A Detailed Examination of the Correlation Between Imports and Failure-Proneness of Software Components

Ekwa Duala-Ekoko and Martin P. Robillard
School of Computer Science, McGill University
Montréal, Québec, Canada
{ekwa, martin}@cs.mcgill.ca

Abstract

Research has provided evidence that type usage in source files is correlated with the risk of failure of software components. Previous studies that investigated the correlation between type usage and component failure assigned equal blame to all the types imported by a component with a failure history, regardless of whether a type is used in the component, or associated to its failures. A failure-prone component may use a type, but it is not always the case that the use of this type has been responsible for any of its failures. To gain more insight about the correlation between type usage and component failure, we introduce the concept of a failure-associated type to represent the imported types referenced within methods fixed due to failures. We conducted two studies to investigate the tradeoffs between the equal-blame approach and the failure-associated type approach. Our results indicate that few of the types or packages imported by a failure-prone component are associated with its failures — less than 25% of the type imports, and less than 55% of the packages whose usage were reported to be highly correlated with failures by the equal-blame approach, were actually correlated with failures when we looked at the failure-associated types.

1 Introduction

Managers of software development teams are often interested in those components of a software that are most prone to failures. If failure-prone components can be identified before the release of a software system, then managers can allocate the often limited quality assurance resources to those components that need it the most — thereby improving the overall quality of the software.

The need to identify failure-prone components has inspired research efforts that investigate the correlation between software failures and various properties of the software, the software process, or a project's change history [6, 10, 13, 14]. For instance, research has provided evidence that *usage relationships*, such as the packages or the type of classes used by a component, are correlated with component failures. In a study of 52 Eclipse plug-ins, Schröter et al. reported that 71% of the components that used the Java compiler package (`org.eclipse.jdt.core.compiler`) needed to be fixed because of a *post-release* failure (that is, a failure that occurred within six months after the release of a system), whereas only 14% of the components that used the Eclipse user interface package (`org.eclipse.ui`) had to be fixed due to a post-release failure [14].

For identifying potentially failure-prone components, the component-import relationship is appealing because it is both easy to compute and straightforward to interpret. However, it is also a coarse measure that captures only elements of a software system's architecture and application domain. We were intrigued about the factors that could come into play when investigating the correlation between type imports and component failures. For example, it is not uncommon to find several unused import statements in Java — up to 67% of the 71 files in version 0.6 of FreeMind[1] have unused import statements. More importantly, some types that are actually imported by a file may never have been used in any code related to a failure. However, the approach used by Schröter et al. considers all the import statements of a failure-prone component as as equally likely to have contributed to the component's failures, with no distinction between unused statements, statements importing types not related to failures, and statements importing types associated with the failures of a component.

To gain more insight about the relation between import statements and component failure, we introduce the concept of a *failure-associated type*. A *failure-associated type* is an imported type referenced within methods modified in order to fix a failure. These type allows us to capture a more precise relationship between type usage and component failure. We conducted two studies to investigate the tradeoffs of using the *failure-associated type* approach instead of the

[1] http://freemind.sourceforge.net

equal-blame approach in capturing the correlation between import statements and component failures.

The first study, henceforth referred to as the *equal-blame* study, is a replication of the work by Schröter et al. The *equal-blame* approach assigns equal responsibility to all the import statements used by a component with failures, including unused imports. The second study, henceforth referred to as the *failure-associated type* study, limits the responsibility of the failure of a component to the types referenced within the methods modified to fix a failure. Our results indicate that less than 25% of the imported types, and less than 55% of the packages whose usage were reported to be statistically correlated with higher risk of failure by the equal-blame approach, were actually correlated with failures when we looked at the failure-associated types. The results were consistent for the imported types and packages for both Eclipse versions 2.0 and 3.0.

Our results suggest that, at least in the context of Eclipse, less than half of the imported types or packages used by a failure-prone component are actually associated with its failures. The failure-associated type concept also provides a more precise knowledge of the specific types of a given domain associated with failures. For example, the equal-blame approach identified the domain related to types in the package `org.eclipse.jdt.internal.compiler.parser` as being risky. Using the failure-associated type, we learned that even though both the `Parser` and `Scanner` classes belong to this domain, the use of `Scanner` is associated with a higher risk of failure, whereas the use of `Parser` is associated with a lower risk of failure. Failure-associated types therefore provide an increased understanding of the links between problem domains and failures, and takes us a step closer to understanding why some problem domains are more prone to failures than others.

The rest of the paper is organized as follows. Section 2 provides a description of the repositories used for collecting the change and failure history of a project; Section 3 presents our data collection techniques; Section 4 presents our data analysis techniques and a discussion of the results. We review related work in Section 5, and conclude the paper in Section 6.

2 Learning from History

The information about software components and software failures is generally available in version archives and bug databases. To identify failure-prone software components, these databases must be mined to map software components to software failures. We discus version archives and bug databases in this section, and defer the discussion on mapping failures to components to Section 3.

2.1 Version Archives

Version archives such as CVS and Subversion repositories contain information about changes to the source code. Such information includes details such as how the code was changed, the files affected by the change, when the change was made, who made the change, and an optional explanation (known as the *log message*) as to why the change was made. Several files submitted to the version archive by the same developer, at the same time, with the same log message, are said to be part of the same *commit transaction* [18]. Reasons for changes to the code include feature additions, bug fixes, or refactoring. Unfortunately, version archives lack detailed information about the purpose for a change or the version of the software to which a change was made. Since our study is focused on changes due to failures, and since we also need to know the version of the software in which the failure occurred, we must combine version archives with bug databases to identify changes due to bug fixes.

