Detecting Missing Method Calls in Object-Oriented Software

Martin Monperrus, Marcel Bruch, Mira Mezini

▶ **To cite this version:**

Martin Monperrus, Marcel Bruch, Mira Mezini. Detecting Missing Method Calls in Object-Oriented Software. Proceedings of the 24th European Conference on Object-Oriented Programming, 2010, Maribor, Slovenia. pp.2-25, 10.1007/978-3-642-14107-2_2 . hal-01575351

HAL Id: hal-01575351
https://hal.science/hal-01575351

Submitted on 20 Sep 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Detecting Missing Method Calls in Object-Oriented Software

Martin Monperrus Marcel Bruch Mira Mezini

Technische Universität Darmstadt
{monperrus,bruch,mezini}@cs.tu-darmstadt.de

Abstract. When using object-oriented frameworks it is easy to overlook certain important method calls that are required at particular places in code. In this paper, we provide a comprehensive set of empirical facts on this problem, starting from traces of missing method calls in a bug repository. We propose a new system, which automatically detects them during both software development and quality assurance phases. The evaluation shows that it has a low false positive rate (<5%) and that it is able to find missing method calls in the source code of the Eclipse IDE.

1 Introduction

"Thanks for letting me know about [...] the missing method call". This was written by a programmer on an Internet forum[1]. This quote indicates that missing method calls may be the source of software defects that are not easy to detect without assistance. Actually, problems related to missing method calls pop up in forums[1], in newsgroups[2], in bug reports[3], in commit texts[4], and in source code[5]. For a more systematic analysis of the problem, we performed a comprehensive study in a well-delimited scope: the Eclipse Bug Repository contains at least 115 bug reports related to missing method calls (cf. section 2.2). The analysis shows that issues caused by missing method calls are manifold[6]: they can produce obscure runtime exceptions at development time, they can be responsible of defects in limit cases, and they generally reveal code smells. These observations have motivated the work presented in this paper.

Our intuition is that missing method calls are a kind of *deviant code*. Previous research proposed different characterizations of *deviant code*. Engler et al. [1] and Li et al. [2] proposed two different characterizations for procedural system-level code. Livshits et al. [3] characterized deviant code as instance of error patterns highlighted by software revisions. Wasylkowski et al. [4] described an

[1] http://www.velocityreviews.com/forums/t111943-customvalidator-for-checkboxes.html
[2] http://dev.eclipse.org/mhonarc/newsLists/news.eclipse.tools/msg46455.html
[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=222305
[4] http://mail.eclipse.org/viewcvs/index.cgi/org.eclipse.team.core/src/org/eclipse/team/core/mapping/provider/MergeContext.java?view=log
[5] http://mail.eclipse.org/viewcvs/index.cgi/equinox-incubator/security/org.eclipse.equinox.security.junit/src/org/eclipse/equinox/security/junit/KeyStoreProxyTest.java?view=co
[6] These issues are further discussed in this paper

In: Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'2010)
This preprint is provided by the contributing authors to ensure timely dissemination of scholarly and technical work.

approach based on mining usage patterns and their violations. However, as we will further elaborate in section 6, the aforementioned proposals either are not dedicated to object-oriented code and subsequently to missing method calls, or suffer from scalability issues, or have a high rate of false positives.

This paper presents *a new characterization of deviant code* suitable to detect missing method calls. The pieces of code that we consider are *type-usages*. A type-usage is a list of method calls on a variable of a given type occurring somewhere within the context of a particular method body. Our characterization of deviant code is done on top of two new definitions of similarity for type-usages, and we propose a new metric called *S-Score* to measure the degree of deviance w.r.t. missing method calls. Our tool produces warnings for type-usages whose *S-Score* is high. Hence, we classify our tool as a code warning tool according to the definition of Robillard [5]: it *"helps identify elements that are likely to be more worthy of investigation than others"*. The tool is a **D**etector of **M**issing **M**ethod **C**all, which grounds its acronymic name: DMMC.

We use different techniques to evaluate the proposed approach. First, statistical methods are used to show that our characterization of deviant code makes sense for detecting missing method calls. Second, we propose and perform a quantitative evaluation based on the simulation of defects by degrading real software. The advantage of this evaluation technique is that it can be fully automated on a large scale while still involving likely defects. This evaluation technique shows that our approach produces less than 2% of false positives (out of +50000 simulated missing calls), a result that does not fit the findings by Kim et al. [6] that usually, *"automatic bug-finding tools have a high false positive rate"*. One might suspect that the low false positive rate is due to our process of artificially creating missing method calls, which might not well simulate real missing method calls of real software. To ensure that this is not the case, our last evaluation technique was to apply the tool to reveal problems related to missing method calls in real software: the user-interface part of the Eclipse IDE codebase (44435 type-usages of `org.eclipse.swt.*` out of 1847431 LOC). We analyzed 19 high-confidence warnings found by our tool and filed the corresponding bug reports if appropriate: 8 of them are already fixed in the latest version of the Eclipse codebase. The results of this last evaluation confirm the findings of the automatic quantitative evaluation.

To summarize, the contributions of this paper are:

- A comprehensive set of empirical facts on the problems caused by missing method calls. We present +30 examples of real software artifacts affected by missing method calls, a comprehensive study of this problem in the Eclipse Bug Repository, and an extensive analysis of the missing calls that our tool found in Eclipse (including an analysis of their causes and their solutions).
- A new characterization of *deviant code* dedicated to missing method calls. This new characterization advances the state-of-the-art especially in terms of rate of false positive.
- A new strategy to evaluate code warning tools, based on the simulation of defects by degrading real software.

In: Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'2010)
This preprint is provided by the contributing authors to ensure timely dissemination of scholarly and technical work.

The reminder of the paper is structured as follows. In section 2, we elaborate on empirical facts about missing method calls. Section 3 presents our approach and the underlying algorithm. Section 4 presents the evaluation strategy and results. Section 5 discusses the integration of the approach in the software development process. Related work is discussed in section 6. Section 7 concludes the paper and sketches areas of future work.

2 The Importance of Detecting Missing Method Calls

This section presents empirical facts supporting the following claims: (a) problems related to missing method calls do happen in practice and can be difficult to understand, and (b) they survive development time.

2.1 Problems Related to Missing Calls are Real and Hard to Understand

Let us tell a little story that shows that missing method calls are likely and can be the source of real problems. The story is inspired from several real world posts to Internet forums and mailing lists[7]. Sandra is a developer who wants to create a dialog page in Eclipse. She finds a class corresponding to her needs in the API named `DialogPage`. Using the new-class-wizard of Eclipse, she automatically gets a code snippet containing the methods to override, shown below:

```
public class MyPage extends DialogPage {
 @Override
 public void createControl(Composite parent) {
  // TODO Auto-generated method stub
 }
}
```

Since the API documentation of `DialogPage` does not mention special things to do, Sandra writes the code for creating a control, a `Composite`, containing all the widgets of her own page. Sandra knows that to register a new widget on the UI, one passes the parent as parameter to the Composite constructor.

```
 public void createControl(Composite parent) {
   Composite mycomp = new Composite(parent);
   ....
 }
```

Sandra get the following error message at the first run of her code (the error log is unfortunately empty)!

```
An error has occurred. See error log for more details.
org.eclipse.core.runtime.AssertionFailedException
null argument:
```

[7] e.g. http://dev.eclipse.org/mhonarc/newsLists/news.eclipse.tools/msg46455.html and http://dev.eclipse.org/newslists/news.eclipse.platform.rcp/msg10075.html

When extending a framework class, there are often some contracts of the form "*call method x when you override method y*", which need to be followed. The Eclipse JFace user-interface framework expects that an application class extending `DialogPage` calls the method `setControl` within the method that overrides the framework method `createControl`. However, the documentation of `DialogPage` does not mention this implicit contract; Sandra thought that registering the new composite with the parent is sufficient.

