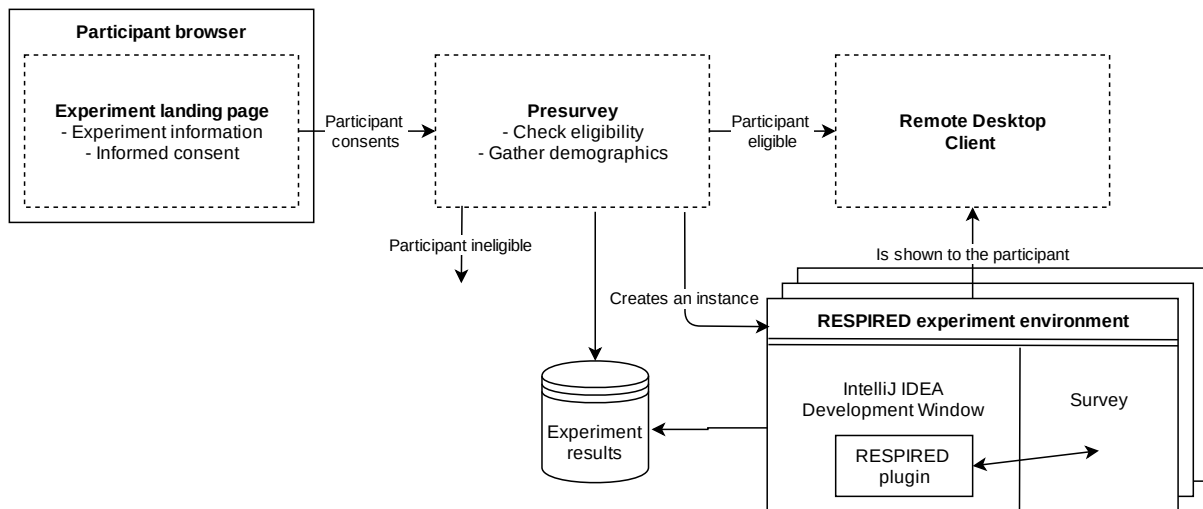


Figure 5.2: Systems overview of RESPIRED.

installation of the Ubuntu operating system in Docker. The modifications made to this image are as follows:

- `i3wm`³, a tiling window manager allows windows to be placed next to each other without borders and window controls. This ensures the IDE and browser window are placed side-by-side and cannot be modified;
- `chromium-browser`⁴, an open-source web browser that runs the survey interface. In particular, this browser supports a “kiosk” mode, removing administrative features such as the ability to open new windows and navigate away from the current page;
- we install IntelliJ IDEA Community Edition, acting as the IDE in which participants will work. We bundle an installation of the RESPIRED IntelliJ IDEA plugin, which disables features related to restarting the IDE and changing the current project. Furthermore, the plugin sets up a web service capable of answering queries about browser state and executing IDE commands;
- and finally, Virtual Network Computing (VNC) software, sharing an instance’s desktop view.

5.5 Features

This section is dedicated to elaborating upon the features RESPIRED provides, to show the value the tool provides. Any prospective researcher planning to make use of RESPIRED should be aware of the features the tool provides and also its limitations.

5.5.1 Setup

A researcher starts by creating an operating system image that describes the experimental environment. A skeleton is provided by the project, which is to be used as basis for the environment’s functionality. From this point, the researcher must configure the environment by creating the interactive survey, by changing the project that will be loaded in the IDE, and by writing any guidance meant for the participant. If the researcher is planning to conduct an experiment in a Java project, then many of the default components

³<https://i3wm.org/>

⁴<https://www.chromium.org/>

can remain unchanged. However, since the platform makes use of functionality shared between JetBrains IDE products, it is possible for the platform to run using another product such as CLion, though no work has been done to attempt it [FAM].

5.5.2 Experimental groups

The platform can assign each participant to an experimental group, allowing the environment to change based on which group a participant has been assigned to. Consider a controlled experiment in which participants are grouped into the control group or the experimental group. All components are aware of the group the active participant is in, allowing for experimental variables to change. For example, a researcher can choose to show different code files for the experimental group compared to the control group, or change questions asked in the survey panel [SUPPORT].

5.5.3 Interactive task programming

The survey panel allows researchers to place components that interact with the IDE. This is exposed by the RESPIRED plugin running in the IDE, and allows the survey window to query information within the IDE and change the IDE state. This allows two-way interaction between the IDE and the survey panel. For instance, a researcher might want to allow a participant to continue to the next part of the survey only when they have changed at least ten characters in a certain file. Alternatively, a researcher could place a button in the survey that runs a unit test and saves the outcome to a participant's results [POWER].

5.5.4 Survey creation

Creating custom survey components is made possible by a visual editor. Internally, the survey panel uses SurveyJS⁵, which provides a visual survey creation tool. This allows researchers to create a survey in an intuitive manner, as in shown in Figure 5.3. We stressed the importance of accessibility in being able to create a survey, to ease the process of setting up an experiment. This library provides many common survey components such as multiple-choice selections, free-form text input with optional validation, and drop-downs. Furthermore, if the researcher has a need for a more complicated component, custom widgets can be created⁶ to suit their needs.

These surveys can be placed within the environment, allowing participants to answer questions while they are interacting with the IDE. However, they can also be placed before the environment is displayed, allowing researchers to ask participants to self-report demographics, perform a warm-up task [52], or filter participants based on previous knowledge [SUPPORT].

5.5.5 Experimental results

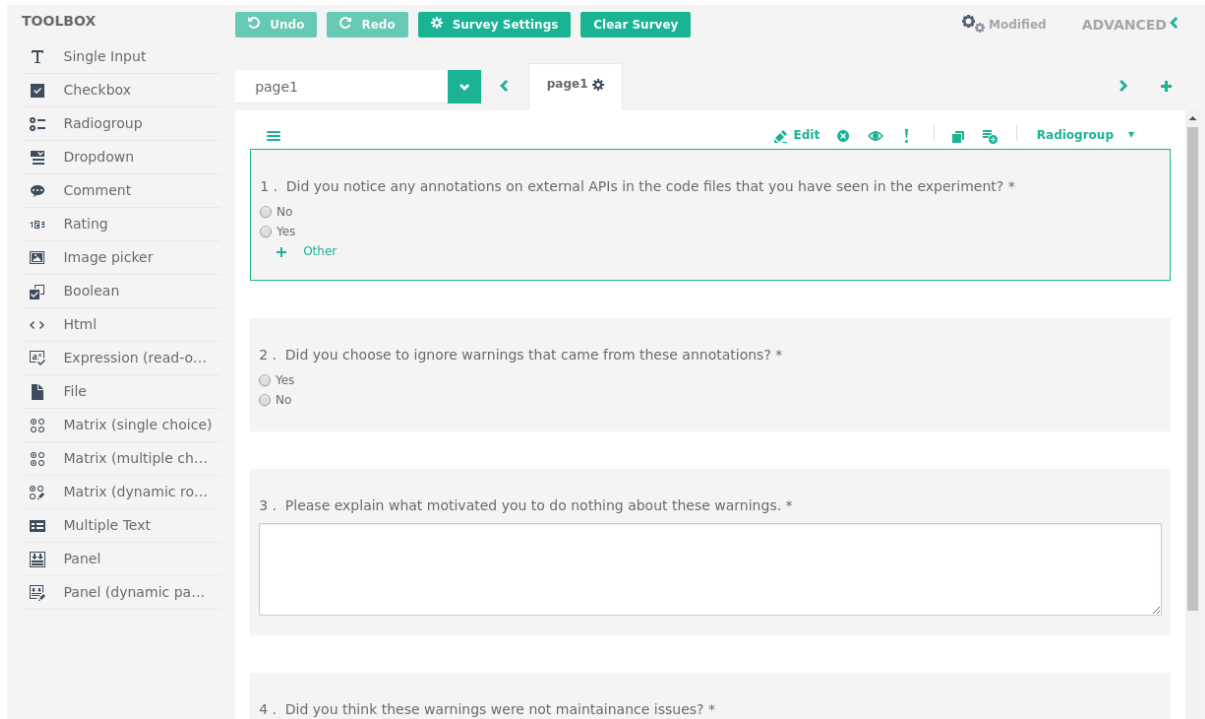
RESPIRED, by default, records the screen of each participant throughout the duration of the participant's session, ensuring that all participant activity can be reviewed after the session is completed. Furthermore, the participant's work-in-progress task files are periodically archived, giving a continuous view of a participant's progress.

The platform also offers components for recording survey results. Each answer to a survey question is saved into the participant's results along with the time of completion. These results can be aggregated by the tool, showing a basic distribution of answers given.

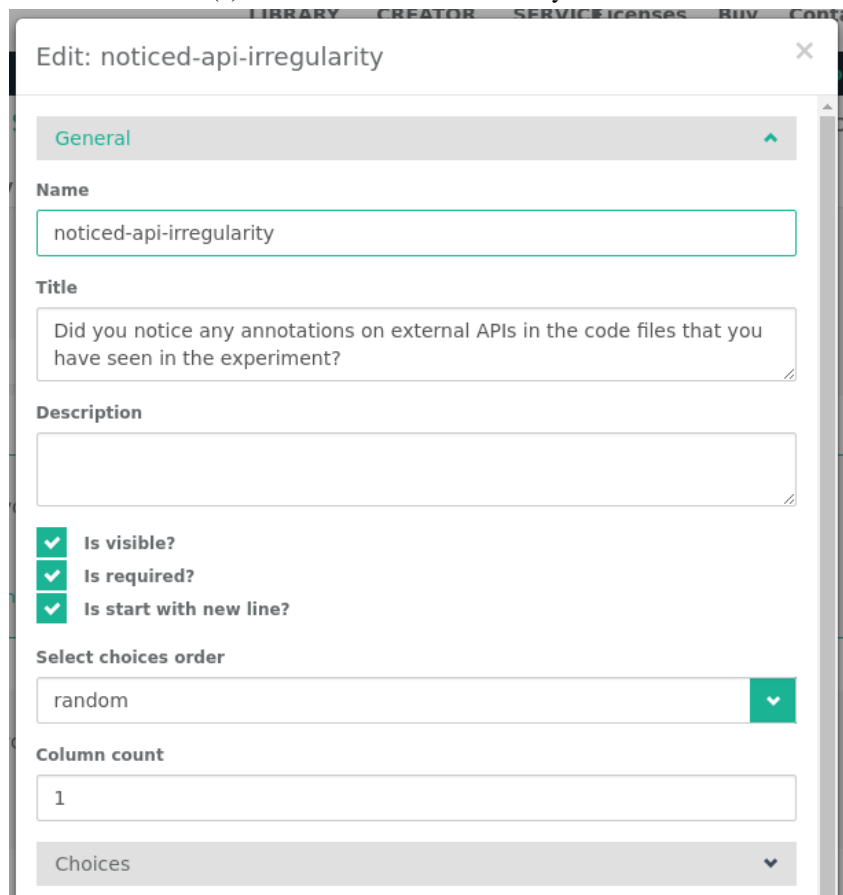
Though the tool gives a researcher a good foundation with which to start, all results are saved as JSON files, which can be read and processed as necessary.

⁵<https://surveyjs.io/>

⁶<https://surveyjs.io/Examples/Survey-Creator/?id=customwidgets>



(a) Overview of the visual survey creator.



(b) An example of configuration options that can be used for a single line input field.

Figure 5.3: SurveyJS's supplied components for survey creation.

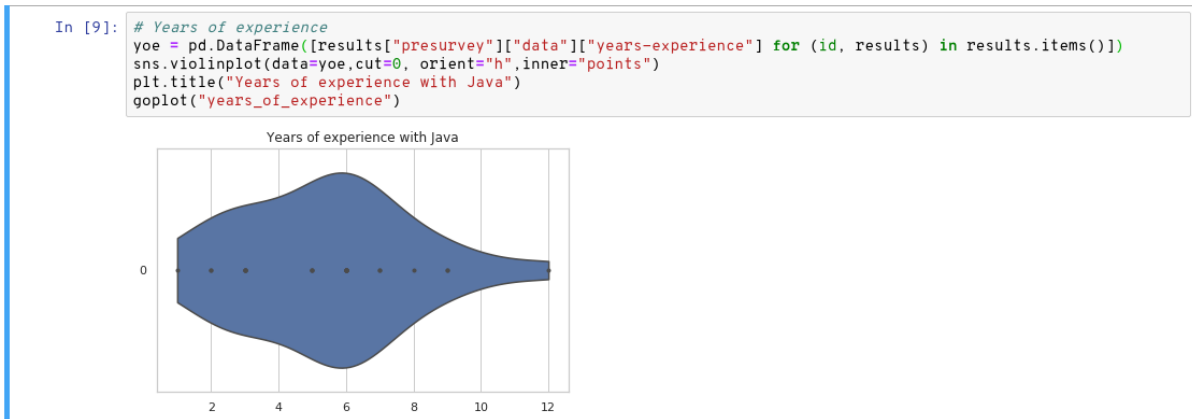


Figure 5.4: An example of creating a violin plot using Jupyter Notebook.