2.2 Bug Databases

Bug databases, such as Bugzilla used for Eclipse, contain reports about failures observed during the operation of a software. Bug reports are submitted by both developers and the users of the software, and contain information such as a description of the bug, the version of the software in which the bug was observed, the component of the software in which the bug was observed, and the severity of the bug — severity of a bug ranges from blocker (i.e., component is unusable) to trivial (i.e., cosmetic problem like misspelled words). Each bug report is assigned a unique ID once the report is submitted. Bug reports also contain a *status* field (e.g., NEW, ASSIGNED, RESOLVED, or CLOSED) and a *resolution* field (e.g., FIXED, DUPLICATE, or INVALID). A detailed description of a bug report, and life cycle of bugs is available from the The Bugzilla Guide.[2]

3 Data Collection

A software component C_i is said to be *failure-prone* in version k of a software if C_i was modified in order to fix a failure observed in version k of the software. *A software component in our context is simply a Java file*. The focus of our study was to identify the imported types used by a file that is failure-prone in version k. Consequently, the number of times a file was fixed in a given version is not considered; we were only interested in whether or not a file was fixed in version k because of a failure. We identify files that are failure-prone in a given version of the software by mapping changes in a version archive to bug reports in the bug database.

[2] www.bugzilla.org/docs/2.18/html/

```
import package1.W;
import package2.X;
import package3.Y;

class Foo {
  void sayIt(Y obj1){
     obj1.m1(null, true);
  }

  void doIt(X obj2){
     obj2.m2();
  }
}
```

Figure 1. Sample failure-prone file: the method `sayIt` occasionally throws a null pointer exception.

3.1 Identifying Failure-Prone Components

Version archives provide developers the option to indicate the purpose for a change in the form of a log message. Developers typically differentiate a change due to bug fixes from other changes by providing a reference to the corresponding bug report in the log message. For instance, the log message may contain references such as "bug 23124", "Fix for 6784", or just "8976".

To identify failure-prone files, we search the log message of each commit transaction for references to bug reports, and treat each reference as a bug ID. For instance, given the reference "Fix for 6784", we treat "6784" as a bug ID. Next, we attempt to retrieve the bug report with ID "6784" from the Bugzilla repository. If a bug report with the given ID does not exists in Bugzilla, we assume the commit transaction is not related to a bug fix. If a bug report with the given ID exists in Bugzilla, and its *resolution* is not DUPLICATE, its *severity* is not low (ENHANCEMENT or TRIVIAL), and its *status* is FIXED or RESOLVED, then we consider the transaction to be a fix to a failure, and the Java files in the commit transaction are classified as failure-prone. We are only interested in identifying failure-prone files, not the number of times a file was involved in a bug fix, as this is not necessary for our study. Failure-prone files are grouped into their corresponding versions using the version field of the bug report. This technique for mapping bug fixes to bug reports has been used by other researchers [5, 14, 15, 16]. We used the repository mining feature of SemDiff [4] to retrieve the commit transactions of Eclipse from CVS, and the Bugzilla connector of Mylyn[3] for communicating with the Eclipse bug database.

[3] www.eclipse.org/mylyn/

3.2 The Equal-Blame Approach

The main result of the work of Schröter et al. is that the use of some import statements is indicative of increased risk of failure of a component. In Java, a source file can import either a *type*, such as a class or an interface (for example, `import org.eclipse.ui.IViewSite;`), or a *package* (for example, `import org.eclipse.ui.*;`). Package imports make all the types within a package available to the source file, although only a handful of the types may actually be used. In studying the correlation between imports and component failures, Schröter et al. considered all the types imported by a component with no regard of whether the types are unused, used but irrelevant to the failures, or used and relevant to the failures of the component.

To replicate their study, we needed to determine for each imported type or package I in version k of the software, both the total number of the Java files using I ($F^I_{\mathbf{all}}$), and the total number of the Java files using I with failures ($F^I_{\mathbf{fail}}$). We identified the imported types and packages for each Java files through syntactic analysis to obtain $F^I_{\mathbf{all}}$, and we used the knowledge of failure-prone files gained as explained in the previous section to obtain $F^I_{\mathbf{fail}}$. We will use $F^I_{\mathbf{all}}$ and $F^I_{\mathbf{fail}}$ in Section 4 to identify the imported types whose usage is statistically correlated with higher levels of failures.

3.3 The Failure-Associated Type Approach

The work of Schröter et al. assigned equal blame to all the import statements for the failure of a component, even unused imports. For example, consider the buggy file in Figure 1: the method `sayIt` occasionally throws a null pointer exception because of the `null` value passed to the method `m1`, called on an object of type `package3.Y`. Although the only imported type referenced within the method `sayIt` is `package3.Y`, the equal-blame approach would assign responsibility for the bug to all the three imported types (i.e., `package1.W`, `package2.X`, and `package3.Y`) of the source file even though the type `package1.W` is unused in the source file, and based on the fix to the bug (see Figure 2), the type `package1.X` was not related to the source of the problem. The component-import relationship alone capture a less precise correlation between type usage and failure-proneness. We hypothesize that looking at a more fine-grained level may provide an increased understanding of the links betweens problem domains and failures.