The described scenario pops up regularly in the Eclipse newsgroup[8] and shows that one can easily fail to make important method calls. Furthermore, the resulting runtime error that Sandra got is really cryptic and it may take time to understand and solve it.

Sandra had to ask a question on a mailing list to discover that this problem comes from a missing call to `this.setControl`. After the addition of `this.setControl(mycomp)` at the end of her code, Sandra could finally run the code and commit it to the repository; yet, she lost 2 hours in solving this bug related to a missing method call.

2.2 Missing Method Calls Survive Development Time

Missing method calls are not all detected before committing code to the version repository. To support this claim, we have searched for bug descriptions related to missing method calls in the Eclipse Bug Repository[9].

Our search process went through the following steps: 1) establish a list of syntactic patterns which could indicate a missing method call, 2) for each pattern of the list created in the previous step, query the bug repository for bug descriptions matching the pattern 3) read the complete description of each resulting bug report to assess whether it is really related to missing method calls.

To know that a report is really due to a missing method call or not, we read the whole sentence or paragraph containing the occurrence of the syntactic pattern. This gives a clear hint to assess whether this is a true or a false positive. For instance, bug #186962 states that *"setFocus in ViewPart is not called systematically"*: it is validated as related to missing method call; bug #13478 mentions that *"CVS perspective should be called CVS Repository Exploring"*: it is a false positive.

Table 1 summarizes the results. For illustration consider the numbers in the first raw, which tell that 49 bug reports contain the syntactic pattern "should call", and 26 of them are actually related to missing method calls. In all 211 bug reports are found by the syntactic patterns we have used, and 117 of them are actually related to missing method calls. This number shows that missing method call survive development time, especially if we consider that the number is probably an underestimation, since we may have missed other syntactic patterns. Indeed, we will also show in the evaluation section that we are able to find other missing method calls in Eclipse.

[8] cf. the Google results of "`setcontrol+site:http://dev.eclipse.org/mhonarc/newsLists/`")

[9] `http://bugs.eclipse.org`

Pattern	Matched	Confirmed
“should call”	49	26 (53%)
“does not call”	39	28 (72%)
“is not called”	36	26 (72%)
“should be called”	34	9 (26%)
“doesn’t call”	16	13 (81%)
“do not call”	10	6 (60%)
“are not called”	7	0 (0%)
“must call”	7	4 (57%)
“don’t call”	6	2 (33%)
“missing call”	6	2 (33%)
“missing method call”	1	1 (100%)
Total	211	117 (55%)

Table 1. The number of bug reports in the Eclipse Bug Repository per syntactic pattern related to missing method calls. The second column shows the number of occurrences of the pattern, the third one is the number of bug reports that are actually related to missing method calls after manual inspection.

2.3 Recapitulation

These empirical facts show that a detector of missing method calls: (a) can help programmers like Sandra write better code in a shorter time, and (b) can help maintainers solve and fix bugs related to missing method calls. Also, from a quality assurance perspective, such a code warning tool lists places in code that are likely to contain missing method calls and that are worth being investigated before they produce a real bug or hinder maintenance.

3 The DMMC System

The DMMC system is a missing method call detection system. It operates statically by analyzing software source code and outputting a list of places in code where there may be problems due to missing method calls. Our intuition is that a piece of code is likely to host defects if there are few similar pieces of code and a lot of slightly different pieces of code[10].

Let us consider an analogy for illustrating the idea. In a restaurant, there is one place p with one fork and one spoon. In the whole restaurant, there is a single other place with one fork and one spoon, i.e. there is one similar but not identical other place (the color of the spoon may change). However, there are 99 other places with one fork, one spoon, and one knife. It is very likely that there is an issue with place p, which can be formulated as "*A knife is missing*".

[10] This idea assumes two different definitions of similarity on code (so far called “similar” and “slightly different”). In contrast, standard detection of code duplicates only involves one definition of similarity.

The rest of this section adapts this idea to object-oriented software. We explain the foundations of our system in natural language before formalizing them using set theory and first-order logic.

3.1 Overview

The pieces of code we consider in our system are *type-usages*. A type-usage is a list of method calls on a variable of a given type occurring in the body of a particular calling method. Figure 1 shows three different code excerpts that are used to illustrate this definition and the ones that follow. All excerpts are from extensions of class `Page`, more precisely from re-implementations of the inherited method `createButton`; there is one type-usage per code excerpt, all are usages of the class `Button`, i.e. type-usages of `Button`. For instance, the excerpt at the right-hand side contains a type-usage of the form `tu1 = {Button.<init>, Button.setText(), Button.setColor()}`.

```
class A extends Page {
  @Override
  Button createButton() {
    Button b = new Button();
    ...(interlaced code)
    b.setText("hello");
    ...(interlaced code)
    b.setColor(GREEN);
    return b;
  }
}
```

```
class B extends Page {
  @Override
  Button void createButton() {
    ... (code before)
    Button aBut = new Button();
    ...
    aBut.setText("great");
    aBut.setColor(RED);
    return aBut;
  }
}
```

```
class C extends Page {
  Button myBut;
  @Override
  Button void createButton() {
    myBut = new Button();
    myBut.setColor(PINK)
    nyButton.setText("world");
    myBut.setLink("http://bit.ly");
    ... (code after)
    return myBut;
  }
}
```

Fig. 1. Exactly-Similar and Almost-Similar Type-Usages

We have clarified the meaning of type-usages, we can now informally define two measures of similarity between type-usages: *exact-similarity* and *almost-similarity*.

A type-usage is *exactly-similar* to another type-usage if it is used in the method body of a similar method and if it contains the same method calls. For instance, in figure 1 the type-usage in class B (middle excerpt) is exactly-similar to the type usage of class A (left-hand excerpt): (a) they both occur in the body of the method `Button createButton()`, i.e. they are used in the same context (the notion of "context" is completely defined in 3.2), and (b) they both have the same set of method calls. We use the term "similar" to highlight that at a certain level of detail the type-usages related by exact-similarity are different: variables names may be different, interlaced and surrounding code, as well.

A type-usage is *almost-similar* to another type-usage if it is used in a similar context and if it contains the same method calls plus another one. In figure 1 the type-usage in class C (right-hand excerpt) is almost-similar to the type usage of class A (left-hand excerpt): they are used in the same context, but the type-usage in class C contains all methods of A plus another one: `setLink`. We

In: Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'2010)
This preprint is provided by the contributing authors to ensure timely dissemination of scholarly and technical work.

need the term *almost-similar* to denote that the relationship between two type-usages is more similar than different, i.e., there is some similarity, while being not *exactly-similar*.

