Recommending Rename Refactorings

Andreas Thies

Lehrgebiet Programmiersysteme
FernUniversität in Hagen
D-58097 Hagen

andreas.thies@fernuni-hagen.de

Christian Roth

Bayer Technology Services GmbH
Building B610
D-51368 Leverkusen

christian.roth@bayertechnology.com

ABSTRACT

Variable names play a major role in program comprehension. However, their choice is often subject to the intuition (or intention) of individual programmers: although code conventions and style guides may constrain identifier usage, programmers are individuals naming program concepts individually. Especially if different parts of a program are written by different programmers, inconsistent naming of program entities may follow. This is unfortunate, since consistent naming would aid program comprehension, in particular if references pointing to same objects used in similar ways are named equally. As a first approach, we focus on assignments to discover possible inconsistency of naming, exploiting that a variable assigned to another likely points to same objects and, if declared with the same type, is likely used for the same purpose.

To explore the feasibility of our approach, we implemented a tool recommending rename refactorings to harmonize variable names based on an analysis of assignments and static type information. Evaluated on some open source projects the results seem promising enough to aim for some extensions, such as application to method names, inferred type information, and weakly type-checked languages.

Categories and Subject Descriptors

D.3.3 [**Software Engineering**]: Management – *Software quality assurance (SQA).*

General Terms

Management, Human Factors, Languages.

Keywords

Assignment analysis, Identifiers, Rename refactoring, Variable naming.

1. INTRODUCTION

Usually, software is developed by teams. In conventional software development processes (waterfall model, spiral model…) the software's design is defined up-front to enable uncomplicated integration of the team members' work. Often, the design granularity reaches down to the class and method level. Below this level, however, the developer may decide freely on implementation details such as local variables, private methods, and their parameters, a freedom that extends to the choice of their names.

From a static design perspective, nothing speaks against this freedom: entities hidden behind interfaces (interface types as in C# and JAVA as well as the class interfaces defined by access modifiers) are not accessible from the outside so that their names need not be subject to standardization, the less so if only a small number of developers deals with the implementation details hidden behind an interface (which is typically the case).

That implementation details such as names of local variables are of no relevance outside their static scope dramatically changes when shifting from a static to a dynamic perspective: at run time, interfaces do not serve as borders anymore, and assignments within a module become indistinguishable from assignments between modules. In fact, as soon as need for debugging arises (an this moment will come in every software development process) the developer needs to look at data flow across module boundaries, and any unintuitive change of names hampers understanding of what is going on.

Generally, identifiers make up to 70 % of the source code, often taking responsibility for the documentation of the program [3]. In fact, the importance of naming for documentation purposes becomes clear in languages like SMALLTALK and JAVA where the method specifications require names for formal parameters which can not be referenced [4] (§ 8.4.3.1) making the name definitions technically obsolete. Besides, in programming languages without explicit type annotations, names are frequently used to document type information of the referenced entity (the omnipresent aString, anInteger etc. parameters in SMALLTALK). This practice even extends to typed languages, as documented by the *Hungarian Notation* [8], coding type and usage information in a single name.

In this paper, we present an approach to harmonize names in JAVA programs by an analysis of variable assignments. Variables assigned to each other result in multiple references – aliases – to the same object. If these references are used in the same manner – we will approximate this here by analyzing static type information – choosing an identical name appears desirable. To avoid unintended inconsistent naming of such variables a recommendation system giving response to the programmer – as we describe here – seems even more advantageous.

In section 2 we motivate our approach by giving examples from the JUNIT project's source code how names could be harmonized leading to an improved code quality. In section 3 we go into details of our approach, and explain how to derive recommendations from a program. Also, we explain where language specifications may constrain harmonization of names. Section 4 presents a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
RSSE'10, May 4, 2010, Cape Town, South Africa.
Copyright © 2010 ACM 978-1-60558-974-9/10/05 ... $10.00

proof-of-concept implementation of our idea as a plug-in for ECLIPSE'S JAVA DEVELOPMENT TOOLS (JDT). Section 5 demonstrates both the effectiveness and limitations of our approach based on a by hand analysis of all recommendations given by our implementation for several open source projects. Section 6 points out how results can be improved and lists ideas for future work.