We support one specific manner of processing this data by including an installation of Jupyter Notebook [31], a web application that assists researchers in scientific computing and data science programming [DATA]. We provide scripts that are able to process output created by the platform, enabling researchers to start analyzing the data as it comes in. Jupyter supports Python 3, Julia, and R as programming languages, making it suitable for use by scientists familiar with these languages. Figure 5.4 shows an example of how a violin plot can be created using seaborn [62] and pandas [40] in Python 3.

5.5.6 Extensibility

In its default configuration, RESPIRED contains an IntelliJ IDEA IDE component. However, all JetBrains products that make use of the IntelliJ Platform SDK [3] are compatible with RESPIRED. This means that a researcher interested in Code Review could use JetBrains' UpSource, or a researcher interested in replicating a study in Python could use PyCharm. If these IDEs are not suitable for the researcher's needs, they could replace it with another component that implements RESPIRED's HTTP API [REUSE].

The survey panel is highly programmable, allowing for a large scala of different experiments to be conducted in the environment. This is because the Platform SDK is exposed through the RESPIRED plugin, meaning that the development environment can be inspected and controlled through the interface that is exposed.

In short, the provided components are a good baseline for creating experiment environments: components have been written against a loosely coupled web interface that can be replaced to fit a researcher's needs.

5.5.7 Security

Security is an important topic for any service exposed to the internet. We assume the following threat model: the service has been distributed on the internet, and can be accessed by anybody. This is a difficult model: there is no filter on actors with malicious intentions. We list possible attack vectors and the mitigations that have been used to combat them.

Denial of Service attacks These attacks target the availability of the service. Since the platform can run arbitrary code, it is important to mitigate the damage that can be done in such a container. For example, all available CPU could be consumed in an experimental environment. For this reason, we limit the amount of instances that can be run and limit the amount of CPU cores that can be occupied by an instance. This ensures that processing power remains available for critical host operating system infrastructure. The same is done for RAM, preventing one instance from monopolizing all working

memory. Furthermore, we attempt to limit disk read and disk write speeds, limiting the maximum usage one instance can use.

Exploitation of computational resources A malicious actor can make use of the resources available in an instance for purposes that are not related to the experiment. We run these containers in an isolated network that cannot access the public internet, preventing external communication. This makes exploitation of resources more difficult: no software libraries that are not provided by the environment can be used. Furthermore, the environment cannot be used to participate in DDoS attacks or cryptocurrency mining efforts.

Container escape However, it cannot be said that the platform is completely secure. The platform uses Docker [7] for process isolation. Common Vulnerabilities and Exposures [6] lists vulnerabilities in Docker, showing that attacks such as privilege escalation and sandbox escape exploits have already been discovered [18]. Future vulnerabilities in Docker can lead to a compromise in the system hosting the tool.

Combe et al. [18] list common flaws in Docker configurations and mitigations that can be used by end-users to prevent known errors. Finally, Martin et al. [36] state that containers such as those provided by Docker are part of a complex ecosystem: this leads to a wide surface area for attack.

Though a lot of work has been done in order to prevent misuse of the platform, we do not guarantee a complete secure software package; future work should involve cybersecurity specialists and penetration testing for the sake of platform stability.

5.6 Tool Distribution

The platform has been released as an artifact in the interest of furthering advancements in the Software Engineering Research community. The platform is open-source, released on GitHub under an organisation⁷. In this way, any researcher interested in the platform can create their own experiment, adjusting the foundation given on that repository [REUSE]. Furthermore, these new experiments can be released as a replication package, helping researchers to create reproducible studies [REPRO].

5.7 Tool Evaluation

A Rapid Iterative Testing and Evaluation study (RITE) [41] was conducted in order to ensure that the prototype of the tool is sufficiently usable. The RITE method is a usability test in which changes to the interface are made as soon as a problem is found by a participant. Then, this iteration of the interface is used with the next participant. The RITE method allows engineers to find a high percentage of usability issues with a small amount of participants.

For the evaluation, a sample experiment was created, in which participants were instructed to remove the text `throw new UnsupportedOperationException();` from a file. Then, they were instructed to make a test pass by trivially adding the line `return true;` to the same file. Afterwards, the demo was completed, and the participant was thanked for their participation.

We solicited seven students from the TU Delft to attempt to walk through a prototype of the tool. For each student, we noted their operating system and browser of choice. The participants were instructed to think aloud and comment on anything that seemed off with their experience with the tool. Participants were given the link and allowed to read and interact with the tool at their own leisure.

The first three participants remarked upon usability issues relating to visual styling. In particular, participants found that the survey interface seemed crude, and that the buttons were not discoverable enough. This was fixed immediately after.

⁷<https://github.com/respired/respired>

Furthermore, the third found that a button on an Apple operating system did not work as expected. After these three participants, we had four trials in which no issues to be fixed were discovered, and could conclude that the demo experiment passed our criteria for the RITE evaluation.

5.8 Requirements Revisited

We return to the requirements posed in section 5.2 and detail how each requirement is met by the features we provide in the platform.

- **[REPRO]**: Experiments created with RESPIRED are simply Docker containers. These container images can be distributed and executed with ease, allowing researchers to inspect and replicate a study conducted with the tool.
- **[REUSE]**: Though the platform has been especially tailored for IntelliJ IDEA, we allow other components to be used in the platform by using a web API to communicate between the survey component and the IDE. Furthermore, survey behavior within the environment is programmable, allowing experiments to be configured in various ways.
- **[FAM]**: IntelliJ IDEA is included in platform by default, offering a familiar and realistic environment for participants. Compared to existing solutions, this is advantageous because it allows participants to express themselves similarly to an actual development scenario.
- **[POWER]**: We offer a powerful two-way communication mechanism between the IDE and the survey environment, allowing for dynamic experiments based on IDE state.
- **[REMOTE]**: Participants do not have to be physically present in order to participate in the experiment, as the experiment is disseminated via the web. Furthermore, no installation is needed, as the participant only needs a web browser.
- **[DATA]**: We include Jupyter Notebook and provide utilities that can read participant results, allowing researchers to process their data with ease.
- **[SUPPORT]**: The platform has built-in support for participant groups, filtering participants based on demographics, and a large set of survey components.

Chapter 6

Evaluation of Deprecation Alternative via a User Study

Now that the alternative to deprecation misuse has been created in chapter 4, and an experimental platform has been created in chapter 5, we can conduct our developer user study. In this chapter, we detail our user study, describing the setup for the controlled experiment aimed at evaluating the difference between users who encounter a deprecation misuse and our offered alternative. Finally, we show the results that our participants provided in two programming tasks, and a post-survey.

6.1 User Study Design

The goal of this chapter is to assess to what extent our alternative annotations are effective. We define effectiveness as an improvement in inspiring API consumers to take action on this piece of code. We hypothesize that a new message that encodes the API producer's intent more clearly will result in API consumers avoiding usage of the annotated code fragment.

To evaluate the effectiveness of the solution we offer, we run a controlled experiment. The goal of the experiment is to evaluate the difference in behavior of users in the control group compared to users in the experimental group. We test how effective these new annotations are compared to the `@Deprecated` annotation in the following way:

1. We recruit participants, informing them about the experiment and asking for informed consent.
2. We ask for participant demographics in a short questionnaire.
3. We ensure that they are skilled enough to complete the tasks using a short skills test.
4. We place them in the experimental environment, splitting the participants into a control group and an experimental group.
5. We collect results from all participants and evaluate the difference in behaviors between the control group and the experimental group.

The following sections list the considerations we made and detail each part of the experiment.

6.1.1 Recruitment

We asked graduate students at the TU Delft to participate in the experiment. Furthermore, the survey was disseminated within the Software Engineering Research Group at the TU Delft, and participants were requested to ask Java developers in their social circles to participate as well.

6.1.2 Experiment and consent information

When a participant decides to participate, they are given brief information concerning the experiment. We explain our affiliation and describe that the participants will be working with deprecation program elements using IntelliJ IDEA. Furthermore, we explain how the data that they provide will be processed, and what data will be collected. Only after receiving a participant’s consent can they continue to the next part of the experiment.

6.1.3 Pre-experimental questionnaire

By means of a pre-experimental questionnaire, we ask the participant to self-disclose information relevant to their programming experience. We ask the following questions in this pre-experimental questionnaire:

1. **Q: Do you consider yourself a Java developer?**

A: Yes / No

We exclude non-Java developers from the experiment. Developers who are not familiar with Java are not a part of the target audience as they are less likely to understand signals triggered by annotations.

2. **Q: How many years of experience do you have with the Java language?**

A: Free input (must be numeric)

We are interested to learn whether length of familiarity with Java is correlated with the likelihood of being dissuaded by alternative annotations.

3. **Q: In which setting do you use Java?**

A: Industry, Open Source, Other (specify) (multiple selections possible)

We are interested to learn whether a difference exists between developers in industry and open source. We allow the “Other” option, allowing participants to report other setting such as academic or hobby work.

4. **Q: How often do you make use of libraries which are not the Java Standard Library when developing for Java? (Examples are Guava, ...)**

A: Never, Almost Never, Sometimes, Almost always

We would like to know if a correlation exists between (lack of) library use and likelihood to respond to deprecation misuse alternatives.

5. **Q: What is your IDE of choice when developing Java applications?**

A: IntelliJ IDEA or related product, Eclipse, Other (specify)

Since the alternative method of signal is integrated in IntelliJ IDEA, we are interested to know whether users of other IDEs are as affected by the signal as IntelliJ IDEA users.

If a participant does not self-report to be a Java developer or if they have zero years of experience with Java, they are exempted from the survey. Then, a cookie is set in the participant’s browser, which serves as a prevention mechanism against retrying the survey. This is not a flawless solution however: a participant clearing their cookies could retry the experiment.

6.1.4 Skills test

We create a test of skill to ensure that a participant has at least a baseline of skill in Java. This task also functions as a warm-up task, lowering participant drop-out [52]. We request that the participant answer the following questions:

```
1 ArrayList<Integer> arr = new ArrayList<>();
2
3 for (int i = 0; i < 10; i ++){
4     arr.add(i);
5 }
6 System.out.println(arr.get(3));
```

Q: What is the result of running the code above?

A: 1, 2, 3, 4, 5 (in randomized order)

```
1 public String comboString(String a, String b) {
2     String result = "";
3     if (a.length() < b.length())
4         result = a + b + a;
5     else
6         result = b + a + b;
7     return result;
8 }
```

Q: What is the output of this function if we call it using `comboString("Hello", "hi")`?

A: "HellohiHello", "HellohiHellohi", "hiHellohi", "null" (in randomized order)

We use these questions to assert that the participant has knowledge of lists, control structures, and basic operations. If a participant does not answer these questions correctly, they are exempted from the survey. Then, a cookie is set in the participant's browser, which serves as a prevention mechanism against retrying the survey. If they do answer these questions correctly, they are allowed to continue to the development environment.

6.1.5 Group creation

To be able to evaluate the difference between the existing misuse of the deprecation mechanism and our offered alternative, we use a controlled experiment. In the control group, the participant will interact with the misuse of deprecation, and in the experimental group, we present the participant with our alternative.

To distribute participants among these groups, we use a sequential round-robin assignment. In this assignment, participants are sequentially distributed amongst these groups, meaning that participant 1 belongs to group 1, participant 2 to group 2, participant 3 to group 3, and so on, wrapping back around to the first group when there are no more groups available.

In an ideal experiment, we would test each of the five misuse categories, evaluating the effect each annotation has on a participant. However, this master dissertation has a limited scope of participants that are available to us. Spreading a low number of participants across two groups for each participant (10 groups total) means that a low number of people would be in each group. In order to be able to draw statistically interesting results from data, we should aim to have many participants in each group.

To ensure statistical relevance, we lower the number of groups under test. In this experiment, we discard `@Internal`, `@Temporary`, and `@NeverUse`, leaving `@Beta` and `@AlwaysThrowsException`. We choose these annotations because of their prevalence within the deprecation dataset. We create participant groupings using these annotations as shown in Table 6.1.

For each annotation, we create both a control group and an experimental group. The control group will encounter deprecation, and the experimental group will have this deprecation replaced with our alternative annotation. In this way, these groups can be compared against each other.