Our system is built on the assumption that a type-usage is deviant if: 1) it has a small number of other type-usages that are *exactly-similar*. and 2) it has a large number of other type-usages that are *almost-similar*. Informally, a small number of *exactly-similar* means "only few other places do the same thing" and a large number of *almost-similar* means "the majority does something slightly different". Assuming that the majority is right, the type-usage under inspection seems deviant and may reveal an issue in software.

Given the intuitive definitions so far, we are now able to give an overview of the logic of our system:

1. Extract every type-usage x from software;
2. For every type-usage x:
   (a) Search for type-usages that are exactly-similar according to our definition of *exact-similarity*;
   (b) Search for type-usages that are almost-similar according to our definition of *almost-similarity*;
   (c) Compute a measure of strangeness. We call this new measure *the S-score*;
   (d) Synthesize a list of likely missing method calls;
3. Output a list of deviant type-usages ordered by S-score and their missing method calls.

3.2 Extracting Type-Usages

The DMMC system uses the Wala bytecode analysis toolkit[11] to extract type-usages from Java code[12]. For each variable x in the code, we extract the declared type $T(x)$ of the variable containing the type-usage, the context $C(x)$ of `x`, that we define as the whole method signature of the containing method (i.e. name, return type, and parameter types), and the names of the methods invoked on `x` within $C(x)$, $M(x) = \{m_1, m_2, ..., m_n\}$). If there are two variables of the same type in the scope of a method, they are two type-usages extracted.

Figure 2 illustrates the encoding of the extracted type-usages by an example. A code snippet is shown on the left-hand side of the figure; the corresponding extracted type-usages are shown on the right-hand side of the figure. There are two extracted type-usages, for Button `b` and for Text `t`. The context is the overriden method `createButton` for both. The set of invoked methods on `b` is $M(b) = \{< init >, setText, setColor\}$, `t` is just instantiated ($M(t) = \{< init >\}$).

Note that our static analysis is not limited to the same method body. It follows all method calls on `this`, i.e. all calls in the same class hierarchy which can be clearly statically resolved. This allows us to handle facility methods which initialize objects.

[11] http://wala.sf.net

[12] However, the approach can be easily adapted to dynamically typed languages.

This preprint is provided by the contributing authors to ensure timely dissemination of scholarly and technical work.

```
class A extends Page {
  Button b;
  @Override                          T(b) = 'Button'
  Button createButton() {            C(b) = 'Page.createButton()'
    b = new Button();                M(b) = {<init>, setText, setColor}
    b.setText("hello");
    b.setColor(GREEN);               T(t) = 'Text'
    ...(other code)                  C(t) = 'Page.createButton()'
    Text t = new Text();             M(t) = {<init>}
    return b;
  }
}
```

Fig. 2. Extraction Process of Type-Usages in Object-Oriented Software.

3.3 Exactly and Almost Similar Type-usages

We define a relationship E over type-usages of object-oriented source code that expresses that two type-usages x and y are exactly-similar if and only if:

$$\begin{aligned} xEy \iff & T(x) = T(y) \\ & \wedge\, C(x) = C(y) \\ & \wedge\, M(x) = M(y) \end{aligned}$$

We then define for each type-usage x the set of exactly-similar type-usages:

$$E(x) = \{y | xEy\}$$

Note that since the relationship holds for the identity, i.e. xEx is always-valid, $E(x)$ always contains x itself, and $|E(x)| \geq 1$.

We define a relationship A over type-usages of object-oriented source code that expresses that two type-usages are almost-similar. A type-usage x is almost-similar to a type-usage y if and only if:

$$\begin{aligned} xAy \iff & T(x) = T(y) \\ & \wedge\, C(x) = C(y) \\ & \wedge\, M(x) \subset M(y) \\ & \wedge\, |M(y)| = |M(x)| + 1 \end{aligned}$$

For each type-usage x of the codebase, the set of almost-similar type-usages is:

$$A(x) = \{y | xAy\}$$

Note that contrary to $E(x)$, $A(x)$ can be empty and $|A(x)| \geq 0$. Also, it is possible to weaken the definition of almost-similarity by allowing a bigger amount

This preprint is provided by the contributing authors to ensure timely dissemination of scholarly and technical work.

of difference, i.e. $|M(y)| = |M(x)| + k, k \geq 1$. However, we concentrate in this paper, esp. in the evaluation, on $k = 1$. Also, one can object that our definition of context is too restrictive. However, we think that it fits to the nature of object-oriented software where most pieces of code lie inside a class hierarchy. Furthermore, the evaluation section (4) shows that this definition still allows us to help developers in a large number of situations.

3.4 S-score: A Measure of Strangeness for Object-Oriented Software

Now we want to define a measure of strangeness for object-oriented type-usages. This measure will allow us to order all the type-usages of a codebase so as to identify the top-K strange type-usages[13] that are worth being manually analyzed by a software engineer. We define the S-score as:

$$\text{S-score}(x) = 1 - \frac{|E(x)|}{|E(x)| + |A(x)|}$$

This definition correctly handles extreme cases: if there are no exactly-similar type-usages and no almost-similar type-usages for a type-usage a, i.e. $|E(a)| = 1$ and $|A(a)| = 0$, then $S - score(a)$ is zero, which means that a unique type-usage is not a strange type-usage at all. On the other extreme, consider a type-usage b with $|E(b)| = 1$ (no other similar type-usages) and $|A(b)| = 99$ (99 almost-similar type-usages). Intuitively, a developer expects that this type-usage is very strange, may contain a bug, and should be investigated. The corresponding S-score is 0.99 and supports the intuition.

Note that if a type-usage is very far from existing code, i.e. has a very low number of *exactly-similar* and a very low number of *almost-similar*, our approach does not raise a warning. This kind of warning is out of the scope of missing method calls. However, it could easily be detected with a condition like $|E(X)| < k \wedge |A(X)| < k$ where k is low.

3.5 Predicting Missing Method Calls

For the type-usages that are really strange, i.e. that have a very high S-score, the system recommends a list of method calls that are likely to be missing.

Core Algorithm: The recommended method calls $R(x)$ for a type-usage x are those calls present in almost-similar type-usages but missing in x. In other terms:

$$R(x) = \{m | m \notin M(x) \wedge m \in \bigcup_{z \in A(x)} M(z)\}$$

For each recommended method in $R(x)$, the system gives a likelihood value $\phi(m, x)$. The likelihood is the frequency of the missing method in the set of almost-similar type-usages:

[13] K may depend on the size and the maturity of the analyzed software.

This preprint is provided by the contributing authors to ensure timely dissemination of scholarly and technical work.

$$\phi(m,x) = \frac{|\{z|z \in A(x) \wedge m \in M(z)\}|}{|A(x)|}$$

For illustration, consider the example in figure 3. The type-usage under study is x of type `Button`, it has a unique call to the constructor. There are 5 almost-similar type-usages in the source code (a, b, c, d, e). They contain method calls to `setText` and `setFont`. `setText` is present in 4 almost-similar type-usages out of a total of 5. Hence, it's likelihood is $4/5 = 80\%$. In this situation, the system recommends to the developer the following missing method calls: `setText` with a likelihood of 80% and `setFont` with a likelihood of 20%.