2. MOTIVATION

From our experience, even high quality code shows misleading names. The following code is taken from the JUNIT project (3.8.2) and declares a method within the BaseTestRunner class:

```
public Test getTest(String suiteClassName) { … }
```

JUnit allows one to combine single test cases together into test suites, where a suite may be part of an enclosing suite as well. A JUNIT test run needs either a single test case or a test suite as an input. The JUNIT implementation's type hierarchy reflects this directly: The classes TestSuite and TestCase both implement the interface Test. While for every test run a Test object must be supplied, the internal implementation differs depending on whether a single test or a test suite is passed.

Because of the different behavior of single tests and suites, meaningful names seem beneficial, describing whether a Test references a single test case, a test suite, or possibly both. The method declaration shown above fails in this respect because the parameter's and method's name taken together draw an inconsistent picture. The formal method parameter suiteClassName clearly suggests that the name of a test suite is passed to the method. Also, the method usage shows that getTest handles test suites only: Within the whole JUNIT project there are three method invocations binding to the method in question:

Within junit.awtui.TestRunner.runSuite():

```
final Test testSuite=
   getTest(fSuiteField.getText());
```

Within junit.awtui.TestRunner.runSuite():

```
final Test testSuite= getTest(suiteName);
```

Within junit.textui.TestRunner.start(String args[]):

```
Test suite= getTest(testCase);
```

All names on the left hand sides of the assignments make clear that the invocations expect the method to return a test suite. But the method's name getTest as well as its different parameters suggest that the method accepts both, a TestSuite and a TestCase.

Also, the name used within the method body leaves it unclear whether the returned reference's type is limited to test suites:

```
…
Test test= null;
…
return test;
```

Instead, the returned variable with its name test suggests that both types of tests may be referenced. A programmer editing the method body remains uninformed that in this specific case only test suites need to be considered (and possibly implements needless case analyses).

Summing up, we have seen misleading names for both, a method and a variable inhibiting concise program comprehension. A more descriptive name for the variable test (e.g. testSuite) as well as for the method getTest (e.g. getTestSuite) seems desirable.

In the former example from JUNIT's source code the starting point for our observations was the method signature of getTest with its clear gap between the formal parameter suiteClassName (apparently limiting the method's scope to suites) and the return value test (suggesting that also single test cases may be returned). But this observation itself does not imply that the name is misleading (because there might be scenarios in which the variable naming makes sense, e.g. if a suite must only contain exactly one test case). The evidence is taken from an analysis of method usage, especially from the names used on the left hand sides of the assignments. In all three cases a variable named test was returned which was assigned to a variable named testSuite or suite.

But should every name change in context of an assignment be harmonized? Truly not! Again, an example is instructive: JUNIT'S Assert class has the following method signature:

```
static public void assertNotNull(Object object)
```

A corresponding method invocation from the method runTest() within the TestCase class is

```
private String fName;
…
assertNotNull(fName);
```

Here, the different names seem adequate – not to say necessary: Neither does it make sense to name the formal parameter of the assert method fName (or name if we recognize the code convention used) nor the name fName within the runTest method should be changed to object.

But what makes the difference to the previous example? It is the *usage* of the references: Within the runTest method fName is used to point to the name of the test – making it quite descriptive. In contrast to this, the only usage of the reference passed to the assert method is a check whether it is a null reference. This check for null can be done for every kind of object, making the naming of the parameter object quite reasonable.