Group number	Annotation	Group name
1	@Beta	BETA_EXP
2	@Deprecated	BETA_CONTR
3	@AlwaysThrowsException	ATE_EXP
4	@Deprecated	ATE_CONTR

Table 6.1: Participant group ordering. **CONTR**: control group. **EXP**: experimental group.

Listing 6.1: Definition of `FileUtils.writeStringToFile(File, String)` in Apache Commons IO.¹

```

1  /**
2   * Writes a String to a file creating the file if it does not exist using
3   *   the default encoding for the VM.
4   *
5   * @param file the file to write
6   * @param data the content to write to the file
7   * @throws IOException in case of an I/O error
8   * @deprecated 2.5 use {@link #writeStringToFile(File, String, Charset)}
9   *   instead (and specify the appropriate encoding)
10  */
11 @Deprecated
12 public static void writeStringToFile(final File file, final String data)
13     throws IOException {
14     writeStringToFile(file, data, Charset.defaultCharset(), false);
15 }

```

6.1.6 Library under test

Our research question deals with the effectiveness of the message we display to the end-developer, meaning that the participant plays the role of an API consumer. Therefore, we should provide an API that can be used by these consumers. In the control groups, we leave these libraries as is, and in the experimental group, we modify these libraries to use our alternative annotation in cases where this makes sense.

For the @Beta groups, we choose the library `commons-io:commons-io:jar:2.7`. Commons IO is a library by the Apache Foundation, aimed to assist with the development of IO functionality such as reading and writing to files and other file system events. This library is suitable for our experiment because it makes for a plausible experiment scenario that could use a deprecated API: `FileUtils.writeStringToFile` and related methods. A selection of these methods do not specify a `Charset` that should be used when writing the `String` to a file. These methods are marked @Deprecated, as API consumers should specify a `Charset`. Listing 6.1 shows the definition of one such method. We adjust all methods in this group, replacing mentions of @Deprecated with @Beta for the experimental group and updating documentation where necessary.

For the @AlwaysThrowsException groups, we need a library that uses this misuse of deprecation in a plausible manner. We choose Google’s Guava, `com.google.guava:guava:jar:28.0-jre`. This library provides an implementation of `ImmutableList`, a read-only implementation of Java’s `List`. Since this class implements `List`, it must confirm to `List`’s interface contract², which specifies that all implementations of `List` must specify an implementation for `add`, `clear`, `remove`, and others. However,

¹<https://github.com/apache/commons-io/blob/58d9d82879b1f/src/main/java/org/apache/commons/io/FileUtils.java#L3068>

²<https://docs.oracle.com/javase/8/docs/api/java/util/List.html>

Listing 6.2: UNSUPP example in ImmutableMap (com.google.guava:guava)³

```

1  /**
2  * Guaranteed to throw an exception and leave the map unmodified.
3  *
4  * @throws UnsupportedOperationException always
5  * @deprecated Unsupported operation.
6  */
7  @Deprecated
8  @Override
9  public final void clear() {
10     throw new UnsupportedOperationException();
11 }

```

these operations are meant to mutate the `List` on which it is called, which clashes with the immutability of the list. This is a violation of the Liskov Substitution Principle [37], as the `ImmutableList` does not obey the same contract as for `List`.

To work around this, Guava implements these methods as shown in Listing 6.2, marking these mutational implementations as `@Deprecated`. When these mutation methods are called, a `UnsupportedOperationException` is thrown. These methods are also marked `@Deprecated` to warn users that these methods should not be called. We modify this library for the control group, annotating these `@Deprecated` methods with `@AlwaysThrowsException` and updating the documentation where necessary.

We end with two versions for each of the two libraries. To summarize, Table 6.2 shows the groups and the libraries that participants in these groups must use.

Group name	Library
BETA_EXP	Commons IO (Deprecation replaced with <code>@Beta</code>)
BETA_CONTR	Commons IO (unmodified)
ATE_EXP	Guava (Deprecation modified with <code>@AlwaysThrowsException</code>)
ATE_CONTR	Guava (unmodified)

Table 6.2: Participant group ordering. **CONTR**: control group. **EXP**: experimental group.

6.1.7 Implementation Task

When interacting with an unknown API, consumers can make use of IDE features such as auto-completion and fragment suggestions [12] to discover API features, or can query external (web) resources [54] to help form decision-making. When writing a code fragment, an API consumer has direct feedback from the IDE on the code they are writing [30]. In this instance, the feedback should be that the use of the experimental API is either deprecated or should be avoided for another reason. When confronted with this information, how will the participant react?

In short, in the implementation task, we are interested in discovering whether or not a participant is more likely to avoid a method annotated with an alternative annotation compared to the control group.

We show the participant the file listed in Listing 6.3 if they are in the **BETA** groups, and show the participant the file in Listing 6.4 if they are in the **ATE** group.

For participants in **BETA**, we expect participants to use `FileUtils.writeStringToFile(File file, String data)` to write a `String` to a `File`. However, when doing so, the IDE will warn the

³<https://github.com/google/guava/blob/6e9069225a46bb152fb7b3fc20324c1600b397df/guava/src/com/google/common/collect/ImmutableMap.java#L665>

Listing 6.3: Implementation task file for the BETA version

```
1 import org.apache.commons.io.FileUtils;
2 import java.io.File;
3 import java.io.IOException;
4
5 class Task {
6     /**
7      * Please complete this method.
8      * The method should write the string in 'toWrite' to 'writeDestination'
9      * Please use FileUtils in your solution.
10    */
11    public void writeString(File writeDestination, String toWrite) throws
12        IOException {
13        // TODO complete this method.
14    }
```

Listing 6.4: Implementation task file for the ATE version

```
1 import com.google.common.collect.ImmutableList;
2
3 class Task {
4     ImmutableList<String> myList;
5     public Task(ImmutableList<String> input) {
6         this.myList = input;
7     }
8
9     /**
10    * Please complete this method.
11    * The method should add the String 'toAdd' to the list contained in
12    * the field 'myList'.
13    */
14    public void addItem(String toAdd) {
15        // TODO complete this method.
16    }
```

participant that this method is annotated, and suggest usage of `FileUtils.writeStringToFile (File file, String data, Charset charset)` instead. This replacement fragment is preferred because the character encoding should be specified explicitly.

For participants in **ATE**, the trivial solution is to add `myList.add(toAdd)`. However, `myList` is of type `ImmutableList<String>`, so it is illegal to modify this collection. The IDE will warn about this if the participant does so. Instead, the participant is to add the item to the collection by using the `ImmutableList.Builder` to create a new list and replacing the reference.

Participants in both versions are only allowed to continue when ten characters have been changed in the source file, excluding comments. Furthermore, we check that the solution can be compiled. For the **BETA** groups, additionally, the participant must use the string `FileUtils` in their solution to ensure that the API is being used instead of an alternative solution.

6.1.8 Maintenance task

The maintenance task examines another side of a development task: given an existing code fragment, do API consumers feel motivated to refactor a usage of deprecation (or an alternative thereof)? In the review task, we present the participant with a file that has multiple reported warnings. We ask participants to maintain the file, that is, to fix maintainability issues at their own discretion.

We show the participant the following text:

```
In this task, you will be asked to review a code file.
Please open the code file using the button below.
<Open code file>
```

Upon clicking the button, the participant is shown the file listed in Listing A.1. We continue with the following text:

```
In this scenario, you are a programmer working on this grade administration
  system.
Your colleague has made some changes to this file. It is your job to
  maintain these changes.

Please take a some time to read over the code to your left. Then, answer
  the following question:

Do you think changes should be made to this file to make it more
  maintainable?

o Yes
o No
```

If the participant answers No, we direct them to the next component. Otherwise, we show them the following text:

```
Please adjust the file to make the code more maintainable.
When you are done, press the Next button to continue.
```

The file contains an example codebase we created, a simple component called `GradeAdministration`. This code file is meant to represent a component that administers student grades. It exposes methods for calculating student averages, writing the administration to a file, and adding new grades to the administration.

We manually added maintainability issues, using IntelliJ IDEAs list of inspections for Java [2] as inspiration for maintenance issues that can be warned upon by the editor. We introduce only maintainability issues that are enabled by default in the editor, choosing plausible issues based on intuition. This results in the following list:

Line	Code	Inspection text	Possible resolution
5	import java.util.LinkedList;	Unused import java.util.LinkedList	Remove this line
9	class GradeItem implements Serializable	'GradeItem' does not define a serialVersionUID field	Add a serialVersionUID field
56	List<GradeItem> grades;	Access can be private	Add access modifier 'private'.
76	sum += (double) item.getMark();	Casting 'item.getMark()' to 'double' is redundant	Remove casting operator
141	while (iterator.hasNext()) { FileUtils.writeString-	'while loop' replaceable with 'foreach'.	Refactor while loop for foreach
145*	ToFile(file, builder.toString());	<i>Message dependant on participant annotation</i>	Replace element with alternative
146	System.out.println(e);	Throwable argument 'e' to 'System.out.println()' call	Change to e.printStackTrace();
148	catch (RuntimeException e) {	'catch' branch identical to 'IOException' branch	Merge exception handling branches
158†	grades.add(item);	Unnecessary semicolon ';'.	Remove semicolon
161	};	Unnecessary semicolon ';'.	Remove semicolon
167	foundGradeFor = true;;	'foundGradeFor == true' can be simplified to 'foundGradeFor'	Simplify expression
170	return foundGradeFor == true;		

Table 6.3: Maintainability issues introduced into the GradeAdministration code file. *: BETA group only. †: ATE group only.

In the maintenance task, we present the participant with a file that has multiple reported warnings. We ask participants to maintain the file, that is, to fix maintainability issues at their own discretion. Twelve reported warnings have been introduced, of which a single warning is relevant to the research question. We measure how many times participants change the warning relevant to either the deprecation or deprecation alternative, and find out if the alternative message and annotation lead to more maintenance fixes.

The goal of this maintenance task is to discover the discoverability of the reported warnings. It is important to note that the warnings introduced into this code file are highlighted in a different manner than both deprecated elements or elements marked with an alternative annotation. Figure 6.1 shows an example of the visuals of such a warning. The highlighted element receives a yellow background, visually differentiating the element from other elements, compared to the strikethrough for deprecation or an alternative, shown in a previous chapter in Figure 4.1.

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

Figure 6.1: An example of the manner in which IntelliJ IDEA signals the introduced warnings by default.

6.1.9 Post-experiment Survey

When the participant is done with the review task, they are presented with a final survey. The goal of this survey is to discover a participant’s reasoning for acting on or not choosing to act on alternatives to deprecation misuse.

We use a dynamic survey to discover these reasons, the choice tree of which is shown in Figure 6.2. We begin by covering the base case: the participant is asked to self-report whether or not the deprecation or deprecation alternative was noticed. If so, they are asked whether or not the reported warning was ignored, and finally asked to elaborate upon their reasoning. Otherwise, we ask whether or not the warning is not a maintenance issue.

Open-ended questions are important in qualitative research, capable of revealing important issues [23] and allowing for spontaneous responses [49]. We use an open-ended question to discover what motivates a participant’s reaction or non-reaction to the annotated API, and manually code participant responses.

6.1.10 Experiment ending page

When the participant has completed the final survey, the participant is instructed to finish the experiment. A final page is presented to the user, thanking them for their participation and stating that the experiment can be closed. This concludes the participant’s experience with the participant.

6.1.11 Summary

We create a user study aimed at developers designed to evaluate a difference between the misuse of deprecation and our proposed alternative. Our experiment asks for participant demographics before conducting a small skills test. After this, the first task asks participants to implement a function that drives them to interact with an annotated API. In the second task, participants are asked to maintain a file with many maintenance issues, of which one is a maintenance issue related to the annotated API. Figure 6.3 shows an overview of the experimental setup in a high-level overview.