To test this hypothesis, we introduce the concept of a *failure-associated type*. We define a failure-associated type as any type referenced within methods modified in order to fix a failure. The failure-associated type approach captures the relationship between component failure and the imported types referenced at the method level, and is therefore more fine-grained than the equal-blame approach.

```
import package1.W;
import package2.X;
import package3.Y;

class Foo {
  void sayIt(Y obj1){
     obj1.m1("", true);
  }

  void doIt(X obj2){
     obj2.m2();
  }
}
```

Figure 2. Sample bug fix: the bug is fixed by replacing *null* with an empty string in the `sayIt` method.

Identifying failure-associated types: Bug fix commit transactions provide information about the code responsible for a bug and how the bug was fixed. This information can be obtained by comparing the revision of the file in which a bug was identified (e.g., revision r_i) against the revision in which the bug was fixed (e.g., revision r_{i+1}). To identify failure-associated types, we compared revision r_i with revision r_{i+1} to identify the methods that were modified in order to fix the bug. Next, using static analysis, we analyzed the methods modified to fix bugs to identify the types referenced within these methods. Revisiting our example of Figure 1 and its fix (Figure 2), our failure-associated type will be the type `package3.Y` alone since only the method `sayIt` was modified to fix the bug, and only the object `obj1` of type `Y` is referenced within `sayIt`. We consider every type referenced within modified methods as participants in the fix of the bug. We identify failure-associated packages by collapsing the types into their enclosing packages. For example, the type `package3.Y` is collapsed into the package `package3.*`.

A bug fix commit transaction includes only a subset of the program source code (i.e., the files modified to fix the bug). This makes it difficult to reconstruct the complete type hierarchy of a program and to identify the packages within which the types are defined. We used the partial program analysis feature [3] of SemDiff [4] to resolve these problems. Partial program analysis provides support for analyzing partial Java programs by inferring the unknown types given a subset of a program's source code. SemDiff also provides support for obtaining the fully qualified names of the types referenced within methods.

Mapping failure-associated types to methods: To determine whether or not the use of a failure-associated type, I, is correlated with higher levels of failures in version k of a software system, we needed both the total number of the methods using I ($M^I_{\mathbf{all}}$), and the total number of the methods using I with failures ($M^I_{\mathbf{fail}}$). We obtain $M^I_{\mathbf{fail}}$ from the failure-associated types data, and $M^I_{\mathbf{all}}$ from the source code of version k through static analysis.

Table 1. The average likelihood of finding a file or a method that was involved in a fix due to a failure.

	total	*with failure*	*likelihood of failure*
Eclipse 2.0			
- Files	5,907	2,910	0.4926
- Methods	56,531	8,367	0.1480
Eclipse 3.0			
- Files	8,243	2,972	0.3605
- Methods	81,769	11,128	0.1361

4 Data Analysis

4.1 Identifying Problematic Imports

We distinguished *problematic imports* — imported types or packages whose usage is statistically correlated with higher levels of failures — from *non-problematic* imports through a three step process used by Schöter et al. First, for every imported type or package I, we computed the likelihood of failure, $p(\text{fail}|I)$, of a file or a method that uses I. The likelihood of finding a Java file with a failure given the use of I corresponds to the equal-blame approach, whereas the likelihood of finding a method with a failure given the use of I corresponds to the failure-associated type approach.

The likelihood of a file that uses I to be fixed due to a failure is computed as:

$$p(\text{fail}|I) = \frac{F^I_{\text{fail}}}{F^I_{\text{all}}}$$

The likelihood of a method that uses I to be fixed due to a failure is computed as:

$$p(\text{fail}|I) = \frac{M^I_{\text{fail}}}{M^I_{\text{all}}}$$

For example, the type `org.eclipse.jdt.core.IMethod` was used in 154 files, and 152 of the files had to be fixed because of a failure in version 3.0 of Eclipse. The likelihood of finding a file that uses `org.eclipse.jdt.core.IMethod` with a failure is therefore 0.99 ($152/154$). We computed the likelihood of finding a component with failure given the use of the imported types and packages from 50 plug-ins of Eclipse versions 2.0 and 3.0 using both the equal-blame approach and the failure-associated type approach.

Table 2. The proportion of *equal-blame* imports that are also failure-associated types.

	total	*are also failure-associated types?*	
Eclipse 2.0			
- type imports	543	324	59%
- package imports	66	52	78%
Eclipse 3.0			
- type imports	635	346	54%
- package imports	90	73	81%

Next, for each imported type or package I, we computed the likelihood of finding a file or method with failure in which I was not used, $p(\text{fail}|\neg I)$. For example, there are 8,243 files in version 3.0 of Eclipse, and 2,972 of the files were fixed due to failures (see Table 1). This implies the type `org.eclipse.jdt.core.IMethod` was not used in 8089 (8243 - 154) files, and 2820 (2972 - 152) of those files had to be fixed due to failures. Therefore, $p(\text{fail}|\neg I)$ for `org.eclipse.jdt.core.IMethod` is 0.35 (2820/8089).