In the former example, the different usage of both references (still pointing to the same object) is reflected by their static types. While fName is typed as a String, object is declared as an Object. In strongly typed languages (as in JAVA) a reference's static type limits the set of its accessible methods and fields – thus its potential behavior. If a programmer intentionally restricts a reference's abilities by using a more restrictive (super) type for an assigned variable he suggests that the reference will be used in another manner. As a first approach, we treat differing static types as an indication for differing usage, so we refrain from harmonizing variables in these cases. Chapter 6 will cover how weakly type-checked languages may be addressed and our idea might be improved for strongly type-checked languages by using inferred types.

2.1 Related Work

Various studies document the influence of entity's names to source code readability and quality. According to Caprile and Tonella "*identifier names are one of the most important sources about program entities*" [2]. Lawrie et al confirmed the correlation between names and program comprehension in an empirical study [6], while Buse and Weimer showed correlation between readability and source code quality [1].

Even though the overall quality of names in programs improved in context of modern programming languages [7] flawed identifiers are still numerous: Høst and Østvold [5] detected what they called *naming bugs* by an analysis of method names and their implementations. A method name of the form contains…() should

likely return a Boolean value. Equally they draw conclusions from the method's body structure – e.g the existence of loops.

To aid in demand for meaningful names different attempts has been made. In [2] an approach for standardization of names is presented which splits names up in their composing terms to standardize the single terms as well as the syntax of their arrangements. As well, the idea of consistent naming was expressed by Deissenboeck and Pizka in [3] who described a formal model to specify a mapping between program concepts and identifiers. Their approach resulted in an identifier dictionary tool giving the programmer detailed information of name usage within a program aiding in consistent naming.

3. MAKING NAME RECOMMENDATIONS

JAVA knows two types of assignments, both have been shown in the motivating examples. Firstly, references may be assigned to variables using the assignment operator as in

```
fName = name;
```

or

```
fName = getName();
```

In the first example, an assignment between two variables (i.e., local variables, fields or formal method parameters) is made. In the second example, a reference returned by a method is assigned to a variable. Even though we have seen in the motivating example of section 2 that method names may be flawed as well, we ignore them in the context of our first analysis and concentrate on names of variables. For that reason we – preliminary – bypass method names by replacing them with the returned expression. For example if the method getName() has a return statement

```
return name;
```

we treat the assignment

```
fName = getName();
```

as an assignment of name to fName.

The opposite case – an assignment *to* a method – is the second type of assignments in JAVA. For a method call as

```
setName(fName);
```

a reference to the object referred by fName is assigned to the formal parameter of the method setName.

The set of a program's references can be described as a graph G(N, E) – the *assignment graph* – in which every variable is represented as a node $n \in N$ and every assignment is represented as a directed edge $e \in E$. An assignment graph is neither necessarily connected (for example in the case of an unused method), nor necessarily free of cycles (e.g. a cycle results when two variables' values are swapped using a third temporary variable). The graph's edges' degrees may reach arbitrary values because a variable may be involved in an arbitrary number of assignments – on the left hand side as well as on the right hand side. Once an assignment graph is available, assignments without type change but holding a name change can be determined with the following algorithm 1.

```
for every edge e = (n1,n2) ∈ E
  if name(n1) ≠ name(n2)
    if declaredType(n1) = declaredType (n2)
      report(e)
```

Algorithm 1. Determining all assignments with name change and same declared types

3.1 Language Restrictions

In practice, not every name change reported by the preceding algorithm 1 can be harmonized and thus result in a reasonable warning. For example in JAVA, two declarations must not be declared with same names if one declaration is located in the scope of the other [4] (§6.3). Also, two entities must not be declared with same identifiers if one variable is referenced without a qualifier [4] (§6.6) in the scope of the other, leading to a change of binding. In the following example the method parameter newName must not be renamed to name, otherwise the binding of the local variable will change.

```
String name;
void setName(String newName){
  name = newName;
}
```

Feasibility of variable renaming is studied in context of refactoring tools quite well. There exists (almost) reliable tool support for these refactorings which we have incorporated to detect whether a variable harmonization is realizable.