Note that our definition also works when $M(x) = \emptyset$ or when x is the current object (i.e. `this` in Java). This is exactly the case of Sandra presented in 2.1. In Sandra's development problem, since our codebase contains many type-usages where $A(this) = \{setControl\}$, our algorithm is able to help Sandra by telling her the missing method call.

$$\begin{aligned}
T(x) &= Button \\
M(x) &= \{< init >\} \\
A(x) &= \{a, b, c, d\} \\
M(a) &= \{< init >, setText\} \\
M(b) &= \{< init >, setText\} \\
M(c) &= \{< init >, setText\} \\
M(d) &= \{< init >, setText\} \\
M(e) &= \{< init >, setFont\}
\end{aligned}$$

$$\begin{aligned}
R(x) &= setText, setFont \\
\phi(setText) &= \frac{4}{5} = 0.80 \\
\phi(setFont) &= \frac{1}{5} = 0.20
\end{aligned}$$

Fig. 3. An example computation of the likelihoods of missing method calls

Variant with Filtering In the example of figure 3, it is much more likely that the missing method call we are searching is `setText` rather than `setFont`, and it seems interesting to set up a threshold t on the likelihood before recommending a missing method call to the user. This defines a filtered set of recommendations $R_f(x)$ which is:

$$R_f(x) = \{m|m \in R(x) \wedge \phi(m,x) > t\}$$

This variant of the system is called DMMC-filter, as opposed to DMMC-core.

4 Evaluation

We propose and conduct an evaluation for the proposed DMMC system, which combines different techniques to validate the system from different perspectives:

- We validate the S-score measure by showing that (a) it is low for most type-usages of real software, i.e. that the majority of real type-usages is not strange (cf. 4.1), and (b) it is able to catch type-usages with a missing method call, i.e. that the S-score of such type-usages is in average higher than the S-Score of normal type-usages (cf. 4.2).
- We show that our algorithm produces good results, i.e., predicts missing method calls that are actually missing (cf. 4.3).
- We evaluate whether our system is able to find meaningful missing method calls in mature software (cf. 4.4).

4.1 The Correctness of the Distribution of the S-Score

We collected the whole Eclipse 3.4.2 codebase for conducting this experiment (564 plugins). To allow future comparisons with other approaches and replication studies, this dataset is publicly available upon request. From this codebase, our static analysis collected 44435 type-usages whose type belongs to the Standard Widget Toolkit (SWT). For each of them, we have computed the sets of exactly and almost-similar type-usages ($E(x)$ and $A(x)$) and their S-score.

Histogram of the S-score Since Eclipse is a real-world and mature software, we assume that most of its type-usages have a low degree of strangeness, i.e. a low S-score. Figure 4 validates the assumption: 78% of the SWT type-usages have indeed a S-score less than 10%, 90% of the SWT type-usages have a S-Score less than 50%. Indeed, the distribution unsurprisingly looks like an exponential distribution, which is regular in software [7].

Representing the S-score in a 2D Space Independently of the S-score, we also assume that most of the type-usages of Eclipse have a low number of almost-similar type-usages.

The number of exactly and almost-similar type-usages ($|E(x)|$ and $|A(x)|$) defines a two-dimensional space, in which we can plot the type-usages of a software package. A scatter plot in this space enables us to graphically validate our assumption, i.e. to see whether the majority of points are in the bottom of the figure. Figure 5 represents the SWT type-usages that we have extracted in this 2D space.

We make the following observations. First, the cloud of points is much more horizontal along the x-axis, which validates our assumption. Second, points depicted as diamonds (e.g. the point at coordinate $(189, 3)$) represent type-usages whose S-score is greater than 97%. The figure shows that strange type-usages are all located in the same zone: the top left-hand side part of the figure, which

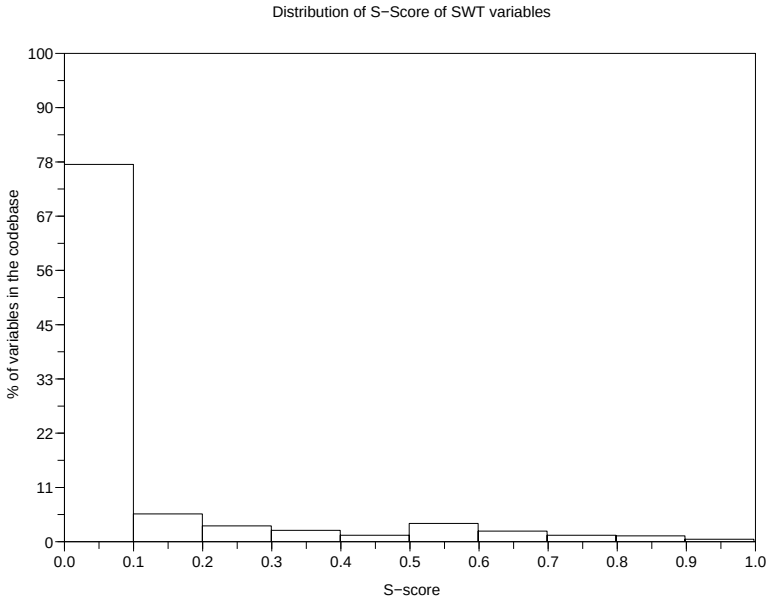

Fig. 4. Distribution of the S-Score based on the type-usages of type SWT.* in the Eclipse codebase. Most type-usages have a low S-Score, i.e. are not strange.

can somehow be called the "zone of strange" (parameterized by a threshold on the S-score). There are 25 type-usages in this zone, out of a total of 44435 type-usages. This can not be clearly seen in the figure, since the space is discrete and some points are exactly at the same place. All type-usages of this zone of strange will be analyzed in section 4.4.

4.2 The Ability of S-score to Catch Degraded Code

Even if the distribution of the S-score seems reasonable, we want to be sure that a faulty type-usage would be caught by the S-score. For this to be true, a type-usage with a missing method call should have a higher S-score than a normal type-usage.

To assess the validity of this assumption, our key insight is to simulate missing method calls. Given a type-usage from real software, the idea is to remove one by one each method call[14], and to check whether the S-score of the artificially created faulty type-usage has a higher S-score than the original one. This validation strategy has several advantages: (a) there is no need to manually assess whether a type-usage is faulty, we know it by construction, (b) it is based on real data (the type-usages come from real software), (c) it is on a large-scale (if

[14] if a type-usage contains at least 2 method calls

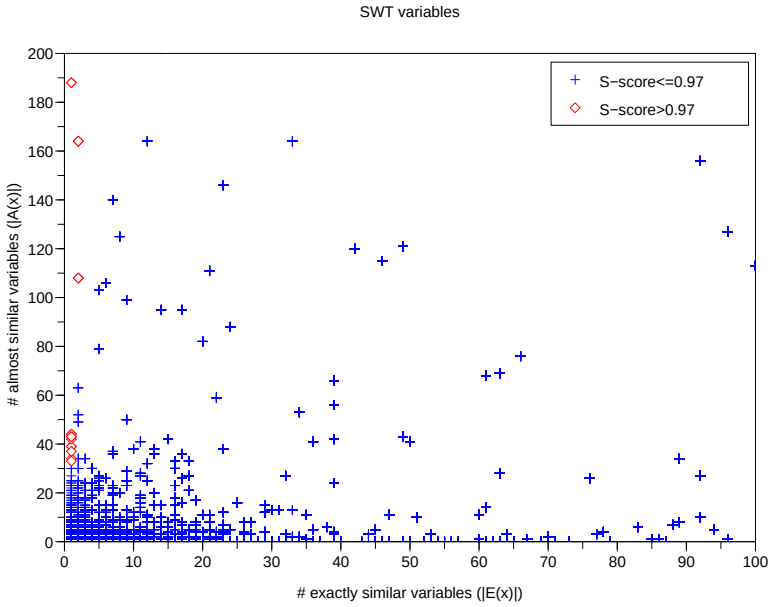

Fig. 5. Scatter Plot of the Type-Usages of Type SWT.* in the Eclipse Codebase. The red diamonds indicate very likely issues (i.e., their S-score in greater than 0.97).

the codebase contains N type-usages, with m_i method calls, the detection system is tested with $\sum m_i$ different queries.