Finally, using the chi-square test of independence ($p < 0.05$), we determined whether the observed difference between $p(\text{fail}|I)$ and $p(\text{fail}|\neg I)$ is statistically significant — that is, reflects the effect of the the presence or absence of I. The imported types or packages for which this difference is significant are said to be correlated with the failure of the components in which they are used. In general, if $p(\text{fail}|I)$ is higher than $p(\text{fail}|\neg I)$, and their difference is shown to be statistically significant using chi-square, then I is considered *problematic*. On the other hand, if $p(\text{fail}|I)$ is lower than $p(\text{fail}|\neg I)$, and their difference is shown to be statistically significant using chi-square, then I is considered *non-problematic*. Table 1 shows the average likelihood for finding a file or a method that was involved in a fix due to a failure in versions 2.0 and 3.0 of Eclipse. Tables 5 to 8 shows a ranked list of some of the imported types and packages for which $p(\text{fail}|I)$ was determined to be significant for both approaches. The prefix "..." of the imports stands for `org.eclipse`. For example, the use of the type `...jdt.internal.compiler.util.CharOperation` is correlated with the highest risk of failure, in version 2.0, since all of the 133 files in which it was used had to be fixed due to a failure. On the other hand, the use of `...corext.refactoring.Assert` is correlated with a lower risk of failure since less than 12% of the 89 files in which it was used had to be fixed due to a failure (see table 5).

4.2 Discussion

An inspection of the data seems to provide little points of agreement between the two studies. For instance, the equal-blame approach identified the import `...jdt.internal.compiler.util.CharOperation` as associated with the highest risk of failure when used in version 2.0 of Eclipse (see Table 5), however the failure-associated type approach identified the same import as being correlated with lower levels of failure (see Table 7). We investigated the following questions to study the discrepancy:

How many *equal-blame* imports are also *failure-associated types?* Table 2 shows the number of the equal-blame imported types and packages for versions 2.0 and 3.0 of Eclipse, and the proportion of these statements that were also failure-associated types. For instance, 59% of the 543 imported types of version 2.0 of Eclipse were also failure-associated types — that is, referenced within methods fixed due to a failure. On average, less than 56% of the imported types, and 80% of the package imports of the equal-blame approach were failure-associated types when we looked at the data of both versions 2.0 and 3.0 of Eclipse. In summary, not every imported type or package in a failure-prone component is associated with its failures.

What proportion of imports were shown to be correlated with a higher risk of failure by the *equal-blame approach*, but not the *failure-associated type* approach?

Some types or packages are widely imported in Eclipse. The equal-blame approach which assigns equal responsibility to all the imports of a file with a failure could push widely used imports to the top of the "high risk of failure" list. We suspect that was the case with the `...jdt.internal.compiler.util.CharOperation` import that was shown to be correlated with higher risk of failure by the equal-blame approach, but not the failure-associated type approach. We determined the proportion of imports reported to be correlated with higher risk of failure by the equal-blame approach, but not the failure-associated type approach by looking for high risk imports of the equal-blame approach that are not associated with higher risk of failure in the failure-associated type approach. Table 3 shows the proportion of imports correlated with higher risk of failure in the equal-blame approach, but not the failure-associated type approach. On average, up to 75% of the type imports, and 45% of the package imports are reported to be correlated with higher risk of failure in the equal-blame approach, but not the failure-associated type approach (both versions combined). In other words, up to 75% of the type imports, and 45% of the package imports that may not have contributed to the failures in versions 2.0 and 3.0 of Eclipse were reported to be correlated with a higher risk of failure by the equal-blame approach. This observation leads to our next question:

Table 3. The proportion of imports correlated with a higher risk of failure in the *equal-blame* approach, but not the *failure-associated type* approach.

	total	***not** correlated with a higher risk of failure in **failure-associated types**?*	
Eclipse 2.0			
- type imports	421	333	79%
- package imports	25	12	48%
Eclipse 3.0			
- type imports	451	321	71%
- package imports	68	28	42%

Table 4. The proportion of imports correlated with a higher risk of failure in the *failure-associated type* approach, but not the *equal-blame* approach.

	total	***not** correlated with a higher risk of failure in **equal-blame**?*	
Eclipse 2.0			
- type imports	188	100	53%
- package imports	101	88	87%
Eclipse 3.0			
- type imports	281	151	53%
- package imports	199	159	79%

Are there imported types correlated with higher risk of failure in the *failure-associated type* approach, but not the *equal-blame* approach? The answer is yes. Whereas the equal-blame approach is biased towards widely used imported types, it is also biased against imports that are not widely used. Consequently, imported types with likelihood of failures greater than that of the population from which they were drawn but not widely used may be reported as not being statistically significant, and thus, not correlated with higher risk of failure. For instance, files that used the import `org.eclipse.jdt.core.search.SearchEngine` in version 2.0 of Eclipse had a likelihood of failure greater than that of their population, but the import was reported as not being correlated with higher risk of failure by the equal-blame approach. However, a look at the failure-associated types revealed that the use of the import `org.eclipse.jdt.core.search.SearchEngine` is correlated with higher risk of failure. By looking at the imports referenced within methods fixed due to a failure and not the file-import relationships, the failure-associated type approach reduces the bias towards popular imports and against not widely used imports.

Table 4 shows the proportion of imports whose usage is correlated with higher risk of failure in the failure-associated type approach, but not the equal-blame approach. On average, up to 53% of the type imports, and 83% of the package imports that were failure-associated types and correlated with higher risk of failure in versions 2.0 and 3.0 of Eclipse were not identified as being correlated with higher risk of failure by the equal-blame approach.