A softer constraint which may limit the output's value of algorithm 1 are naming conventions. As seen in the numerous examples given above, programmers may follow code conventions to incorporate additional information into identifiers. Often, names are equipped with a prefix to highlight their different scopes. Also, constant names often use upper case letters. The following code illustrates a popular naming convention.

```
public final String DEFAULT_NAME = "n.n.";
String fName = DEFAULT_NAME;

void m(String aName) {
  String name = fName;
  m(name);
}
```

Here, the identifiers fName, name and aName should not be harmonized. Strictly speaking, they are harmonized already if the used convention is considered. In practice, such conventions (and also ones including type information such as [8]) are easy to respect for an automatic recommendation system, even though an individual consideration of every convention is required during tool implementation. Another exception from harmonization of names are constants (in JAVA static final fields): In practice, we never found a situation where harmonization of a constant name seemed beneficial (even when respecting case conventions). Our automatic analysis therefore ignores assignments between variables and constants in general.

3.2 Reasons for Recommendations

Until now, we focused on the assignment graph's edges (that is, what algorithm 1 reports) to find pairs of variables to harmonize. However, detecting the assignments pointing to problematic names is only the half way to go. In the end, *declaration's names* need to be harmonized – not assignments. The user must get informed which entity to rename, so recommendations need to correspond to nodes instead of edges. There is no general rule which declaration adjacent to a reported edge to rename. Instead, this matter depends on the concrete cause for the differing names. From our experience (see section 5), there are three different types of problem names: misspelling, synonyms and inaccurate names. Depending on the type of problem, different strategies apply which declaration to annotate with a warning.

Misspelled names Spelling errors in identifiers can remain undetected by the compiler, since integrated development environments assist in completing names. Once an identifier is misspelled, the mistake gets propagated through the whole program, and becomes apparent only in assignments to other names that are spelled correctly. In such cases, only a recommendation for the (differing) misspelled name should be given.

Misspellings can be detected with a dictionary approach; also the variable's declared type name might hold information about the programmers intent. For the statement

```
Object objcet = a.object;
```

both approaches are helpful and lead to a recommendation to rename `objcet` to `object`.

Synonymous names If a check for misspelling fails to generate a suggestion, another type of error may be the cause: synonyms. Synonymous names have the same meaning but a different string representation, for example synonyms in a lexicographic sense (*buy* vs. *purchase*) or different abbreviations of the same word (*temp* vs. *tmp*). For synonyms, it is hard to decide automatically which identifier to choose. Often, this depends on the programmer's individual preferences but it seems beneficial to prefer names more common for assigned variables. Thus, only the less frequent name gets marked with a corresponding recommendation.

Inaccurate Names The third type of fault results from inaccurate naming. An inaccurate name simply does not express what the reference's intention is. An automated analysis will be unable to guess which of two differing identifiers expresses the programmer's intention, but the same algorithm as introduced in the former paragraph for synonym names turns out to be beneficial for at least some cases: The first motivating example in section 2 with its local variable `test` showed an instance of inaccurate naming. As can be seen, the name `test` is used once, as well as the name `suite`. The name `testSuite` is used twice. Thus, a recommendation is given to rename `test` to `testSuite`.

4. IMPLEMENTATION

To get an impression of the effectiveness of our approach for variable name recommendations harmonizing variable names, we implemented a recommendation system as a plug-in for the ECLIPSE JAVA DEVELOPMENT TOOLS (JDT). For our purposes the ECLIPSE platform offers various advantages. A valuable assignment analysis for JAVA code in ECLIPSE projects is available, for instance from the INFERTYPE refactoring implementation [9]. The high quality refactoring engines as a part of the ECLIPSE LANGUAGE TOOLKIT (LTK) enable to check for infeasible rename refactorings and the built-in dictionary of ECLIPSE helps to discover misspelled names. Furthermore, the *warnings* available in ECLIPSE offer a comfortable way of displaying suggestions, and so called *quick fixes* allow an immediate rename refactoring execution.