We have conducted this evaluation on our SWT dataset. The evaluation strategy described in the previous paragraph creates 55623 different simulated missing method calls. We have checked whether the S-Score of these different simulated bugs is higher than the S-Score of the original (non-degraded) type-usage. The results of this evaluation is that 94% (52154/55623) of the degraded type-usages are stranger than the initial one i.e., they have a S-Score higher than the S-score of the initial one.

Furthermore, the difference of the S-score is high as shown by table 2: the average S-score of normal data is 0.16 (low) and the average S-score of degraded data is 0.76 (high). The percentile shown on rows #2 and #3 of the table strengthen the confidence in this finding, while 62% of the normal type-usages have a S-score lower than 0.1, there is only 0.3% of degraded type-usages whose S-score is under 0.1. These numbers validate that the S-score correctly recognizes faulty type-usages.

	Normal type-usages	Degraded type-usages
Mean S-score	0.16	0.76
S-score<0.1	62%	0.3%
S-score>0.9	0.6%	35%

Table 2. The S-Score for initial and degraded data. The S-Score is able to capture faulty type-usages.

4.3 The Performance of the Missing Method Calls Prediction

The third evaluation of our system measures its ability to guess missing method calls. The assumption underlying this evaluation is that our algorithm to predict missing method calls (cf. 3.5) should be able to predict calls that were artificially removed.

We have used the same setup as for evaluating the characteristics of the S-score (cf. 4.2), i.e. we have simulated missing method calls. However, instead of looking at the difference of S-score between real and degraded data, we have tried to guess the method call that was artificially removed with the technique described in 3.5.

For instance, given a real type-usage of the codebase representing a `Button` and containing `<init>` and `setText`, we test the system with two different queries: 1) `<init>` only and 2) `setText` only. The system may predict several missing method calls, but a perfect prediction would be `setText` as missing method call for the first query and `<init>` for the second query.

Hence, the system is evaluated with the same number of queries as in 4.2 i.e. 55623 artificial bugs. Then, we can measure the relevance of the missing method calls that are predicted. We measure the relevance of a single query using precision and recall:

- **PRECISION** is the ratio between the number of correct missing method calls (i.e. that were actually removed during the degradation of the real type-usage) and the number of guessed missing method calls. Note that the precision is not computable if the system outputs nothing, i.e. if the number of guessed missing method calls is null.
- **RECALL** is the number of correct missing method calls over the number of expected answers. In our evaluation setup, the number of correct missing method calls is either 0 or 1 and the number of expected answers is always one, hence the recall is a binary value, either 0 or 1;

We measure the overall performance of the system using the following metrics:

- **ANSWERED** is the percentage of answered queries. A query is considered as answered if the system outputs at least one missing method call.

$$ANSWERED = \frac{N_{answered}}{N_{query}}$$

– **MEANPRECISION** is the mean of the precision of answered queries. Since the precision is not computable for empty recommendations (i.e. unanswered queries),

$$MEANPRECISION = \frac{\sum_i PRECISION_i}{N_{answered}}$$

– **MEANRECALL** is the mean of the recall of all queries[15], i.e.

$$MEANRECALL = \frac{\sum_i RECALL_i}{N_{query}}$$

MEANRECALL is directly related to ANSWERED, since an unanswered query has a null recall. Hence, the lower *ANSWERED*, the lower *MEANRECALL* and vice versa.

MEANPRECISION describes the rate of false positives: the lower *MEANPRECISION*, the greater the number of false positives. Hence, we would like to have a high *MEANPRECISION*. Also, even if the precision is high, the system might simply recommend nothing for most queries (cf. formula above). Hence, a good system must have a high *MEANPRECISION* and a high *ANSWERED*, which means it is right when it predicts a missing method calls and it does not miss too many missing method calls.

Results We evaluated the DMMC system based on the evaluation process and performance metrics presented above. We have evaluated the two algorithms presented in 3.5 (DMMC-core and the variant with filtering DMMC-filter). The filtering version simply removes certain recommendations from the initial set of recommended method calls $R(x)$. Hence, the filtering version will mechanically have lower or equal ANSWERED and MEANRECALL. However, we hope that the filtering strategy would increase MEANPRECISION.

Table 3 presents the results. The three performance metrics of DMMC-core are high: the PRECISION of 84% shows that the core-system has a low false positive rate while still being able to answer 80% (ANSWERED) of the generated queries. Second, table 3 validates the filtering strategy defined in 3.5. As shown by the reported numbers, it significantly improves the precision. With a filter of 90%[16], the system is able to have a precision of 98% while still answering 67% of the queries. These numbers validate the ability of the system to correctly detect missing method calls.

[15] setting the denominator of MEANRECALL to $N_{answered}$ would be misleading because a system that predicts something only for 1% of the queries could still have a high MEANRECALL

[16] This threshold was chosen by intuition: it implies that a method call should be predicted if it is present in most of the almost-similar type-usages. Our various experiments showed that this threshold is not sensitive, all values from 80% to 95% produce results of the same order of magnitude.

System	N_{query}	ANSWERED	MEANPRECISION	MEANRECALL
DMMC-core	55623	80%	84%	78%
DMMC-filter (90%)	55623	67%	98%	66%

Table 3. Performance metrics of two variants of the DMMC system. Both have high precision and recall.

4.4 Finding New Missing Method Calls in Eclipse

The evaluation results presented in 4.2 and 4.3 suggest that a software engineer should seriously consider to analyze and change a type-usage, if it has a high S-score and recommended methods with a high likelihood (say $\phi(m, x) > 90\%$). However, it may be the case that our process of artificially creating missing method calls does not reflect real missing method calls that occur in real software.

As a counter-measure to this threat of validity, in this fourth evaluation, we applied the DMMC system to Eclipse v3.4.1. We searched for missing method calls in the Standard Widget Toolkit of Eclipse (SWT) type-usages of the Eclipse codebase. We chose to search missing method calls related to the SWT for the following reasons. First, we can reuse the extracted dataset used for the automatic evaluation. Second, finding method calls that remain in the user interface of Eclipse after several years of production is ambitious:

- since the community of users is large, the software is used daily in plenty of different manners, and missing method calls had a chance to produce a strange behavior. Furthermore, bugs in the user interface are mostly *visible* and easy to localize in code.
- since the community of developers is large and the codebase is several years old, most of the code has been read by several developers, which increases the probability of detecting suspicious code;

The following shows that we actually found some missing method calls related to the user interface of Eclipse.