Summary: The component-import relationship may identify imported types not associated to the fix of failures as being highly correlated to failures, and the imported types associated to the fix of failures as being not risky. Whereas the component-import relationship is easy to compute and straightforward to interpret, it provides a coarse relationship between the problem domain and failures. By looking at the failure-associated types, we were able to identify the specific imported types and packages relevant to the context of the failures, and also to eliminate those import statements, which although widely used, are unrelated to the failures. For example, the equal-blame approach identified the domain `org.eclipse.jdt.internal.compiler.parser` as being risky. Using the failure-associated types, we learned that even though both the `Parser` and `Scanner` classes belong to this domain, the use of `Scanner` is associated with a higher risk of failure, whereas the use of `Parser` is associated with a lower risk of failure. Failure-associated types therefore provide an increased understanding of the links between problem domains and failures, and takes us a step closer to understanding why some problem domains are more prone to failures than others.

Threats to Validity: In studying the correlation between the use of import statements and the risk of failure of software components, we made assumptions that may not be representative of every software project.

We assumed that references to words such as "bug" or "fix" together with a number in the log message refers to fixes to failures. While this assumption may not always be true, it remains the most widely used techniques for identifying bug fix commit transactions [7, 12, 14].

A bug fix commit transaction provides only a subset of the entire source code — the source code of the files modified to fix failures. Thus, the type of some objects referenced within methods are unknown. We used the Partial Program Analysis (PPA) feature of SemDiff to infer unknown types given just a subset of the code.

Table 5. Problematic and non-problematic *type* imports in Eclipse version 2.0 (*equal-blame* approach).

type imports	$F^I_{\mathbf{fail}}$	$F^I_{\mathbf{all}}$	$p(\text{fail}\|I)$
....nal.compiler.util.CharOperation	113	113	1.00
...core.resources.IWorkspaceRoot	46	46	1.00
...jface.action.IMenuListener	22	22	1.00
...jdt.internal.compiler.Compiler	11	11	1.00
...			
...core.resources.IMarkerDelta	12	13	0.9231
...ui.IWorkbenchPart	47	51	0.9216
...swt.events.SelectionListener	34	37	0.9189
...jdt.core.IJavaModelMarker	10	11	0.9091
...			
...team.internal.ui.Policy	25	28	0.8929
...ui.IWorkbenchPage	58	65	0.8923
...ui.IEditorPart	66	74	0.8919
...ui.texteditor.IDocumentProvider	16	18	0.8889
...swt.events.ModifyEvent	32	36	0.8889
...			
...core.runtime.IPath	233	292	0.7979
...jdt.internal.ui.JavaPlugin	199	250	0.7960
...jdt.core.IPackageDeclaration	15	19	0.7895
...swt.events.SelectionEvent	74	95	0.7789
...jdt.core.JavaCore	149	194	0.7680
...			
...jdt.core.IMember	55	83	0.6627
...ui.IWorkbenchSite	47	71	0.6620
..jdt.core.JavaModelException	278	458	0.6070
...jdt.core.ICompilationUnit	155	258	0.6008
...			
...pde.internal.core.PDECore	19	53	0.3585
...jdt.core.dom.SimpleName	9	30	0.3000
...jdt.core.dom.CompilationUnit	8	28	0.2857
...jdt.core.dom.ITypeBinding	8	31	0.2581
...jdt.core.dom.ASTNode	8	48	0.1667
...corext.refactoring.Assert	10	89	0.1124
...			

Table 6. Problematic and non-problematic *package* imports in Eclipse version 2.0 (*equal-blame* approach).

package imports	$F^I_{\mathbf{fail}}$	$F^I_{\mathbf{all}}$	$p(\text{fail}\|I)$
...jdt.internal.compiler.ast.*	31	31	1.00
...jdt.internal.compiler.*	19	19	1.00
...jdt.core.compiler.*	17	17	1.00
...jdt.internal.compiler.parser.*	17	17	1.00
...			
...jdt.internal.compiler.impl.*	48	50	0.9600
...jdt.internal.compiler.lookup.*	92	98	0.9388
...compare.structuremergeviewer.*	13	14	0.9286
...			
...update.internal.ui.parts.*	11	13	0.8462
...jface.dialogs.*	22	27	0.8148
org.xml.sax.*	13	17	0.7647
..update.internal.ui.model.*	20	29	0.6897
org.w3c.dom.*	30	44	0.6818
org.eclipse.compare.*	29	43	0.6744
...core.resources.*	109	187	0.5829
...swt.widgets.*	96	240	0.4000
...			
...update.ui.forms.internal.*	27	83	0.3253
...pde.core.plugin.*	41	131	0.3130
...pde.internal.ui.*	30	105	0.2857
...swt.graphics.*	25	91	0.2747
...jface.resource.*	6	23	0.2609
...pde.internal.ui.editor.*	14	57	0.2456
...jface.wizard.*	11	45	0.2444
...ui.part.*	7	22	0.2381
...pde.internal.core.schema.*	6	28	0.1852
...pde.internal.core.ischema.*	14	79	0.1772
...			