Our recommendation system's implementation is itself divided into several ECLIPSE plug-ins. A single core plug-in provides access to the assignment graph and offers functionality to determine type information. It also provides extension points for *cause finders* and *recommendation generators*. *Cause finders* are detectors for certain patterns in assignment graphs. For our analysis, we implemented a cause finder to detect name changes as reported by algorithm 1.

A *recommendation generator* returns suggestions for a cause reported by a certain cause finder and provides all parameters required for a rename refactoring. We implemented two recommendation generators, one to make recommendations for misspelled names and another to handle both, synonyms and inaccurate names by suggesting the more common identifier. Figure 1 shows a screenshot with a generated warning. The according source code comes from JUNIT and matches the first motivating example.

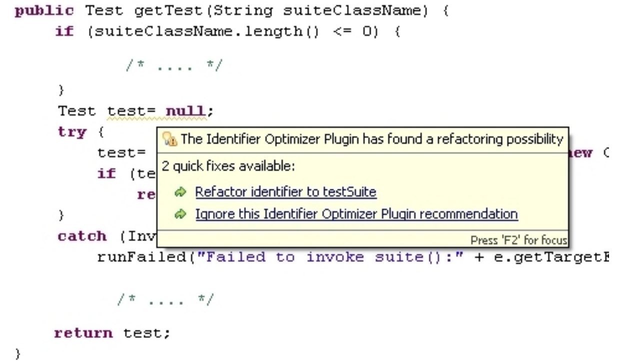

Figure 1. A warning generated due to an identifier change

5. EVALUATION

To evaluate our approach, we applied our plug-in to several open source projects. Because the evaluation involved much manual effort (we checked every generated warning by hand) we concentrated on variables typed with a class or interface type (instead of a primitive type). For these elements type changes in context of an assignment seem more frequent so focusing on them gives us a better feedback as to whether the incorporation of type information in algorithm 1 is beneficial. Table 1 lists the examined projects, the number of variables declared with a non primitive type, and the number of assignments between these variables. Furthermore, Table 1 shows the number of assignments with same type but differing identifiers as computed by algorithm 1. Altogether, 12.7 % of all assignments are of this kind.

Table 2 gives a detailed insight in the warnings generated from the reported assignments. The first column shows the number of assignments without a generated warning because of language constraints restricting a harmonization of names (see section 3.1). The following three columns summarize the 32 displayed warnings. An evaluation by hand revealed that 21 warnings were beneficial pointing to synonymous (4) or inaccurate (17) names.

Table 1. Sample projects used for the evaluation

Project	*Number of considered variables*	*Number of assign-ments*	*Reported assign-ments*
JUnit 3.8.2	175	270	22
Jester 1.3.7b	92	133	8
Apache Commons IO 1.4	106	214	16
Apache Commons Codec 1.3	25	60	0
total	398	677	46

Table 2. Analysis of the plug-ins suggestions

Project	*suggestions: without*	*beneficial*		*unserviceable*
	infeasible refactorings	*synonym names*	*identical usage*	*differing usage*
JUnit	0	3	12	7
Jester	0	1	3	4
IO	14	0	2	0
Codec	0	0	0	0
total	14	4	17	11

The dictionary approach for misspelled names mentioned in chapter 3.2 failed to generate any warnings (and is not mentioned in table 2 for this reason) In fact, the considered variables contained no misspellings. Here, projects in an early production status might show differing behavior.