We analyzed the strangest type-usages found by DMMC in Eclipse v3.4.1. They are in the zone of strange of figure 5 and have all a S-score greater than 0.97. For each strange type-usage, we tried to understand the problem so as classify it as a true or false positive and so as to file a bug in the Eclipse Bug Repository.

Table 4 gives the results of this evaluation. The first column gives the location of the strange type-usage. The second column gives the acronym of the problem underlying the strange type-usage (the following subsections elaborate on each such problem). The third column gives the S-score of the type-usage and the fourth column the bug id of our bug report or "NR" for non-reported. The last column indicates the feedback of the Eclipse developers[17] based on the bug report.

[17] We gratefully thank them, esp. Daniel Megert (IBM) and Markus Keller (IBM).

Location	Problems	S-score	Bug Id	Val.
ExpressionInputDialog.okPressed	SA	0.99	296552	√√
ExpressionInputDialog.close	EB+VAPIBP	0.99	297840	?
RefactoringWizardDialog2.okPressed	SU	0.99	296585	x
NameValuePairDialog.createDialogArea	VAPIBP	0.98	296581	√√
AlreadyExistsDialog.createDialogArea	VAPIBP	0.98	296781	√√
CreateProfileDialog.createDialogArea	VAPIBP	0.98	296782	√√
AboutDialog;.createDialogArea	EB+VAPIBP	0.98	296578	?
TextDecoratorTab.<init>	WA	0.98	NR	
UpdateAndInstallDialog:createDialogArea	VAPIBP	0.98	296554	√
RefactoringStatusDialog;.createDialogArea	VAPIBP	0.97	296784	√√
AddSourceContainerDialog:createDialogArea	VAPIBP	0.97	296481	√√
GoToAddressDialog:createDialogArea	VAPIBP	0.97	296483	√√
TrustCertificateDialog.createDialogArea	EB	0.97	296568	√
TitleAreaDialog;.createDialogArea	FP	0.97	NR	
StorageLoginDialog.createContents	WA	0.97	NR	
BrowserDescriptorDialog.createDialogArea	WA	0.97	NR	
StandardSystemToolbar.<init>	FP	0.97	NR	
ChangeEncodingAction$1.createDialogArea	SA+VAPIBP	0.97	275891	√√
JarVerificationDialog.createDialogArea	EB+VAPIBP	0.97	296560	√

Table 4. Strange type-usages (S-Score > 0.97) in Eclipse, and the corresponding problems and bug reports.

The symbol √√ means that the head version of Eclipse is already patched accordingly, √ means that the bug is validated but not fixed (for instance, because it is part of no longer maintained code), `x` indicates that the bug has been marked as invalid, and finally a question mark "?" indicates that the comments and the status of the bug report do not yet allow us to conclude. We now elaborate on the kind of problems revealed by the detected missing method calls.

SA: Software Aging Missing method calls may reveal problems related to software aging. Let us elaborate on the two corresponding strange type-usages of table 4.

According to the system, `ExpressionInputDialog` contains strange code related to closing and disposing the widgets of the dialog. Our manual analysis confirms that there are plenty of strange things happening in the interplay of methods `okPressed`, `close` and `dispose`. We found out that these strange pieces of code date from a set of commits and a discussion around a bug report [18]. Even if the bug was closed, the code was never cleaned. In this case, software aging comes from measures and counter-measures made on the code in an inconsistent manner.

The other example is in `ChangeEncodingAction` which contains code related to a very old version of the API and completely unnecessary with the current version. Following our remark about this class to the Eclipse developers, the

[18] see https://bugs.eclipse.org/bugs/show_bug.cgi?id=80068

code has been actualized. In this case, software aging comes from changes of the API that were not reflected in client code.

SU: Software Understanding Missing method calls may reveal problems related to software understanding. The third warning produced by the system concerns `RefactoringWizardDialog2.okPressed`. The system misses a call to `dispose` for certain widgets involved. The reason is that `okPressed` usually closes the dialog and dispose widgets. However, `RefactoringWizardDialog2.okPressed` uses the dialog itself to show an error, hence does not dispose anything, which is a very strange manner according to the Eclipse practices. Hence, this piece of code has a very high S-score. Interestingly, the code was so hard to understand that we created an incorrect bug report, which was invalidated by an Eclipse developer. Since it would require a significant refactoring to clean this strange code (and decrease its S-score), the developer did not modify this method. This case study validates the warning produced by a high S-score and also shows that it is not always possible to solve a warning by simply adding the missing method call.

VAPIBP: Violation of API Best Practices Strange type-usages often reveal violations of API best practices. An API best practice is a programming rule which is not enforced by the framework code or the programming language. In the following, we discuss several API best practices of Eclipse whose violations can be detected by our system.

Call super.createDialogArea: It is standard to create the new container widget of a dialog using the framework method `createDialogArea` of the super class `Dialog`. This best practice is documented in the API documentation of `Dialog`. However, certain type-usages do not follow this API best practice and create an incorrect clone of `super.createDialogArea`: there is an important method call present in `super.createDialogArea` and which is missing in the clone (e.g. setting the dialog margin using `convertVerticalDLUsToPixels`). For instance, `AddSourceContainerDialog` instantiates and initializes the new Composite by hand and `UpdateAndInstallDialog` uses an ad hoc method: both are not 100% compliant with `super.createDialogArea` and trigger a very high S-score. Both violations have been reported and are now fixed in the Eclipse codebase.

Setting fonts: A best practice of Eclipse consists of setting the font of new widgets based on the font of the parent widget and not on the system-wide font. Not following this best practice may produce an inconsistent UI. To our knowledge, this API best practice is not explicitly documented but pops up in diverse locations such as: newsgroups[19], bug reports[20], and commit texts[21].

[19] http://mail.eclipse.org/viewcvs/index.cgi/org.eclipse.ui.browser/src/org/eclipse/ui/internal/browser/BrowserDescriptorDialog.java

[20] https://bugs.eclipse.org/bugs/show_bug.cgi?id=175069 and https://bugs.eclipse.org/bugs/show_bug.cgi?id=268816

[21] http://mail.eclipse.org/viewcvs/index.cgi/org.eclipse.ui.ide/src/org/eclipse/ui/internal/ide/dialogs/ResourceInfoPage.java?sortby=log&view=log and http://mail.eclipse.

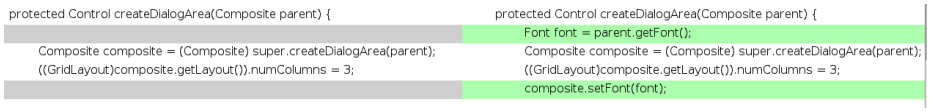

Fig. 6. Excerpt of revision 1.5 of Apr. 10 2006 of `BrowserDescriptorDialog.java`. Two missing method calls related to setting fonts are added.

The programming rule associated to this API best practice is to call `getFont` on the parent widget and to call `setFont` on the newly created widget. Figure 6 illustrates this point by showing the result of a commit which solves a violation of this best practice: the new code at the right hand side contains the previously missing method calls. Our system automatically detects the missing calls related to such violations.