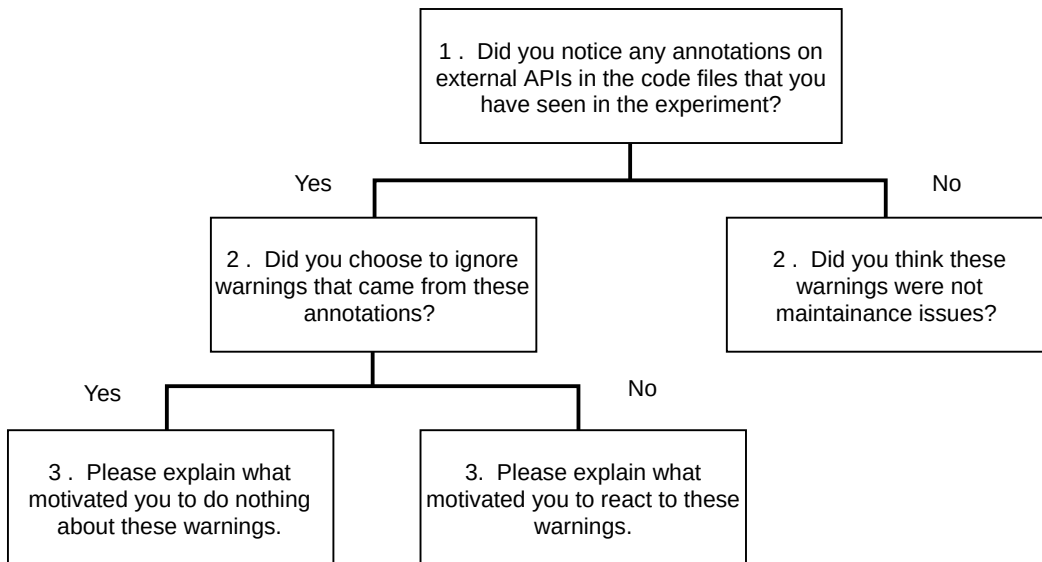


Figure 6.2: Post-experiment choice tree for discovering participant sentiment.

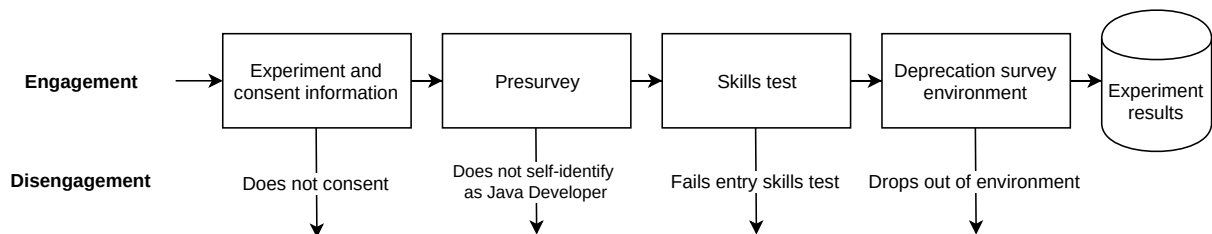


Figure 6.3: Deprecation experiment overview

6.2 Results

In this section, we describe the results of the user study described in the previous section. To evaluate the effectiveness of the alternative annotations, we take a critical look into the data that has been gathered. In this section, we evaluate the properties of the participant groups, and show the data that these participants have displayed.

6.2.1 Data filtering

In the first stage, we examine our participant's experimental recordings. As the experiment was distributed via an online link without any need for authentication or other cost, there is a possibility that some results were an outcome of a non-serious attempt. For this reason, we manually inspect each screen recording: for results which we deem were not a serious attempt at the experiment, we remove this result from the dataset. We removed two of such results. Furthermore, we only consider participants that have completed the entirety of the experiment. Since the experiment is conducted inside a participant's browser, it is possible for a participant to abort the experiment at any time by closing the window in which the experiment takes place. Though it is possible to use these partial results, for brevity and clarity these results are not taken into account in this dissertation.

After this filtering, we are left with 25 results.

6.2.2 Pre-survey: Demographics

We start with demographical data into our participants. As described in section 6.1, we start by asking our participants to self-report their demographics.

Figure 6.4a shows the distribution of the years of experience our participants reported to have. Figure 6.4b shows the settings in which these participants use Java. We see that industry practitioners are represented within the group of participants, along with open-source developers. We examine the “other” category, to see self-reported answers to this question. The answers we see in this category are related to work during a degree, with answers similar to “University” and “Study projects”. Figure 6.4c shows the combinations of settings that participants chose. Figure 6.4d shows the IDE participants prefer to use. A concern was that participants would not be used to the IDE used, leading to a signal being missed. However, we see that a large majority of the participants prefer to use IntelliJ IDEA, meaning that these participants are likely to be familiar with the way warnings are shown in this IDE. Figure 6.4e shows how often participants indicated to use libraries when developing in Java.

Additionally, we observe the groups the participants have been placed in. Figure 6.4f shows the group distribution within the participants. We observe a skew in the number of participants in the group `BETA.CONTRRECATED`. Due to a bug in the manner in which participants were distributed, this group has twice as many participants as the other groups.

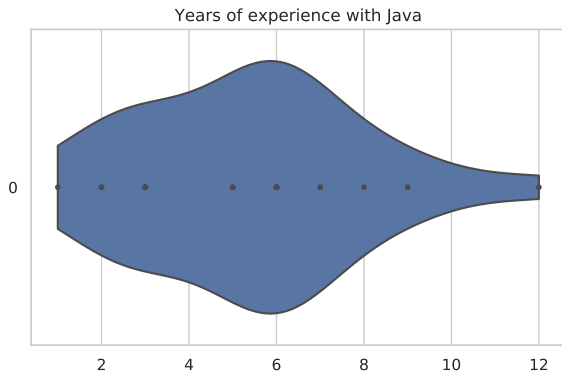
6.2.3 Implementation Task

We start with evaluating the implementation task. In this task, the participant was asked to make use of a deprecated (or otherwise annotated) API to implement a function.

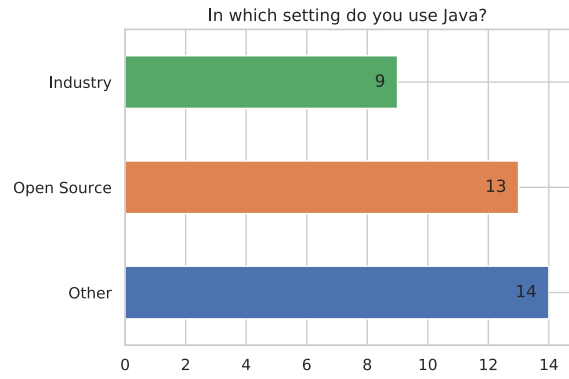
We use the participant’s video screen recording and annotate the recording with additional metadata. We record whether or not the participant has hovered over the annotated term for long enough for the warning text to be displayed, since there is no visual distinction between the alternative annotation and `@Deprecated` otherwise. The intuition is that seeing the targeted message will motivate a developer to handle the API consumption differently. Then, we judge if the annotated element was used in the participant’s final solution.

To visualize the results, we plot a Sankey diagram [35], a type of visualisation that shows flow. On the left, we plot the participant’s group. These groups then flow to whether or not the participant saw the warning text. Finally, this flow continues to whether or not the task was solved without using the annotated element. This is shown in Figure 6.5. We see four different paths a participant can take:

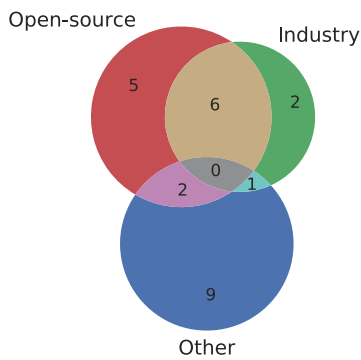
1. Participant sees the warning text, and avoids the annotated method (9 participants). In this case, we can observe the difference between participants in the control groups and the experimental groups: the control group sees the generic `@Deprecated` warning text, where the experimental group receives the elaborated text. This participant also avoids using the annotated method.
2. Participant sees the warning text, and uses the annotated method despite this (6 participants). In this case, the participant are also subjected to the difference between the groups. However, the participant uses the annotation despite seeing warning text.
3. Participant does not see warning text, and avoids the annotated method (4 participants). This can be explained by previous participant knowledge, i.e. the participant knows to add a `Charset` when using `write` without needing information from a warning to dissuade the improper method use.
4. Participant does not see warning text, then does not avoid the annotated method (7 participants). In this case, the participant does not interact with the difference between the experimental group and the control group.



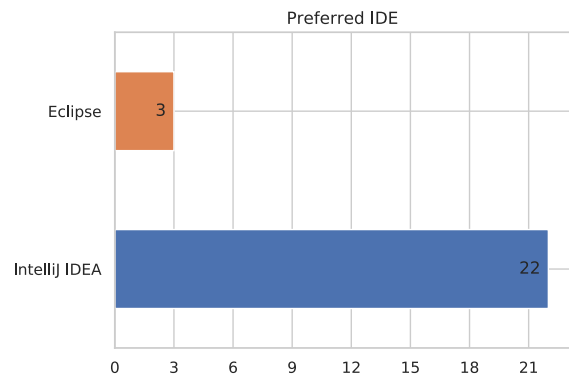
(a) Years of experience



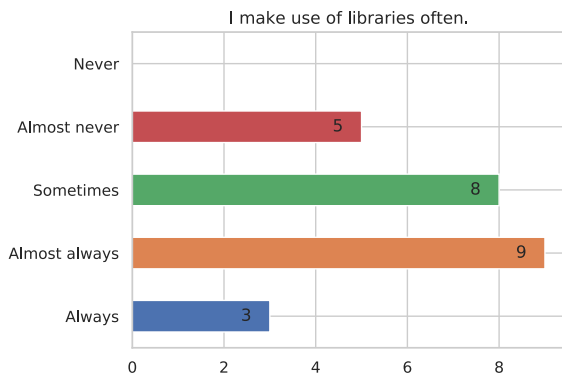
(b) Setting in which Java is used



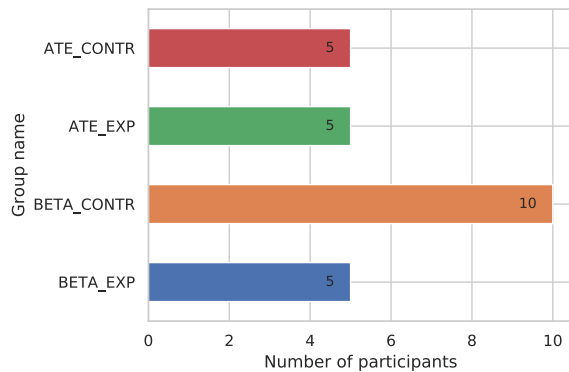
(c) Venn diagram with settings combinations



(d) IDE usage



(e) Library usage



(f) Participant group distribution

Figure 6.4: Diagrams of participant demographics

6.2.4 Maintenance task

For the second task, we showed participants a code file to which we added several maintenance issues. First, we look at a general overview of the maintenance issues that were solved by participants. To show which issues were most often tackled by participants, we present Figure 6.6a. This shows which issues were most often solved by participants, demonstrating that the most issues were reasonable to expect participants to resolve. We use number of maintenance issues solved as a proxy for participant effort. By plotting number of maintenance issues solved by participants in each group, shown in Figure 6.6b, we can lend more credibility to the results in each group.