11 suggestions turned out to be useless. These useless recommendations resulted from situations in which the concrete usage of a referenced object changes, while its declared type stays identical. This change of use is correctly reflected by the identifier. The following example from JUnit, namely from rerunTest() in junit.swingui.Testrunner is exemplary for such a case:

```
Test rerunTest= view.getSelectedTest();
```

In context of this assignment a name change is reported by our tool because the method getSelectedTest() returns a local variable named test. Even though the type stays the same the name change is quite reasonable, because the referenced object is used with another meaning for another purpose: the repeated run of a test.

Nevertheless, in our test projects 21 of 32 warnings were valuable resulting in a success rate of 65.6 % making the tool quite attractive and giving reason to aim for further extensions.

6. FUTURE WORK

Until now, we did not consider method identifiers. Because of specific prefixes as *get*, *set*, *count*, *is*… and often used additional specifications as in getObjectFromRepository a deeper insight into method naming is required (but can build on [5] and [2]). We are confident that an incorporation of method names will turn out as useful.

To reduce the percentage of useless suggestions a more detailed type analysis can be of value. Instead of interpreting the declared type, consideration of the accessed methods within the same block or class may be useful. Likewise, the inferred type (as defined in [9]) can offer more detailed information on the concrete usage of a reference as well as enabling our approach for weakly typed-checked languages.

Further ideas for future work may be taken from recent insights to naming practice made in context of the *convention over configuration* paradigm. Frameworks following this concept map entities to each other only by consideration of their names – sometimes even if the name's string representations do not mach. For example the RUBY ON RAILS framework maps a class named Person to a database table people by default. Such mappings may be incorporated in an assignment analysis by treating these mappings likewise assignments.

Also, the former example from RUBY ON RAILS leads to another interesting extension for name harmonization: By not only considering variable assignments but also incorporating add and get operations on collections, interesting results may follow. A variable person added to a collection people should be judged as reasonable by a recommendation system for rename refactorings. Incorporation of word pairs as regarded by RUBY ON RAILS may enhance naming recommendations for collections.

Last but not least, our research is not limited to suggestions for rename refactorings. Assignments between variables with different names but equal types can point to a recommendable type generalization performed by the EXTRACT INTERFACE or USE SUPERTYPE WHERE POSSIBLE refactorings, extending our approach to a recommendation system for a larger number of refactorings.

7. REFERENCES

[1] Buse, R. P. L., Weimer, W. R. 2008. A Metric for Software Readability. In Proceedings of the 2008 international symposium on Software testing and analysis (2008), 121-130.

[2] Caprile, B., Tonella, P. 2009. Restructuring Program Identifier Names. In Proceedings of the International Conference on Software Maintenance (2000), 97.

[3] Deissenboeck F., Pizka M. 2006 Concise and consistent naming. In Software Quality Control 14:3 (2006) 261-282.

[4] Gosling, J., Joy, B., Bracha, G. 2005. The Java Language Specification, 3rd Edition. Addison-Wesley.

[5] Høst, W., Østvold, B. M. 2009. Debugging Method Names. In Proceedings of the 23rd European Conference on Object-Oriented Programming (2009), 294-317.

[6] Lawrie, D., Morrekk, C., Field, H., Binkley, D. 2006. What's in a Name? A Study of Identifiers. In Proceedings of the 14th IEEE International Conference on Program Comprehension (2006), 3-12.

[7] Lawrie, D., Field, H., Binkley, D. 2007. Quantifying Identifier Quality: An Analysis of Trends. In: Empirical Software Engineering 12:4 (2007), 359-388.

[8] Simonyi, C. 1999. Hungarian Notation. In Visual Studio 6.0 Technical Articles (1999) http://msdn.microsoft.com/en-us/library/aa260976%28VS.60%29.aspx.

[9] Steimann, F. 2007. The Infer Type refactoring and its use for interface-based programming. In Journal of Object Technology 6:2 (2007), 67-89.

[10] Steimann, F., Mayer, P., Meißner, A. 2006. Decoupling classes with inferred interfaces. In Proceedings of the 2006 ACM symposium on Applied computing (2006), 1404-1408.