Don't set the layout of the parent: Another API best practice of Eclipse consists of never setting the layout of the parent widget, i.e. not calling `setLayout` on the parent. Our system finds violations of the API best practice. At first sight, it seems contradictory since our system searches for missing rather than extraneous method calls. However, there is a logical explanation.

When one overrides `createDialogArea`, there are always two composites to work with: the `parent` and the newly created one for the dialog area, which we call `newcomp`. Their responsibilities are different, and so are the typical methods that developers call on them. Typically, one calls `setLayout` on the newly created `Composite`, but never on the parent.

When a developer accidentally calls `setLayout` on the parent widget, the set of almost exact type-usages consists mostly of `newcomp`-based type-usages and not of `parent`-based type-usages. In other terms, the system believes that this type-usage is a new created widget and misses the corresponding calls. In this case, the system is right in predicting a strange type-usage but wrong as far as the predicted missing method call is concerned. That is why software engineers always have to analyze and understand the causes of the strange type-usage before adding the predicted missing method call.

For illustration, let us consider such a violation in table 4: in the class `ChangeEncodingAction` of Eclipse Ganymede, there is a call to `setLayout` on the parent. To confirm the analysis we have just presented, we asked the Eclipse developers if this code is correct: they agreed on our diagnosis, filed a bug in the repository[22] and changed the code of `ChangeEncodingAction` accordingly[23].

`org/viewcvs/index.cgi/org.eclipse.ui.browser/src/org/eclipse/ui/internal/browser/BrowserDescriptorDialog.java`

[22] https://bugs.eclipse.org/bugs/show_bug.cgi?id=275891

[23] http://mail.eclipse.org/viewcvs/index.cgi/org.eclipse.ui.editors/src/org/eclipse/ui/texteditor/ChangeEncodingAction.java?r1=1.19&r2=1.20

Calling dispose: The SWT toolkit uses operating system resources to deliver native graphics and widget functionalities. While the Java garbage collector handles the memory management of Java objects, it can not handle the memory management of operating system resources. Not disposing graphical objects is a memory leak, which can be harmful for long-running applications. For instance, the following code of `ExpandableLayout` produces a high S-Score (0.96):

```
// Location: ExpandableLayout.layout
size= FormUtil.computeWrapSize(new GC(..),..)
```

The newly created graphical object (`new GC()`) is not assigned to a variable. However, the Java compiler inserts one in the Java bytecode. Since the method `computeWrapSize`, which receives the new object as a parameter, does not dispose the new object, it is never disposed. That's why our system predicts a missing call to `dispose`. This problem was filed and solved in the Bugzilla repository independently of our work[24].

FP: False Positive Our system suffered from two false positives in this evaluation setup.

The first one is rather subjective (`TitleAreaDialog`), it is about the creation of a `Composite` instance, already discussed above (cf. *Call super.createDialogArea*) On the one hand, the problematic type-usage should use a `super.createDialogArea` for being initialized, and then it should be tailored by overriding certain default choices. In this perspective, it is an incorrect clone and a violation of the API best practice. On the other hand, the initialization code is quite different compared to the body of the framework method, hence it is almost not anymore a clone! There is no objective and clear separation between a clone and a source of inspiration.

The second false positive reveals that our algorithm is sensitive to the tyranny of majority, when the three following conditions are met: 1) there is huge number of almost-similar type-usages, 2) there is a small number of exactly-similar type-usages and 3) the type-usage is correct, i.e. it is normal to have only this set of method calls. One type-usage in the manually analyzed warnings turned out to be such a false positive (`StandardSystemToolbar`). Let us explain it briefly. Most SWT widgets uses a layout manager based on grids (`GridLayout`), hence most SWT objects have the corresponding layout data (`GridData`) set using the method call `setLayoutData`. However, the caller of `StandardSystemToolbar.createDialogArea` uses a `CLayout` which does not require having the layout data set; and `StandardSystemToolbar.createDialogArea` logically does not call `setLayoutData`. However, the tyranny of majority makes of our system believes that a call to `setLayoutData` is missing in this context.

EB: Encapsulation Breaking In three cases among the strangest type-usages, the system finds breakings of object encapsulation, a particular case of the law

[24] https://bugs.eclipse.org/bugs/show_bug.cgi?id=257327

This preprint is provided by the contributing authors to ensure timely dissemination of scholarly and technical work.

of Demeter [8]. While our system is not designed to find such violations, it turns out that these violations are also caught by the S-score. For instance, let us consider the following excerpt of `TrustCertificateDialog`:

```
certificateChainViewer = new TreeViewer(composite, SWT.BORDER);
certificateChainViewer.getTree().setLayout(layout);
certificateChainViewer.getTree().setLayoutData(data);
```

This code contains two violations of the law of Demeter, which both break the encapsulation provided by a `TreeViewer`. Our system detects them because these two violations are reflected in bytecode with two type-usages containing a single method call each. However, `Tree` objects never have only one call to `setLayout` or to `setLayoutData`, and the S-score of the two type-usages are accordingly very high.

WA: Workaround The analysis of 3 strange type-usages revealed a phenomenon that we call *workarounds*. Both use an inappropriate widget to create a filler (an empty `Label` and an empty `Composite`). While this works (the resulting UI is satisfying), fillers are normally set using margins and paddings of `GridLayout`. In some special cases, the standard way of creating fillers is not perfect-looking and such workarounds are useful. The system catches these workarounds since the corresponding type-usages do not contain the usual method calls (e.g. `setText` on a `Label`).

In such cases, the client code is neither faulty nor bad designed, and we did not file bug reports because they are due to the usage of a workaround that addresses a limitation of the API itself.

4.5 Summary of the Evaluation

To conclude this section, we sum up the main results of our evaluation of the DMMC system:

- The S-score captures faulty type-usages.
- The algorithm which predicts missing method calls achieves a precision of 84% in guessing simulated missing method calls.
- A pragmatic variant of the algorithm improves the precision up to 98%, i.e. less than 2% of false positives.
- Our analysis of the strangest type-usages of the Eclipse codebase showed that the S-Score produced 17/19 true positive warnings and 2/19 false positive warnings. Furthermore, the bug reports we wrote have already resulted in 8 patches to the Eclipse source code. These results confirm the very high precision obtained with the simulation of missing method calls.

5 Missing Method Call Detection in the Development Process

We pointed out in 2 that there are several manners of using DMMC. At development time, DMMC outputs the very likely missing method calls in the problem

```
$ java -jar de.tud.st.DMMC.jar eclipse.jar
1. Analyzing code of eclipse.jar ...
2. Computing E(x) and A(x) for 55623 type-usages ...
3. Computing S-score ...
4. Ordering by S-score ...
5. Computing missing method calls ...
type-usage: Composite
  location: GotoAddressDialog:createDialogArea, line 300
  S-score: 0.97
  missing call: <init>
....
```

Fig. 7. The DMMC system in batch mode. The output could be text or XML.

view of Eclipse. The corresponding prototype is an incubator project of the Code Recommender System that we build at TU Darmstadt[25].