Figure 6.5: Task 1 deprecation flow diagram

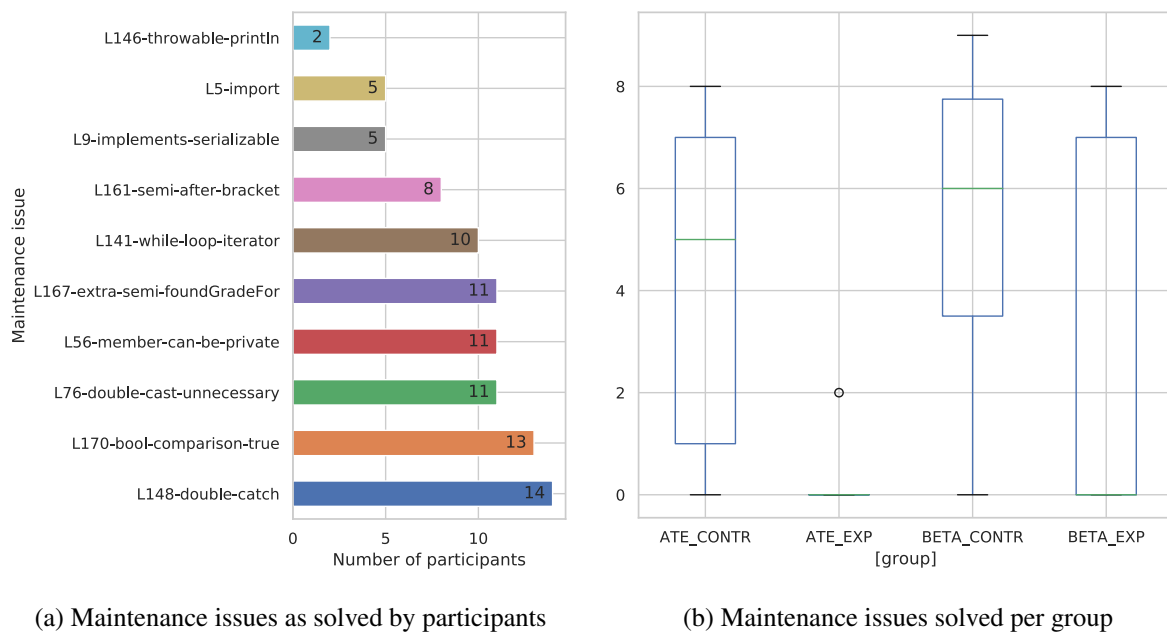
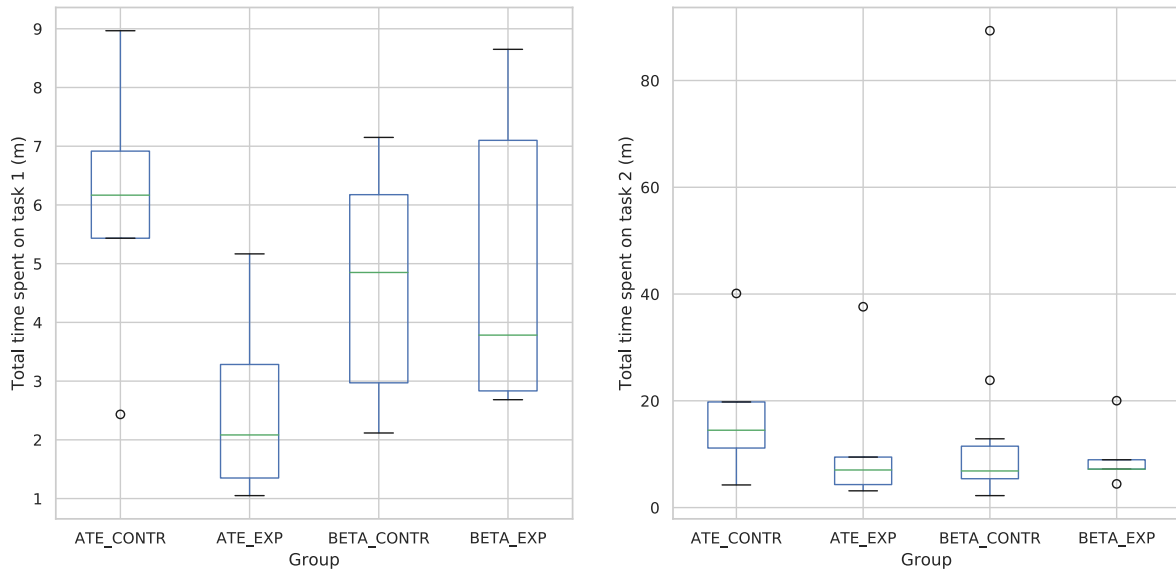


Figure 6.6: Task 2 maintenance issues results

We find that three groups are similar in number of maintenance issues solved, with the `ATE_CONTR` group having a significantly lower amount of issues resolved. We examine possible reasons for this, checking the distributions of time spent on each part of the experiment, found in Figure 6.7. In Figure 6.7a, we see that this group spent a far lower amount of time on task 1. In Figure 6.7b, we see no large disparity between this group and other groups. To elaborate on where these participants spent their time if it was not on maintenance issues, we inspect these participants' recordings. We find that these participants spent their time on this task on documenting code and rewriting parts of the implementation that were not issues that we sought after. From this, we suspect that the task was not clear enough, and that this way of measuring does not accurately represent avoidance of annotated usages.



(a) Time spent on the implementation task.

(b) Time spent on the review task.

Figure 6.7: Distributions of time spent on each section of the experiment.

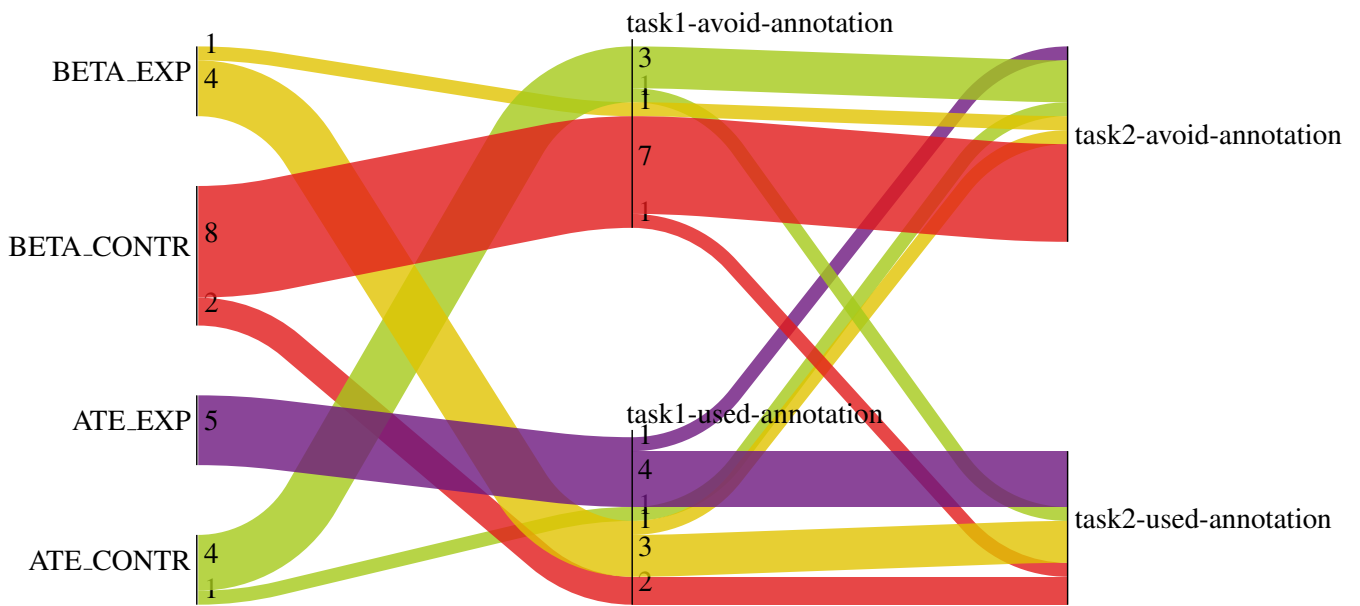


Figure 6.8: Participant groups avoiding the annotation in task 1 and task 2.

6.2.5 Relation between Implementation Task and Review task

We also investigate the relation between this task and the first task. In particular, we examine participants who used the annotated element in the implementation assignment. These participants were able to observe at least some form of dissuasion: every used method is decorated with strikethrough to indicate that this method should not be used. However, these participants elected to use this method despite these warnings. We suspect that these participants will not resolve the same method call in task 1, and plot this in Figure 6.8. This figure shows that this is correct for 20 out of 25 participants, where the remaining 5 are inconsistent in their behavior.

6.2.6 Post-survey results

For the final survey, we evaluate our participants' answers. Figure 6.9 shows a flow diagram of the answers the participant groups gave to the post-survey described in subsection 6.1.9. This diagram shows all combinations of answers a participant could give to the closed questions, and how many participants choose which options.

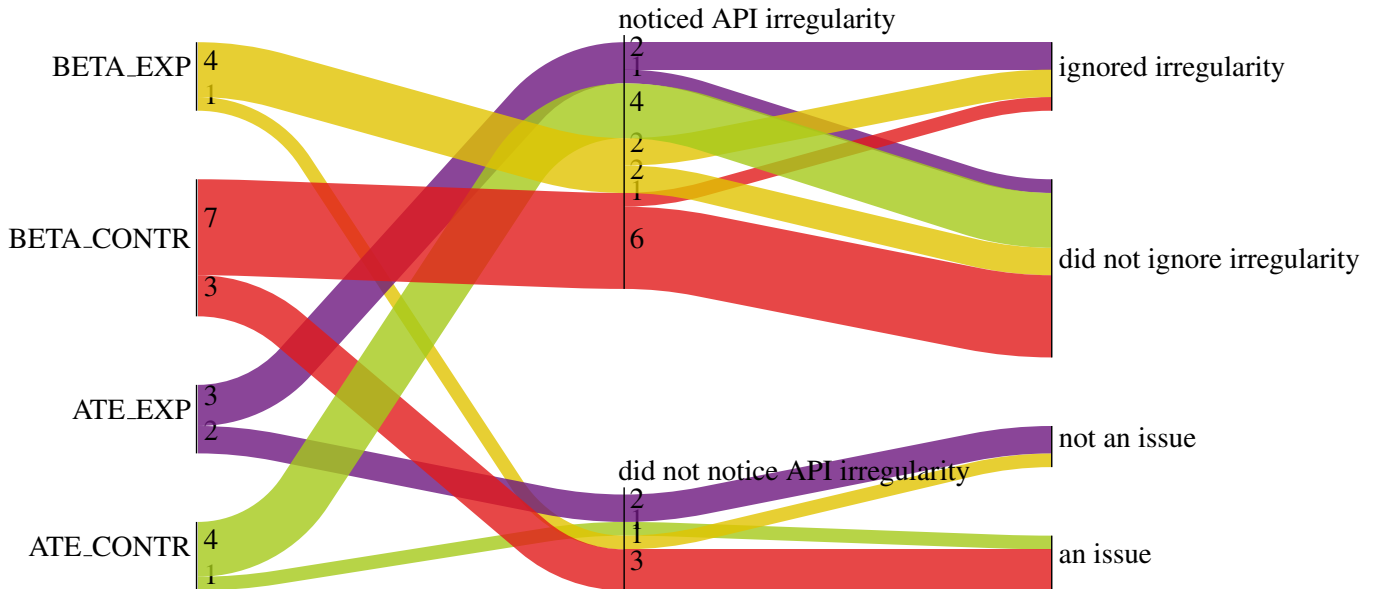


Figure 6.9: Post-survey questionnaire flow diagram

We also give an impression of the answers given to the open-answer questions, by distilling participant answers into codes. Table 6.4 shows the categorisation of participants' motivation to react to the API warnings we provide. Table 6.5 shows the categorisation of participants' motivation not to react to the API warnings.

Table 6.4: Resulting coding of post-survey answers to “Please explain what motivated you to react to these warnings.”

Group	Coding of motivation to react
P0 BETA_EXP	avoid-deprecation
P1 BETA_EXP	avoid-deprecation, declaration-documentation
P2 BETA_CONTR	avoid-deprecation
P3 BETA_CONTR	avoid-deprecation
P4 BETA_CONTR	avoid-deprecation
P5 BETA_CONTR	intellij-signal
P6 BETA_CONTR	avoid-deprecation
P7 BETA_CONTR	avoid-deprecation, accessible-alternative
P8 ATE_EXP	avoid-dep-alternative
P9 ATE_CONTR	refactor-maintainability
P10 ATE_CONTR	intellij-signal
P11 ATE_CONTR	avoid-deprecation
P12 ATE_CONTR	avoid-deprecation, intellij-signal

We give the following definitions to the tags:

Table 6.5: Resulting coding of post-survey answers to “Please explain what motivated you to do nothing about these warnings.”

	Group	Coding of reason to do nothing
P13	BETA_EXP	experimental-setting
P14	BETA_EXP	other-warnings
P15	BETA_CONTR	warnings-have-no-effect
P16	ATE_EXP	no-warnings
P17	ATE_EXP	uncertainty-of-change-effects, experimental-setting

- `avoid-deprecation`: Participants mention that deprecated elements should be avoided.
- `declaration-documentation`: Participants mention being assisted by declaration documentation.
- `intellij-signal`: Participants mention the way in which IntelliJ highlights the annotated use (strikethrough).
- `accessible-alternative`: Participants respond that the alternative to the annotated element was easy to find and use.
- `refactor-maintainability`: Participants point out that the current solution is not maintainable with no details.
- `avoid-dep-alternative`: Participants report a message related to the warning alternative annotation’s warning text.
- `experimental-setting`: Participants indicate that their behavior was different because this is an experimental setting.
- `other-warnings`: Participants talk about other warnings, but not any deprecated or otherwise annotated warnings.
- `warnings-have-no-effect`: Participants report that warnings have no effect on program behavior.
- `uncertainty-of-change-effects`: Participants indicate that they are unsure of the effects of changing the annotated element.

6.2.7 Relation between definition observation and dissuasion

In this section, we observe the relation between a participant observing the definition of the annotated element and avoiding use of the annotated element. Figure 6.10 shows the various groups, whether or not the participant saw the definition of the annotation element, and finally, whether the element was used or not in either the implementation task or the maintenance task.