Table 7. Problematic and non-problematic *type* imports in Eclipse version 2.0 (*failure-associated type* approach).

type imports	$M^I_{\mathbf{fail}}$	$M^I_{\mathbf{all}}$	$p(\text{fail}\|I)$
...jdt.launching.JavaRuntime	59	61	0.9672
...team.internal.ccvs.ui.Policy	174	197	0.8832
...team.internal.ccvs.core.Policy	136	159	0.8553
...debug.core.DebugPlugin	109	134	0.8134
...jdt.core.dom.ASTConverter	57	80	0.7125
...core.boot.BootLoader	28	40	0.7000
...jdt.launching.IVMInstall	49	75	0.6533
...jdt.core.dom.BindingResolver	23	38	0.6053
...jdt.core.JavaConventions	23	38	0.6053
org.w3c.dom.Document	26	44	0.5909
...ui.IActionBars	26	44	0.5909
...jdt.core.Flags	26	46	0.5652
...			
...jdt.core.Signature	51	109	0.4679
...jdt.core.JavaCore	145	323	0.4489
...jface.dialogs.MessageDialog	139	364	0.3819
org.w3c.dom.Node	31	84	0.3690
...team.core.TeamException	60	164	0.3659
...jdt.core.JavaModelException	64	176	0.3636
...jdt.core.dom.CompilationUnit	17	48	0.3542
...			
...ui.IFileEditorInput	16	54	0.2963
...swt.widgets.Display	70	237	0.2954
...jdt.core.dom.TypeDeclaration	14	48	0.2917
...ui.help.WorkbenchHelp	95	333	0.2853
...jface.viewers.LabelProvider	15	56	0.2679
...			
...swt.events.SelectionAdapter	62	328	0.1890
...core.resources.IResource	232	1771	0.1310
...jdt.core.ICompilationUnit	158	1268	0.1246
....nal.compiler.util.CharOperation	20	258	0.0755
...			

Table 8. Problematic and non-problematic *package* imports in Eclipse version 2.0 (*failure-associated type* approach).

package imports	$M^I_{\mathbf{fail}}$	$M^I_{\mathbf{all}}$	$p(\text{fail}\|I)$
...jdt.internal.launching.*	97	138	0.7029
...team.core.*	146	211	0.6919
...debug.core.*	527	766	0.6880
...pde.internal.build.*	32	51	0.6275
...jdt.launching.*	195	313	0.6230
...team.ui.*	26	42	0.6190
...debug.internal.ui.*	323	546	0.5916
...update.internal.core.*	198	340	0.5824
...			
org.xml.sax.*	38	77	0.4935
...ui.externaltools.internal.ui.*	56	114	0.4912
...core.internal.runtime.*	67	142	0.4718
...jdt.internal.debug.ui.*	373	796	0.4686
...team.internal.ccvs.ui.*	269	592	0.4544
...jdt.internal.compiler.ast	50	158	0.3165
...			
..nal.corext.refactoring.structure.*	49	179	0.2737
...debug.core.model.*	199	731	0.2722
...jface.resource.*	65	243	0.2675
...			
...jface.viewers.*	581	3016	0.1926
org.w3c.dom.*	53	277	0.1913
...swt.widgets.*	806	4373	0.1843
org.eclipse.ui.*	330	2582	0.1278
...			

An empirical study on four large open source systems showed that PPA can correctly recover up to 91% of the types of a partial program [3]. This implies that up to 10% of the method-to-type relationships of our failure-associated type approach may be erroneous. We do not expect this to affect the conclusions of our study since the error margin is much smaller than the difference between the equal-blame and the failure-associated type approach.

In their work, Schöter et al. looked at just post-release failures — failures observed within six months after a version of Eclipse was released to the public. Our study made no such distinction; we looked at all the commit transactions that were matched to failures in Bugzilla for all the plug-ins. We compared the data reported by Schöter et al. to our replication study, and all the imported types reported to be risky by Schöter et al. were also risky in our data. However, only 50% of the package imports reported to be risky by Schöter et al. were risky in our data. We do not expect this difference to influence the conclusions of our study since the comparison between the equal-blame approach and the failure-associated type approach was made from the same data which looked at all the commit transactions that were matched to failures in Bugzilla.

The failure-associated type concept correlates the use of classes, interfaces, or packages to component failure by looking at the elements referenced within methods that were fixed due to failures. Although failure-associated types provide more precise data about the types associated with failures, it ignores other contributing factors, such as the classes or packages that may be indirectly referenced from within methods, that may influence the behavior of a method. We make no claim that the types referenced within methods are exclusively responsible for its failures, but we consider them to be part of the context of failure, and thus, a contributing factor.

5 Related Work

Several researchers have investigated the correlation between software failures and various attributes of a software system such as its structural properties, the software process, and the software change history. In this paper, we investigated the correlation between imports and component failure, a work inspired by the study of Schöter et al. We discuss other representative examples of publications that have investigated the relation between component failures and other attributes of software systems.

Software Metrics: The relationship between software metrics and software failures is based on the premise that the more complex the code of a component is, the more failure-prone the component is likely to be. Chidamber and Kemerer (CK) were amongst the first to propose design complexity metrics of a system [2]. The CK metric suite consists of six object-oriented metrics that capture the design complexity of classes. The relationship between the CK metrics and software failures was first validated by Basili et al. [1]. They used eight student projects and software failure data collected by an independent team of professional programmers while testing the student projects. Basili et al. reported that all but one — the Lack of cohesion among methods metric — of the CK metrics were significantly correlated with software failures.