At maintenance and quality assurance time, our missing method call detector is used in batch mode. The software engineer has to give the tool a set of Java class files (e.g. in a JAR file). Figure 7 illustrates the command line usage of our prototype and its output. Since the output consists of a list of recommendation, it can be easily integrated into any development tool and process (e.g. into an XML file or a table in a user-interface).

We now present a checklist that helps engineers interpret the missing method calls predicted by the DMMC system. This checklist comes from our own experience on using the system for searching for missing method calls in Eclipse.

1. **What is the responsibility of this method call?** The first thing to do is to carefully read the documentation of the missing method call to understand what its function is. This gives crucial insights on what type of problems we might encounter when this method call is missing.
2. **Is the surrounding code of the type-usage strange?** Interestingly, our evaluation on Eclipse showed that often, although a method call is really missing, the solution is not to insert the missing call. It's rather more meaningful to identify and fix the warning at a larger scope (e.g. fixing a violation of API best practices). Hence, it is very important to analyze the context of a strange type-usage before modifying the code.
3. **Can this missing method call produce a bug in special use cases?** If yes, one can imagine a special usage of the software to let the problem appear at runtime. Then, it is possible to describe a reproducible procedure in a bug report.
4. **Is it a false positive due to the tyranny of majority?** Section 4.4 highlighted that our system is sensitive to tyranny of majority: developers might assume this if the two previous analyses (2 and 3) were inconclusive.

[25] see http://www.stg.tu-darmstadt.de/research/core/

To sum up, the interpretation of the *predicted* missing calls is not straightforward but the time spent in such an analysis is rewarded by an improvement of the software quality and the diminution of the number of latent defects (cf. the evaluation on Eclipse). Hence, we do believe that using our approach systematically in software development processes helps to produce better software and is eventually economically valuable.

6 Related Work

Engler et al. [1] presented a generic approach to infer errors in system code as violations of implicit contracts. Their approach is more general-purpose than ours in the sense that we only detect a special kind of problems: missing method calls. The corresponding advantage is that our approach is automatic and does not require a template of deviant behavior and the implementation of one checker per template. The same argument applies for FindBugs [9], which also addresses low-level bugs and is successful only if an error pattern can be formalized.

Another interesting approach from the OS research community is PR-Miner [2]. PR-Miner addresses missing procedure calls in system code and not API-specific bugs as we do at the scope of each type-usage. Further, PR-Miner uses frequent item set mining, which is a NP-hard problem [10]; on the contrary, the computation of the sets of exactly-similar and almost-similar type-usages is done in polynomial time ($O(N^2)$, where N is the total number of type-usages).

There are several techniques for finding defects in programs based on the analysis of execution traces. For instance, Ernst et al [11], Hangal and Lam [12], and Csallner et al. [13] mine for undocumented invariants. Yang et al [14] and Dallmeier et al. [15] mine traces for ordered sequences of functions. Since our approach is based on the static analysis of source code, our approach requires less input data: it needs neither large traces of real usages nor comprehensive test suites, which are both difficult and costly to obtain.

Williams and Hollingsworth [16] propose an automatic checking of return values of function calls: this is a completely different kind of bugs compared to missing method calls. Chang et al. [17] also target another kind of bug: neglected tests on limit cases. Livshits et al. [3] extract common patterns from software revision histories. Hence, to be able to catch a defect, the repository must contain 1) a large number of occurrences of the same kind of bug and 2) a large number of corrections of these bugs. Our approach does not have these requirements, it is able to catch a strange type-usage even if this kind of strange code has occurred only once in the whole software history.

Wasylkowski et al. [4] searched for *locations in programs that deviate from normal object usage – that is, defect candidates*. This approach could be applied to missing method call detection. The main limitation is, however, its high rate of false positives (e.g. 694/790 for AspectJ). On the contrary our approach has a very low rate of false positives, as shown by both the automatic evaluation (4.3) and the analysis of the Eclipse codebase (4.4).

In: Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'2010)
This preprint is provided by the contributing authors to ensure timely dissemination of scholarly and technical work.

Finally, none of these related papers leverage the idea of simulating likely bugs to extensively explore the prediction space of the tool and thus achieve a large-scale evaluation.

7 Conclusion

In this paper, we have presented a system to detect missing method calls in object-oriented software. Providing automated support to find and solve missing method calls is useful at all moments of the software lifetime, from development of new software, to maintenance of old and mature software.

The evaluation of the system showed that: 1) the system has a precision of 98% in the context of an automatic evaluation which simulates missing method calls (55623 defects simulated) and 2) the high confidence warnings produced by the system convinced the Eclipse developers to patch the codebase. This is promising, especially if one considers the usual high false positive rates discussed in the literature.

One area of future work is to apply the concept of *almost-similarity* not only to method calls but to other parts of software. For instance, searching for *almost-similar* traces could yield major improvements in the area of runtime defect detections. Also, searching for *almost-similar* conditional statements is worth further investigation to improve the resilience of software w.r.t. incorrect inputs.

References

1. D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proceedings of SOSP'01*, vol. 35, pp. 57–72, 2001.
2. Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 306–315, 2005.
3. B. Livshits and T. Zimmermann, "Dynamine: finding common error patterns by mining software revision histories," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 296–305, 2005.
4. A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of ESEC-FSE'07*, (New York, NY, USA), pp. 35–44, ACM, 2007.
5. M. P. Robillard, "Topology analysis of software dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 4, pp. 1–36, 2008.
6. S. Kim and M. D. Ernst, "Which warnings should i fix first?," in *Proceedings of ESEC/FSE*, (New York, NY, USA), pp. 45–54, ACM, 2007.
7. G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero, "Understanding the shape of java software," in *Proceedings of OOPSLA*, ACM, 2006.
8. K. Lienberherr, "Formulations and benefits of the law of demeter," *ACM SIGPLAN Notices*, vol. 24, no. 3, pp. 67–78, 1989.
9. D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004.

In: Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'2010)

This preprint is provided by the contributing authors to ensure timely dissemination of scholarly and technical work.

10. G. Yang, “The complexity of mining maximal frequent itemsets and maximal frequent patterns,” in *KDD’04*, 2004.
11. M. D. Ernst, J. Cockrell, W. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.
12. S. Hangal and M. S. Lam, “Tracking down software bugs using automatic anomaly detection,” in *Proceedings of the 24th International Conference on Software Engineering (ICSE’02)*, pp. 291–301, 2002.
13. C. Csallner, Y. Smaragdakis, and T. Xie, “Dsd-crasher: A hybrid analysis tool for bug finding,” *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, pp. 1–37, 2008.
14. J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, “Perracotta: mining temporal api rules from imperfect traces,” in *ICSE ’06: Proceedings of the 28th international conference on Software engineering*, (New York, NY, USA), pp. 282–291, ACM, 2006.
15. V. Dallmeier, C. Lindig, and A. Zeller, “Lightweight defect localization for java,” in *Proceedings of ECOOP’2005*, 2005.
16. C. C. Williams and J. K. Hollingsworth, “Automatic mining of source code repositories to improve bug finding techniques,” *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 466–480, 2005.
17. R.-Y. Chang, A. Podgurski, and J. Yang, “Finding what’s not there: A new approach to revealing neglected conditions in software,” in *Proceedings of ISSTA’07*, 2007.

This preprint is provided by the contributing authors to ensure timely dissemination of scholarly and technical work.