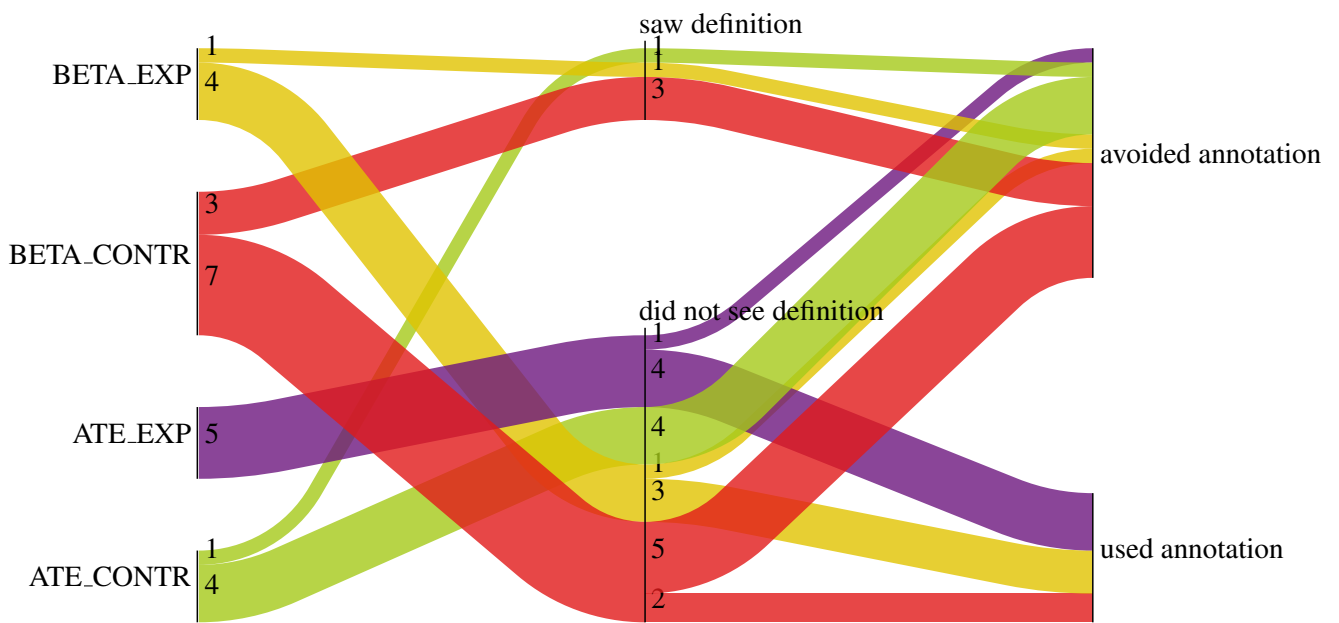


Figure 6.10: Flow diagram showing participant groups, definition click-through, and dissuasion

Chapter 7

Evaluation of RESPIRED

This deprecation study is the first usage of RESPIRED. In this chapter, we explore the utility the platform offered in this user study compared to other methods, demonstrating the value of the tool. We take multiple viewpoints on this work, examining the importance of this work for experiment participants and for the researchers conducting the experiment. Finally, we describe where the tool can still be improved, giving more avenues for future developments.

7.1 Value for Participants

One way the platform benefits our participants is in setup time for the experiment. As this user study required a modified JDK to function properly, without the platform, we would have to provide installation instructions and methods for many kinds of devices. In order to mirror the same feature set as RESPIRED, we would also have to request that participants install and activate screen recording software, and save and send results. Alternatively, we could offer participants a virtual machine, but installing the emulation software is fraught. This setup time is immensely more involved and longer than this web-based solution, and we believe that participants would be far less willing to participate if the setup had been so complicated. They were able to use hardware that they were used to, making for a more comfortable working environment.

Additionally, the participant could choose to work on the experiment when it was convenient for them, benefiting results by allowing participants to cooperate when their mental state was at its best.

Furthermore, this environment more accurately represents the manner in which an actual developer would work. Compared to using a simple survey form, participants are able to discover an API by making use of auto-completion suggestions. We see our participants make use of these possibilities, which played an important role in our user study. We also observe participants using “Go to declaration”, a feature that switches the current code view to the declaration of the selected element, allowing participants to view documentation and declaration details for external classes. Finally, we saw participants use features like checking refactoring suggestions, using these refactoring suggestions, and the potential bugs list. We believe that the platform allows developers to express themselves more naturally, leading to results that are more closely correlated to the reality of developer behavior and thus more accurate developer studies.

7.2 Value for Researchers

Ensuring that the environment is equal for each participant requires careful attention, as an incorrect reset can defeat the validity of a measurement and cause an otherwise eligible participant to become unsuitable for the experiment. Use of the platform ensures that the environment we create behaves in the same manner for each participant.

The installation of Jupyter Notebook [31] is useful as a starting point for data analysis. The scripts that are included allow us to manually classify participant results and start visualizing data using `matplotlib` [27], a popular plotting library for the Python programming language. This spares us cumbersome work in marshalling and aggregating data.

Finally, it was useful that the experiment could be conducted in parallel, and without having to loan out a computer. For this dissertation, it would not have been feasible to find different timeslots, reserve rooms, prepare a computer, and request the participant be physically present.

7.3 Future Improvements

Though the platform showed a promising perspective into what remote survey developer environments could be, there are still some points of improvement that should be worked on so that both researchers and participants can get the most out the platform.

- We observe participants moving their cursor out of the experiment window, and no interactions taking place for a small period of time. We speculate that participants use a different browser window to consult the internet [42], using online resources to learn. This is inconvenient because researchers cannot inspect what a participant is looking for and seeing as it occurs outside of the experiment. A possible approach for this could be to offer the participant a way to consult the internet within the experimental environment, allowing for it to be recorded just as the rest of the interactions are.
- We observe some erratic user behavior in screen recordings. It seems that in some user setups, a participant was unable to input `Alt + Enter`, a hotkey in IntelliJ IDEA that raises possible intention actions and quick-fixes at the current text cursor location. Instead, we observed just a press of `Enter`, the participant undoing the operation, another press of `Enter`, then another `undo`, then the participant raising the same window using a mouse command instead of the keyboard command. This leads us to believe that some setups remain in which modifier keys such as `Control`, `Alt`, and, on Apple-based computers, `Command`, work incorrectly. These cases should be identified and a update should be made to the platform that resolves these issues.
- The process of creating an experimental environment from start to finish is not simple. A researcher needs knowledge of `Docker` [7] to test, run, and deploy environments. In order to unburden researchers from needing the prior knowledge of the intricacies of `Docker`, work should be done into creating extensive documentation that does not assume this knowledge.
- In this experiment, we use time spent in each task to compare similarity between participant groups. However, this is not the same amount of time spent *on* the task, as participants can temporarily pause the experiment by switching to a different window. This could lower the credibility of time spent on a task as a metric. Future improvements could detect the active window on the participant's system, though this still does not account for physically leaving the computer.

Chapter 8

Discussion

This chapter focusses on answering the research questions, giving our interpretation of the data found in previous chapters, and presenting the threats to validity that are related to our study.

8.1 Research Questions Revisited

In this section, we repeat the research questions posed in chapter 1. We give an answer to each of the research questions based on our interpretation of the results found in previous chapters.

8.1.1 Deprecation Misuse Taxonomy

Research Question 1: What motivations are there behind developers misusing Java’s deprecation mechanism?

In chapter 3, we found five different reasons API producers have for misusing the `@Deprecated` functionality. These were categorized as **UNSUPP**, **TEMP**, **BETA**, **INTERNAL**, and **DONTCALL**. We propose a grouping for these reasons, (1) relating to versioning aspects, and (2) relating to deficiencies in the Java Programming Language. Figure 8.1 shows the proposed hierarchy for the taxonomy.

API versioning has to do with programming elements with an uncertain future: these could be removed or changed in the future. An example of a language that groups these reasons together is Kotlin, which provides its developers with `@Experimental` and `@UseExperimental` [28]. This allows a producer to

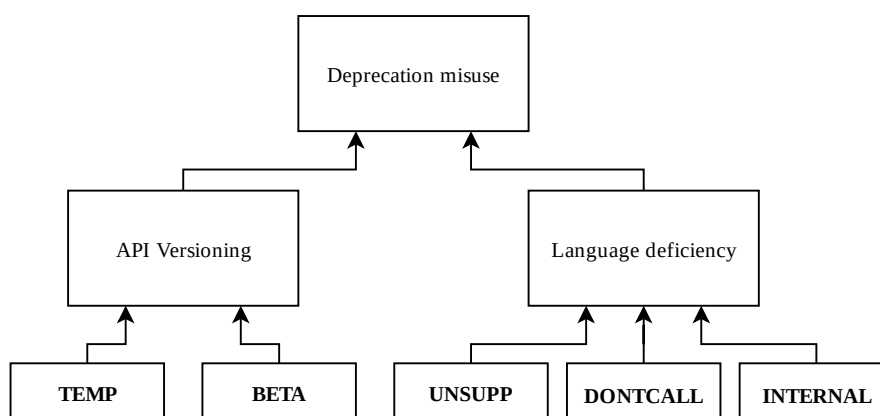


Figure 8.1: Taxonomy of deprecation misuse reasons.

mark an API or parts of one as experimental. Using one of these APIs without `@UseExperimental` causes a compiler warning to be dispatched upon a build.

Language deficiency refers to deprecation reasons used to work around an unmet need in the programming language. In the case of **UNSUPP**, we see that API producers want the benefits of `ImmutableList` being a sub-class of `List`, but without the requirement of having to specify mutable operations. Scala provides `scala.collection.AbstractSeq`¹, which does not have these mutable operations, creating a hierarchy in which these needs can be modelled. For **DONTCALL**, we observe API producers needing a more direct way to communicate with API consumers. Finally, **INTERNAL** refers to a deficiency in which current visibility modifiers do not suit developers' needs. An example of a language providing a new modifier is Kotlin, which provides the `internal` keyword², which gives visibility within the same module, but hides the element outside of the module.

We conclude that this taxonomy shows areas in which the Java ecosystem can yet improve. Java Language designers should add new mechanisms allowing API producers to communicate peculiarities concerning their API versioning. Furthermore, there are deficiencies in the language and standard library that cause API producers to mark parts of their APIs as deprecated, as it is a convenient method to message consumers of their APIs directly.

8.1.2 Deprecation Misuse Alternative

Research Question 2: What are the possible ways in which misuse of the deprecation mechanism can be avoided or eliminated?

We describe various alternatives in chapter 4, elaborating upon the advantages in disadvantages of each. We choose to create new annotations for each of the misuse reasons found in **RQ1** and implement them into our own build of a OpenJDK. This has multiple advantages above other methods: developers can make use of our annotations in an idiomatic way, as annotations are already in use in software projects, and integration in the JDK provides a stepping stone towards submitting a Java Enhancement Proposal. If accepted, the work done in this dissertation could be shipped to Java developers and be used in software projects as a drop-in replacement to current deprecation misuse. This helps API producers avoid misuse of the deprecation mechanism by providing them with an alternative that more clearly communicates certain messages to API consumers.

8.1.3 Deprecation Misuse Alternative Effectiveness

Research Question 3: How effective is the mechanism we offer compared to the existing deprecation mechanism?

We conduct a controlled experiment with Java programmers, where the control group sees existing cases of deprecation misuse in libraries currently in use by many developers, and the experimental group encounters our proposed alternatives in its place.

With 25 participants in diverse settings, we test `@Beta` and `@AlwaysThrowsException` in an implementation task and maintenance task. Using these participants, we find no significant difference between the control group and the experimental group when it comes to dissuading API consumers from its usage. We suspect that the current manner of signalling in IntelliJ may not be effective enough compared to other warning types, as other warnings were more often resolved. We also hypothesize that the strikethrough manner of signalling is too intertwined with the concept of deprecation, as participants name deprecation

¹<https://www.scala-lang.org/api/2.12.2/scala/collection/AbstractSeq.html>

²<https://kotlinlang.org/docs/reference/visibility-modifiers.html>

despite our alternative being used. Furthermore, we find that participants who observe the definition of an annotated element are likely to also refactor this element, assisted by documentation located near the definition.

8.2 Impact and Implications

This work is novel in exploring the reasons why the deprecation mechanism is misused in Java in detail. For researchers, the taxonomy we give in this research furthers the understanding we have of such an essential functionality in the Java programming language. We provide a valuable dataset for deprecation usage across many open-source projects [14], giving researchers a corpus in which more research can be done into deprecation. For industry practitioners, the knowledge in this body of work is important in understanding the concept of IDE signalling, and the effectiveness of warning text. We hope that this work push efforts towards improving the Java programming language and can reach both API producers and API consumers, eliminating bugs and reducing costs of software maintenance.