Nagappan et al. also investigated the correlation between several code complexity metrics and the post-release failures of six Microsoft systems [11]. They reported some complexity metrics to be significantly correlated with post-release failures. However, the correlations were mainly project specific since they could not find a single metric that correlated with failures in all of the six projects. Zimmermann and Nagappan proposed new metrics based on the dependencies between software components [17]. They captured the data and call dependencies between the binaries of Windows Server 2003 on dependency graphs, and used several network analysis techniques on the graphs to identify network measures that correlate with post-release failures. Zimmermann and Nagappan observed that network measures such as the closeness of the binaries and the clique[4] size were correlated with failures, and in most cases, outperformed traditional complexity metrics, such as McCabe's complexity, in predicting defect-prone components. Our work also studied the correlation between dependencies and failures, however in our case, dependencies are represented by the import statements in Java, not the data and call dependencies.

Historical Data: Researchers have mined version archives and bug tracking systems to investigate attributes of the change process that may be correlated with failures. In a study of the failure and change history of a 1.5 million LOC system, Graves et al. reported correlations between the number of changes to a component and the number of failures [6]. For instance, components with a higher number of changes in the past were reported to be more failure-prone than components with fewer changes. Also, they observed no evidence to suggest that components developed by a larger group of programmers were more failure-prone than those developed by a smaller group. Nagappan and Ball also investigated the correlation between code churn — a measure of the amount of the changes made to a component over a period of time — and software failures [9]. They also observed that code that was changed many times before the release of a software was likely to have more post-release failures than code that was changed less often over the same period. Finally, a comparative study of three

[4] A clique is a set of binaries in which every pair has a dependency relationship.

defect predictors — Naive Bayes, Decision Trees, and Logistic Regression — by Moser et al. [8] reached the conclusion that defect predictors based on change data significantly outperform predictors based on code metrics. Further, combining change data and code metrics did not produce predictions better than those from using change data only.

6 Conclusions

The use of some imported types or packages in Java increases the risk of failure of a software component. We investigated the correlation between imported types and component failure both at the component-import level (that is, by looking at all the import statements used in a component with failures) and the method-import level (that is, by looking at the *failure associated types* — the imported types or packages referenced within methods fixed due to failures).

The component-import level captures a coarser relationship between import statements and failures, and does not distinguish unused imports and imports not associated to the failure of a component from those imported types most closely associated to its failures. Consequently, import statements unused in a component or not correlated to the failures of a component may be reported as being risky. For example, in Eclipse, less than 25% of the type imports, and less than 55% of the package imports reported to be highly correlated with failures by the component-import relationships were actually correlated with failures when we looked at the failure associated types.

The failure-associated type approach is capable of identifying the imported types and packages of a domain associated to the context of a component's failures, and to eliminate both unused and irrelevant import statements, which although widely used, are not correlated to failures. Failure-associated types therefore support an increased understanding of the links between problem domains and failures, and takes us a step closer to understanding why some problem domains are more prone to failures than others.

Acknowledgments

The authors would like to thank Barthélémy Dagenais for his assistance with the use of SemDiff and for his valuable comments on this paper. We also thank Thomas Zimmermann for his valuable comments on this paper. This work was supported by NSERC.

References

[1] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *Transaction of Software Engineering*, 22(10):751–761, 1996.

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *Transaction of Software Engineering*, 20(6):476–493, 1994.

[3] B. Dagenais and L. Hendren. Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, pages 313–328, 2008.

[4] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th International Conference on Software Engineering*, pages 481–490, 2008.

[5] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger. Analysing software repositories to understand software evolution. In *Software Evolution*, pages 37–67. 2008.

[6] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *Transactions of Software Engineering*, 26(7):653–661, 2000.

[7] S. Kim, K. Pan, and J. E. E. James Whitehead. Memories of bug fixes. In *Proceedings of the 14th International Symposium on Foundations of Software Engineering*, pages 35–45, 2006.

[8] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190, 2008.

[9] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, pages 284–292, 2005.

[10] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, pages 452–461, 2006.

[11] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, pages 452–461, 2006.

[12] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 529–540, 2007.

[13] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[14] A. Schröter, T. Zimmermann, and A. Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 18–27, 2006.

[15] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[16] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, 2003.

[17] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, pages 531–540, 2008.

[18] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 2–6, May 2004.

Appendix

Table 9. Problematic and non-problematic *type* imports in Eclipse version 3.0 (*equal-blame* approach).

type imports	$F^I_{\mathbf{fail}}$	$F^I_{\mathbf{all}}$	$p(\text{fail}\|I)$
...jdt.ui.JavaUI	95	95	1.00
...jface.dialogs.ErrorDialog	87	87	1.00
...core.resources.IWorkspaceRoot	58	58	1.00
...jdt.core.dom.Expression	51	51	1.00
...			
...jface.viewers.Iselection	299	302	0.9901
...ui.IWorkbenchPart	175	177	0.9887
...jdt.core.Imethod	152	154	0.9870
...jface.viewers.IStructuredSelection	356	363	0.9807
...jdt.core.dom.ASTVisitor	36	37	0.9730
...jdt.internal.compiler.util.Util	30	31	0.9677
...			
...ui.ISharedImages	45	50	0.9000
...jface.wizard.Wizard	43	48	0.8958
...jdt.core.compiler.IProblem	77	86	0.8953
...jdt.core.ISourceRange	67	75	0.8933
...ui.texteditor.ITextEditor	74	83	0.8916
...			
...jdt.core.compiler.IScanner	24	30	0.8000
...jdt.core.IBuffer	23	29	0.7931
...ui.IWorkbenchSite	83	106	0.7830
...jdt.internal.corext.SourceRange	18	23	0.7826
...core.resources.IResourceStatus	29	40	0.7250
...debug.core.DebugException	134	188	0.7128
...			
...ltk.ui.refactoring.RefactoringWizard	20	55	0.3636
...jface.text.templates.TemplateContextType	7	20	0.3500
...			