Furthermore, we believe that the experimental environment presented in this dissertation will be of enormous value to the field of software engineering. The ease in which very realistic experimental settings can be offered is invaluable for developer user studies. We provide the work in this these for the research community, anticipating an enthusiastic response from researchers.

8.3 Threats to Validity

We examine the factors within this body of work that reduce the generalizability of these results. In the interest of future similar work, we give recommendations for research expanding on this work.

8.3.1 Misuse Categorization Construct Validity

In this section, we take a critical look at the method used to find and categorize deprecation misuse cases in order to recognize where our study may be short-reaching and to list factors future work should be recognizant of when using this thesis as a basis.

Dataset construction The selection method used does not accurately represent the entirety of the Java ecosystem. The main limitation of using Sawant and Bacchelli [57]’s dataset for this purpose is that it does not accurately represent code hosted on other platforms, using build tools other than Maven, or private code used in industry. Furthermore, we bias our results by only taking the most depended on APIs, skewing representation towards popular APIs. The results were reduced once more by including only libraries that included source code archives.

Sample size Of the 306.529 total deprecated methods across all versions we use as a dataset, only the introduction of the deprecation property was considered. The intuition is that a method would only become deprecated with a reason of misuse, as opposed to being deprecated as the intended use, then evolving towards a misuse. However, this intuition was not confirmed formally.

Sample bias Though we attempt to diversify the samples we take from the dataset, categorizing 763 out of 13,137 methods (5.8%) is by far not a total view of the actual data. Therefore, we make no assertion that the categories of deprecation misuse we find are all possible reasons that exist. However, we find no new misuses in Sawant et al. [60]’s work on deprecation reasons among 374 deprecated features over 1,1100 documents.

External information Our approach uses only information provided in the source code archives of these libraries. However, alternative methods of communication are often used in software projects, in which more detailed reasoning could have been provided for deprecation that was not provided in documentation surrounding the method. A more involved approach could investigate these communication channels for more reasons.

Categorization validity The categories of deprecation misuse were created ad-hoc during analysis of the methods and related documentation. This was done by a single person as opposed to a collaborative approach, meaning that categories may be different in a replication study. In the future, a collaborative approach should be used, using an exploratory method to create categories and a card sorting method to ensure fair categorization of misuse cases.

8.3.2 User Study Validity

A user study can have many hidden pitfalls. As such, it is imperative to be critical of any limitations and biases we may have introduced. In this section, we examine threats to validity in this experimental design. As our experiment has a large human factor, we look at the discipline of psychology and human-computer interaction to understand the limitations of our user study.

Environmental differences This experiment is conducted in a specific environment, using IntelliJ IDEA as the IDE. It could be that a participant misses our alternative annotations because they are not used to the way IntelliJ shows warnings. We attempt to control for this by asking a participant's preferred IDE, as a check to find out if a disparity exists between users familiar with IntelliJ or not. Though a large majority of participants reported to prefer IntelliJ IDEA, it could be that the participants more familiar with other IDEs skewed results.

Previous Knowledge of Participants In particular, we examine the task containing `ImmutableList`. We ask participants to add items to this list, assuming previous knowledge related to the immutability of this list. However, we see that a high number of participants still choose to call a mutating function on this list. This could be due to missing previous knowledge by these participants: the concept of an immutable list could be unclear, causing participants to believe that calling `add` on this list would work as it would with any other list. In a follow-up study, we recommend using an API that participants find easier to use, ensuring that results are not affected by this difference in knowledge.

Novelty effect The novelty effect [47, 48] is a cognitive bias describing the relation between the novelty of a subject under test and a participant's behavior. In a control study, this effect can cause participants to have a stronger response in an experimental group, as the interest or unfamiliarity for the novel entices the participants to act differently in the experimental setting. As novelty effects wear off after repeated usage, a larger-scale user study should expose participants to the new annotations for a longer time.

Generalizability of sampled population The population of participants is not representative of the entirety of Java programmers. In particular, we find that a high number of participants were students. Though studies show that the performance of students and industry professionals is comparable [56, 26], we recognize that these results still might not be an accurate representation. As such, the results we find in this study may not be true in general. Future research should repeat this experiment with a larger audience, allowing for more certainty in the generalizability of the results.

Social desirability This bias has to do with participants giving socially desirable responses instead of what would occur in a real environment [24]. This results in over-reporting responses that are desirable and under-reporting responses that are less desirable. Grimm [24] suggest that survey methods that allow a researcher to identify an individual are most susceptible to social desirability bias. As such, we attempt to lower the need to perform desirably by taking the experiment anonymously.

Hawthorne effect Related to the social desirability bias, the Hawthorne effect [38, 39] refers to a modification in behavior due to being aware of being observed. In this study, this could cause participants to be more attentive for deprecation warnings than they otherwise would be in a realistic setting. We attempt to lower this effect by removing a human factor: the participant conducts this experiment with just their computer instead of the experiment being conducted under the supervision of a researcher.

Coding errors We use response coding to find reasons for a participant's motivation to react or not to react to the warnings they were shown. Codifying responses is an error-prone task [55], influenced by a coder's interpretations and internal biases. This coding was conducted by the author without a

cross-verification via a coding by a different researcher. We recognize this as a flaw to the given encoding, and in a more optimal study, we advise the use of multiple coders to lower biases and differences in possible codings.

Chapter 9

Conclusions and Future Work

This final chapter lists this dissertation’s contributions and gives possible avenues for future work.

9.1 Contributions

This thesis brings multiple contributions to the field of software engineering. In this section, we describe the takeaways of each contribution and elaborate on the value of the findings.

9.1.1 Taxonomy of Deprecation Misuse

In this dissertation, we create a dataset of usages of deprecation and publish it as an artifact in the interest of future deprecation research. Using this dataset, we present a taxonomy of deprecation misuse found in open-source projects. We analyze a sample of 763 methods, discovering reasons why API producers misuse deprecation features in open-source projects. From this analysis, we discovered the following five deprecation misuse categories:

- **UNSUPP:** deprecation used to signal that this method will always throw an `UnsupportedOperationException`, sometimes used when an implementation of an interface is incomplete;
- **TEMP:** deprecation used to indicate that this element is temporary; introduced with the intention of being removed in the future as opposed to having value that is reduced over time;
- **BETA:** deprecation used to indicate that the details of a feature are not completely finalized and are subject to change in the future;
- **INTERNAL:** deprecation used to warn API consumers that this feature should not be used because it is meant to be exclusively for use by the API producer only;
- **DONTCALL:** deprecation used to warn consumers that this method should be never be called for reasons other than always throwing an exception.

Using these reasons, we create two main categories of reasons why developers misuse the deprecation functionality: to communicate peculiarities concerning API versioning and to work around programming language deficiencies.

We believe that this taxonomy will guide future proposals in deprecation research, providing a foundation for future research on this topic and giving Java language developers an insight into how the deprecation feature is being used in real-world projects.

9.1.2 Deprecation Misuse Alternative Proposals

We present multiple options to provide API producers with meaningful ways to communicate aside from using the `@Deprecated` augmented with documentation, and evaluate each method's advantages and disadvantages. We propose three ways of introducing an alternative: augmenting the JDK with additional annotations, creating an external library with additional annotations, and introducing a Javadoc tag.

We chose to augment the JDK, keeping eventual introduction into the JDK in mind. Should the utility be shown in the results of this dissertation, then this way allows easier integration into the JDK, as the implementation of these annotations can be used without much modification. For each of the deprecation misuse reasons we found, a new annotation was created that aims to communicate this misuse reason more clearly.

Furthermore, a plugin was made in IntelliJ IDEA to support these annotations. This plugin analyzes Java code that is open in the editor and warns the user when a code fragment makes use of a fragment annotated with one of these annotations. A warning message is provided in the same way that is done for deprecation, giving the user a clear understanding of what can be expected when using this code.

This work provides the foundations in introducing these annotations to developers in the JDK as well as creating an IntelliJ IDEA plugin aimed to assist developers when encountering these annotations in APIs they use.

9.1.3 Deprecation Misuse Alternative User Study

We present an experimental design for testing our implementation. As the annotation is meant to provide developers with information, it is imperative to conduct a user study to evaluate the usefulness of the new annotations compared to deprecation misuse. We create a controlled experiment which attempts to measure any difference between user dissuasion in the control group, i.e. participants using fragments with deprecation misuse.

Our results show that our participants displayed no significant difference in behavior between encountering a misuse of deprecation versus an alternative that we propose. We hypothesize that current methods of signaling deprecation are not visually indicative enough of the problem severity, especially for method usages that will always throw an exception.

9.1.4 RESPIRED: Distributed Experiment Platform

We present a distributed experiment platform, meant for researchers planning to create a (control) experiment in the field of software technology. We strongly feel that the field of software engineering will benefit from this contribution, saving researchers from reinventing the wheel and providing them with a powerful environment that can mirror a real-world development scenario. The platform makes use of modern technology to allow participants to cooperate in a user study remotely using their own hardware, removing hassle in organizing spaces and devices. Furthermore, this platform is outfitted with utilities such as screen recording and data analysis libraries, making the prospect of conducting a controlled experiment less daunting.

9.2 Future Work

This thesis is another brick in the construction of our understanding of deprecation and its intricate impact on both API consumers and API producers. Though this work examines yet another facet of deprecation, namely that of misuse, the topic could benefit from following works and future research. In this section, we give possible suggestions for approaches to be used in future work.

Verification and replication In this study, we have only a limited pool of participants and open-source projects that we could use for categorisation and for observation. As with many user-studies, the

experiment should be repeated on industry practitioners to verify similar patterns in different contexts. Furthermore, we should consider the possibility that industry software projects display other types of misuse undiscovered by this study. Furthermore, we use a sampling strategy to analyze deprecated methods. It could be possible that

Misuse alternative alternatives In section 4.1, we describe alternative implementations that we could offer consumers. Future research could investigate the effects of these other suggestions or propose new alternatives, testing these according to a similar method. In particular, a generic warning mechanism could be promising, allowing an API producer to specify their own reason why the warning is being placed instead of having to pick from a preselect group of reasons.

JCP Should the results of this dissertation prove convincing enough for the JCP Executive Committee, work should be done to introduce these annotations into the Java Programming language. Though this is an involved process, API producers would be able to replace their current and future misuse cases with the officially supported mechanism.

9.3 Conclusions

In this dissertation, we explore the concept of API deprecation misuse, categorizing the reasons why API producers introduce this in their libraries. We find five different reasons why this is done and group them under API versioning peculiarities and overcoming Java language deficiencies. Furthermore, we create an alternative for this deprecation misuse in the form of five annotations introduced into the JDK. We introduce IDE support in IntelliJ IDEA, creating a warning mechanism similar to the current deprecation mechanism, but with a different warning message. We create an experiment platform, aimed to improve the state-of-the-art of software engineering experiments, and use it to test our proposed alternative against existing misuses of deprecation. We find no significant difference between the implementation behaviors or maintenance behaviors between the two groups, and hypothesize that developers do not respond strongly to current deprecation signalling in IntelliJ IDEA.

Bibliography

- [1] (ann) Add @Unsupported annotation. URL <https://bugs.openjdk.java.net/browse/JDK-6447051>.
- [2] List of Java Inspections - Help — IntelliJ IDEA, . URL <https://www.jetbrains.com/help/idea/list-of-java-inspections.html>. Last visited 2019-10-28.
- [3] IntelliJ Platform SDK, . URL <http://www.jetbrains.org/intellij/sdk/docs/>.
- [4] JVM Ecosystem Report 2018. URL <https://res.cloudinary.com/snyk/image/upload/v1539774333/blog/jvm-ecosystem-report-2018.pdf>.
- [5] StackOverflow: Android BitmapDrawable Constructor Undefined. URL <https://stackoverflow.com/questions/4904660/android-bitmapdrawable-constructor-undefined>.
- [6] CVE - Common Vulnerabilities and Exploits. URL <https://cve.mitre.org/>.
- [7] Enterprise Container Platform — Docker. URL <https://www.docker.com/>.
- [8] Web Archive: JDK 1.0.2 api documentation. URL <http://web.archive.org/web/19990202194011/http://java.sun.com/products/jdk/1.0.2/api/>.
- [9] JEP 277: Enhanced Deprecation. URL <http://openjdk.java.net/jeps/277>.
- [10] StackOverflow. URL <https://stackoverflow.com/>.
- [11] JEP 1: JDK Enhancement-Proposal & Roadmap Process, 2019. URL <https://openjdk.java.net/jeps/1>.
- [12] Rahul Amlekar, Andrés Felipe Rincón Gamboa, Keheliya Gallaba, and Shane McIntosh. Do software engineers use autocompletion features differently than other developers? In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 86–89. IEEE, 2018.
- [13] Victor R Basili, Lionel C Briand, and Walcélio L Melo. How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39(10):104–116, 1996.
- [14] Dereck J Bridie, Anand Sawant, and Alberto Bacchelli. A sample of all usages of Java’s Deprecation functionality found in open-source Maven Central projects with at least 2000 dependents. 2019. doi: 10.4121/uuid:ea6b1511-da36-4d16-868e-401b461eafb4. URL <http://doi.org/10.4121/uuid:ea6b1511-da36-4d16-868e-401b461eafb4>.

- [15] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 360–369. IEEE, 2016.
- [16] Raymond PL Buse and Westley R Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130. ACM, 2008.
- [17] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. Glacier: Transitive class immutability for Java. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 496–506. IEEE, 2017.
- [18] Theo Combe, Antony Martin, and Roberto Di Pietro. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Computing*, 3(5):54–62, 2016.
- [19] Danny Coward. JSR 175: A Metadata Facility for the JavaTM Programming Language. *Java Community Process*. <https://www.jcp.org/en/jsr/detail>, pages 3–23, 2004.
- [20] Danny Dig and Ralph Johnson. The role of refactorings in API evolution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389–398. IEEE, 2005.
- [21] PHP Documentation. Error Handling Predefined Constants. URL <https://www.php.net/manual/en/errorfunc.constants.php>.
- [22] Eclipse Foundation. Java Compile Errors/Warnings Preferences. URL https://help.eclipse.org/2019-09/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fpreferences%2Fjava%2Fcompiler%2Fref-preferences-errors-warnings.htm&cp=1_4_4_0_3_1.
- [23] John G Geer. Do open-ended questions measure salient issues? *Public Opinion Quarterly*, 55(3): 360–370, 1991.
- [24] Pamela Grimm. Social desirability bias. *Wiley international encyclopedia of marketing*, 2010.
- [25] Michael Hoerger. Participant dropout as a function of survey length in internet-mediated university studies: Implications for study design and voluntary participation in psychological research. *Cyberpsychology, behavior, and social networking*, 13(6):697–700, 2010.
- [26] Martin Höst, Björn Regnell, and Claes Wohlin. Using students as subjects a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5(3): 201–214, 2000.
- [27] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3): 90, 2007.
- [28] JetBrains s.r.o. Kotlin experimental api markers. URL <https://kotlinlang.org/docs/reference/experimental.html>.
- [29] Ralph E Johnson and Brian Foote. Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35, 1988.
- [30] Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and IDEs. *ACM sigplan notices*, 45(10):444–463, 2010.

- [31] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks—a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.
- [32] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.
- [33] Mark A Lemley and David W O’Brien. Encouraging software reuse. *Stanford Law Review*, pages 255–304, 1997.
- [34] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *proceedings of the 22nd International Conference on Program Comprehension*, pages 83–94. ACM, 2014.
- [35] Richard C Lupton and Julian M Allwood. Hybrid Sankey diagrams: Visual analysis of multidimensional data for understanding resource use. *Resources, Conservation and Recycling*, 124:141–151, 2017.
- [36] Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. Docker ecosystem–vulnerability analysis. *Computer Communications*, 122:30–43, 2018.
- [37] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.
- [38] Elton Mayo. *The human problems of an industrial civilization*. Routledge, 2004.
- [39] Rob McCarney, James Warner, Steve Iliffe, Robbert Van Haselen, Mark Griffin, and Peter Fisher. The Hawthorne Effect: a randomised, controlled trial. *BMC medical research methodology*, 7(1):30, 2007.
- [40] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.
- [41] Michael C Medlock, Dennis Wixon, Mark Terrano, Ramon Romero, and Bill Fulton. Using the RITE method to improve products: A definition and a case study. *Usability Professionals Association*, 51, 2002.
- [42] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming Q&A in StackOverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34. IEEE, 2012.
- [43] Oracle. Java Language Specification 9.6.4.6. Deprecated, . URL <https://docs.oracle.com/javase/specs/jls/se13/html/jls-9.html#jls-9.6.4.6>.
- [44] Oracle. Enhanced Deprecation, . URL <https://docs.oracle.com/en/java/javase/13/core/enhanced-deprecation1.html>.
- [45] Oracle. How and When To Deprecate APIs, . URL <https://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/deprecation/deprecation.html>.
- [46] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46:1155–1179, 2015. doi: 10.1002/spe.2346. URL <https://hal.archives-ouvertes.fr/hal-01078532/document>.

- [47] John Pisapia, Jeanne Schlesinger, and Amanda Parks. Learning technologies in the classroom: Review of the literature. 1993.
- [48] Jordan Poppenk, Stefan Köhler, and Morris Moscovitch. Revisiting the novelty effect: When familiarity, not novelty, enhances memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 36(5):1321, 2010.
- [49] Roel Popping. Analyzing open-ended questions by means of text analysis procedures. *Bulletin of Sociological Methodology/Bulletin de Méthodologie Sociologique*, 128(1):23–39, 2015.
- [50] Dong Qiu, Bixin Li, and Hareton Leung. Understanding the API usage in Java. *Information and software technology*, 73:81–100, 2016.
- [51] Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 215–224. IEEE, 2014.
- [52] Ulf-Dietrich Reips. Internet experiments: Methods, guidelines, metadata. In *Human Vision and Electronic Imaging XIV*, volume 7240, page 724008. International Society for Optics and Photonics, 2009.
- [53] Romain Robbes, Mircea Lungu, and David Röthlisberger. How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem. In *FSE’12*, 2012. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393662.
- [54] Martin P Robillard and Robert Deline. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.
- [55] Julius A Roth. Coding responses to open-ended questions. *Sociological Methodology*, 3:60–78, 1971.
- [56] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. Are students representatives of professionals in software engineering experiments? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 666–676. IEEE, 2015.
- [57] Anand Ashok Sawant and Alberto Bacchelli. A dataset for API usage. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 506–509. IEEE Press, 2015.
- [58] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs. In *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016*, 2017. ISBN 9781509038060. doi: 10.1109/ICSM.2016.64.
- [59] Anand Ashok Sawant, Mauricio Aniche, Arie van Deursen, and Alberto Bacchelli. Understanding developers’ needs on deprecation as a language feature. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 561–571. IEEE, 2018.
- [60] Anand Ashok Sawant, Guangzhe Huang, Gabriel Vilen, Stefan Stojkovski, and Alberto Bacchelli. Why are features deprecated? An investigation into the motivation behind deprecation. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 13–24. IEEE, 2018.
- [61] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. An empirical study on the impact of C++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering*, pages 760–771. ACM, 2016.

- [62] Michael Waskom, Olga Botvinnik, Drew O’Kane, Paul Hobson, Saulius Lukauskas, David C Gemperline, Tom Augspurger, Yaroslav Halchenko, John B. Cole, Jordi Warmenhoven, Julian de Ruyter, Cameron Pye, Stephan Hoyer, Jake Vanderplas, Santi Villalba, Gero Kunter, Eric Quintero, Pete Bachant, Marcel Martin, Kyle Meyer, Alistair Miles, Yoav Ram, Tal Yarkoni, Mike Lee Williams, Constantine Evans, Clark Fitzgerald, Brian, Chris Fonnesebeck, Antony Lee, and Adel Qalieh. mwaskom/seaborn: v0.8.1 (september 2017), September 2017. URL <https://doi.org/10.5281/zenodo.883859>.
- [63] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 351–361. IEEE, 2016.
- [64] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147. IEEE, 2017.
- [65] Jing Zhou and Robert J Walker. API deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 266–277. ACM, 2016.

Appendix A

Code fragments

Listing A.1: File to be maintained by the user

```
1 import org.apache.commons.io.FileUtils;
2
3 import java.io.*;
4 import java.util.ArrayList;
5 import java.util.LinkedList;
6 import java.util.Iterator;
7 import java.util.List;
8
9 class GradeItem implements Serializable {
10     private String courseCode;
11     private double mark;
12     private int studentID;
13
14     public GradeItem(String courseCode, double mark, int studentID) {
15         this.courseCode = courseCode;
16         this.mark = mark;
17         this.studentID = studentID;
18     }
19
20     public String getCourseCode() {
21         return courseCode;
22     }
23
24     public void setCourseCode(String courseCode) {
25         this.courseCode = courseCode;
26     }
27
28     public double getMark() {
29         return mark;
30     }
31
32     public void setMark(double mark) {
33         this.mark = mark;
34     }
35
36     public int getStudentID() {
37         return studentID;
38     }
39 }
```

```

40     public void setStudentID(int studentID) {
41         this.studentID = studentID;
42     }
43
44     @Override
45     public String toString() {
46         return "GradeItem{" +
47             "courseCode='" + courseCode + '\'' +
48             ", mark=" + mark +
49             ", studentID=" + studentID +
50             '}';
51     }
52 }
53
54
55 public class GradeAdministration {
56     List<GradeItem> grades;
57     public GradeAdministration(List<GradeItem> grades) {
58         this.grades = grades;
59     }
60
61     /**
62      * Calculates a grade average for a student identified by an ID.
63      * This method should return null if there is no student by this ID.
64      */
65     public Double calculateAverageGradeForStudent(int studentId) {
66         List<GradeItem> studentItems = new ArrayList<>();
67         for (GradeItem item : grades) {
68             if (item.getStudentID() == studentId) {
69                 studentItems.add(item);
70             }
71         }
72         if (studentItems.size() == 0) { return null; }
73         else {
74             double sum = 0.0;
75             for (GradeItem item : studentItems) {
76                 sum += (double) item.getMark();
77             }
78             return sum / ((double) studentItems.size());
79         }
80     }
81
82     /**
83      * Calculates a grade average for a course identified by a course code.
84      * The method should return null if there is no course code by this ID.
85      */
86     public Double calculateAverageGradeForCourse(String courseCode) {
87         List<GradeItem> courseItems = new ArrayList<>();
88         for (GradeItem item : grades) {
89             if (item.getCourseCode().equals(courseCode)) {
90                 courseItems.add(item);
91             }
92         }
93
94         if (courseItems.size() == 0) { return null; }
95

```

```

96     double sum = 0.0;
97     for (GradeItem item : courseItems) {
98         sum += item.getMark();
99     }
100    return sum / ((double) courseItems.size());
101 }
102
103 public int getUniqueCoursesTotal() {
104     List<String> foundCourseCodes = new ArrayList<>();
105     for (GradeItem item : grades) {
106         if (!foundCourseCodes.contains(item.getCourseCode())) {
107             foundCourseCodes.add(item.getCourseCode());
108         }
109     }
110     return foundCourseCodes.size();
111 }
112
113 public List<Integer> getAllStudentsInCourse(String courseCode) {
114     List<GradeItem> courseItems = new ArrayList<>();
115     for (GradeItem item : grades) {
116         if (item.getCourseCode().equals(courseCode)) {
117             courseItems.add(item);
118         }
119     }
120
121     if (courseItems.size() == 0) return null;
122
123     List<Integer> students = new ArrayList<>();
124     for (GradeItem item : courseItems) {
125         if (!students.contains(item.getStudentID())) {
126             students.add(item.getStudentID());
127         }
128     }
129
130     return students;
131 }
132
133 /**
134  * Writes all grades in this administration to a file.
135  */
136 public void writeToFile(File file) {
137     try {
138         StringBuilder builder = new StringBuilder();
139         builder.append("courseCode\tmark\tstudentID\n");
140         Iterator<GradeItem> iterator = grades.iterator();
141         while (iterator.hasNext()) {
142             GradeItem item = iterator.next();
143             builder.append(item.getCourseCode() + "\t" + item.getMark()
144                 + "\t" + item.getStudentID() + "\n");
145         }
146         FileUtils.writeStringToFile(file, builder.toString());
147     } catch (IOException e) {
148         System.out.println(e);
149     } catch (RuntimeException e) {
150         System.out.println(e);
151     }

```

```

151     }
152
153     /**
154      * Adds an item to this grade administration.
155      * This method should make sure that this item does not have a null
156      * course code.
157      */
158     public void addGrade(GradeItem item) {
159         if (item.getCourseCode() != null) {
160             grades.add(item);
161         }
162     };
163
164     public boolean hasGradeFor(String courseCode, int studentID) {
165         boolean foundGradeFor = false;
166         for (GradeItem grade : grades) {
167             if (grade.getCourseCode().equals(courseCode) && studentID ==
168                 grade.getStudentID()) {
169                 foundGradeFor = true;;
170             }
171         }
172         return foundGradeFor == true;
173     }

```