Table 10. Problematic and non-problematic *package* imports in Eclipse version 3.0 (*equal-blame* approach).

package imports	$F^I_{\mathbf{fail}}$	$F^I_{\mathbf{all}}$	$p(\text{fail}\|I)$
...swt.custom.*	40	40	1.00
...pde.internal.build.*	26	26	1.00
...pde.ui.*	18	18	1.00
...pde.internal.*	13	13	1.00
...jdt.internal.compiler.ast.*	50	51	0.9804
...jdt.internal.compiler.impl.*	38	39	0.9744
...jdt.internal.compiler.lookup.*	113	116	0.9741
...jdt.core.search.*	25	26	0.9615
...jface.wizard.*	65	69	0.9420
...update.internal.ui.model.*	16	17	0.9412
...ui.actions.*	28	30	0.9333
...jface.viewers.*	208	223	0.9327
...swt.graphics.*	81	87	0.9310
...core.internal.resources.*	12	13	0.9231
...update.configuration.*	57	63	0.9048
...swt.widgets.*	296	339	0.8732
...jface.operation.*	31	37	0.8378
...			
...pde.internal.core.ifeature.*	28	67	0.4179
...pde.internal.ui.parts.*	16	41	0.3902
...pde.internal.core.ischema.*	22	58	0.3793
...ui.forms.*	6	21	0.2857
org.w3c.dom.*	11	49	0.2245
...			

Table 11. Problematic and non-problematic *type* imports in Eclipse version 3.0 (*failure-associated type* approach).

type imports	$M^I_{\mathbf{fail}}$	$M^I_{\mathbf{all}}$	$p(\text{fail}\|I)$
...core.internal.content.ContentType	43	45	0.9556
...ui.internal.progress.ProgressMessages	40	44	0.9091
...ltk.core.refactoring.Refactoring	43	51	0.8431
...jdt.internal.formatter.Scribe	94	121	0.7769
...team.internal.ui.TeamUIPlugin	39	51	0.7647
...jdt.ui.JavaUI	29	44	0.6591
org.w3c.dom.Document	50	77	0.6494
...search2.internal.ui.InternalSearchUI	27	42	0.6429
...team.internal.ui.Policy	59	96	0.6146
...debug.internal.ui.DebugUIMessages	30	49	0.6122
...ui.help.WorkbenchHelp	57	95	0.6000
...ui.internal.WorkbenchImages	28	47	0.5957
...core.runtime.Platform	144	247	0.5830
...			
...jdt.core.dom.rewrite.ASTRewrite	93	225	0.4133
...core.runtime.IProgressMonitor	283	694	0.4078
...ui.IActionBars	22	54	0.4074
...jdt.debug.core.IJavaStackFrame	19	47	0.4043
..jdt.internal.ui.util.SWTUtil	17	43	0.3953
...swt.graphics.Rectangle	17	45	0.3778
org.osgi.framework.Bundle	17	45	0.3778
...jdt.core.dom.ASTParser	15	40	0.3750
...core.internal.utils.Assert	24	64	0.3750
...swt.widgets.Label	203	1063	0.1910
...jface.dialogs.IDialogSettings	54	283	0.1908
...jface.text.ITextViewer	81	457	0.1772
...jdt.core.IType	195	1203	0.1621
...swt.widgets.Shell	132	1220	0.1082
...jface.viewers.TableViewer	132	1255	0.1052
...jdt.core.dom.ASTNode	103	1050	0.0981
...			

Table 12. Problematic and non-problematic *package* imports in Eclipse version 3.0 (*failure-associated type* approach).

package imports	$M^I_{\mathbf{fail}}$	$M^I_{\mathbf{all}}$	$p(\text{fail}\|I)$
...jdt.internal.compiler.*	86	86	1.00
...core.internal.events.*	40	51	0.7843
...core.internal.jobs.*	64	85	0.7529
...team.internal.ui.*	113	165	0.6848
...ui.help.*	57	95	0.6000
...debug.internal.ui.*	236	394	0.5990
...pde.internal.build.*	49	3	0.5904
...jdt.core.search.*	90	153	0.5882
...jdt.internal.compiler.util.*	66	142	0.4648
...jdt.internal.debug.core.*	35	76	0.4605
...search2.internal.ui.*	39	87	0.4483
...			
...			
...jface.window.*	45	187	0.2406
...jdt.core.dom.*	531	2403	0.2210
...team.internal.core.*	66	299	0.2207
...core.internal.resources.*	111	508	0.2185
...ltk.ui.refactoring.*	24	116	0.2069
...swt.graphics.*	113	592	0.1909
...ui.part.*	136	752	0.1809
...swt.custom.*	259	1433	0.1807
...jface.action.*	314	1902	0.1651
...pde.internal.ui.*	80	906	0.0883
...team.core.subscribers.*	22	285	0.0772
...			