Java Generics Adoption: How New Features are Introduced, Championed, or Ignored

Chris Parnin
College of Computing
Georgia Institute of
Technology
Atlanta, Georgia
chris.parnin@gatech.edu

Christian Bird
Microsoft Research
Redmond, Washington
cbird@microsoft.com

Emerson Murphy-Hill
Dept. of Computer Science
North Carolina State
University
Raleigh, North Carolina
emerson@csc.ncsu.edu

ABSTRACT

Support for generic programming was added to the Java language in 2004, representing perhaps the most significant change to one of the most widely used programming languages today. Researchers and language designers anticipated this addition would relieve many long-standing problems plaguing developers, but surprisingly, no one has yet measured whether generics actually provide such relief. In this paper, we report on the first empirical investigation into how Java generics have been integrated into open source software by automatically mining the history of 20 popular open source Java programs, traversing more than 500 million lines of code in the process. We evaluate five hypotheses, each based on assertions made by prior researchers, about how Java developers use generics. For example, our results suggest that generics do not significantly reduce the number of type casts and that generics are usually adopted by a single champion in a project, rather than all committers.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Data types and structures; F.3.3 [**Studies of Program Constructs**]: Type structure

General Terms

Languages, Experimentation

Keywords

generics, Java, languages, post-mortem analysis

1. INTRODUCTION

Programming languages and tools evolve to match industry trends, revolutionary shifts, or refined developer tastes. But not all evolutions are successes; the technology landscape is pocked with examples of evolutionary dead-ends and dead-on-arrival concepts.

Far too often, greatly heralded claims and visions of new language features fail to hold or persist in practice. Discussions of the costs and benefits of language features can easily devolve into a religious war with both sides armed with little more than anecdotes [13]. Empirical evidence about the adoption and use of past language features should inform and encourage a more rational discussion when designing language features and considering how they should be deployed. Collecting this evidence is not just sensible but a responsibility of our community.

In this paper, we examine the adoption and use of generics, which were introduced into the Java language in 2004. When Sun introduced generics, they claimed that the language feature was "a long-awaited enhancement to the type system" that "eliminates the drudgery of casting." Sun recommended that programmers "should use generics everywhere [they] can. The extra efforts in generifying code is well worth the gains in clarity and type safety."[1] But is it?

Here, we take the first look at how features of Java generics, such as type declarations, type-safe collections, generic methods, and wildcards, have been introduced and used in real programs. With the benefit of six years of hindsight, we investigate how the predictions, assertions, and claims that were initially made by both research and industry have played out in the wild. Further, we investigate the course and timeline of adoption: what happens to old code, who buys in, how soon are features adopted, and how many projects and people ignore new features? The results allow us to adjust our expectations about how developers will adopt future language features.

We make the following contributions in this paper:

- we enumerate the assumptions and claims made in the past about Java generics (Section 3);

- we investigate how 20 open source projects have used—and have not used—Java generics (Section 5 to 7); and

- we discuss the implications of the adoption and usage patterns of generics (Section 8).

2. AN OVERVIEW OF GENERICS

In this section we briefly describe the motivation and use of generics. In an effort to maintain consistent terminology, we present in **bold** the terms that we use in this paper, drawing from standard terminology where possible. Readers who are familiar with Java generics may safely skip this section.

---

[1] http://download.oracle.com/javase/1.5.0/docs/guide/language/generics.html

2.1 Motivation for Generics

In programming languages such as Java, type systems can ensure that certain kinds of runtime errors do not occur. For example, consider the following Java code:

```
List  l = getList();
System.out.println(l.get(10));
```

This code will print the value of the 10th element of the list. The type system ensures that whatever object `getList()` returns, it will understand the `get` message, and no runtime type error will occur when invoking that method. In this way, the type system provides safety guarantees at compile time so that bugs do not manifest at run time.

Now suppose we want to take the example a step further; suppose that we know that `l` contains objects of type `File`, and we would like to know whether the tenth file in the `List` is a directory. We might naturally (and incorrectly) write:

```
List  l = getList();
System.out.println(l.get(10).isDirectory());
```

Unfortunately, this leads to a compile-time error, because the return type of the `get` method is specified at compile-time as `Object`. The type checker gives an error because it does not know what types of objects are actually in the list.

In early Java, programmers had two ways to solve this problem, the first is casting, and the second we call **home-grown data structures**. If the programmer implements the casting solution, her code would look like this:

```
List  l = getList();
System.out.println(((File)l.get(10)).isDirectory());
```

The cast is the `(File)` part, which forces the compiler to recognize that the expression `l.get(10)` actually evaluates to the `File` type. While this solves one problem, it causes another; suppose that a programmer at some point later forgets that the list was intended to hold `File`s, and inadvertently puts a `String` into the `List`. Then when this code is executed, a runtime exception will be thrown at the cast. A related problem is that the code is not as clear as it could be, because nowhere does the program explicitly specify what kind of objects the list returned by `getList()` contains.

If the programmer instead implements the home-grown data structure solution, the code will look like this:

```
FileList  l = getList();
System.out.println(l.get(10).isDirectory());
```

Additionally, the programmer would need to create a `FileList` class. This solution also introduces new problems. Perhaps the most significant is the code explosion problem; for each and every list that contains a different type, the programmer will want to create a different special list class, such as `StringList`, `IntegerList`, and `NodeList`. These classes will inevitably contain significant duplication, because they all perform the same functions, differing only by data type.

2.2 Programming with Generics

These problems were solved with the introduction of *generics* to Java in 2004. Generics allow programmers to create their own **generic type declarations** [4] (we call these **generic types**, for short). For example, a programmer can create a **user-defined generic declaration** for a list like so:

```
class MyList<T>{
  List  internal ;
```

```
  public T get(int index){
    return (T)internal.get(index);
  } ...
```

In this code, the `T` is called the **formal type parameter**. The programmer can use her `MyList` class by instantiating the formal type parameter by using a **type argument** [4], such as `Integer` or `File` in the following examples:

```
MyList<Integer> intList = new MyList<Integer>();
MyList<File> fileList = new MyList<File>();
```

Each place where a generic type declaration is invoked (in this example, there are four) is known as a **parameterized type** [5]. On the first line, the programmer has declared the type of the `intList` object so that the compiler knows that it contains objects of type `Integer`, and thus that the expression `intList.get(10)` will be of type `Integer`. The result is that the client code is both type safe and clearly expresses the programmer's intent. The programmer can also use generic type declarations without taking advantage of generics by using them as **raw types**, such as `MyList objectList`, in which case the expression `objectList.get(10)` will be of type `Object`.

In addition to creating their own generic type declarations, programmers can use generic type declarations from libraries. For example, software developers at Sun **generified** [5], or migrated to use generics, the Java collections classes. For instance, the `List` class was parameterized, so that the previous problem could also be solved like so:

```
List<File> l = getList();
System.out.println(l.get(10).isDirectory());
```

In addition to using generics in type declarations, generics can also be applied to individual methods to create **generic methods**, like so:

```
<A> A bigHead(List<A> as1, List<A> as2){
  return as1.get(0) > as2.get(0) ? as1.get(0) : as2.get(0);
}
```

In this code, the programmer can pass to the `bigHead` method two generic lists containing any type, and the type checker will assure that those two types are the same.

3. RELATED WORK

In this section, we discuss previous claims about and studies of generics.

3.1 Claims Regarding Generics

There have been a number of papers and books that have extolled the benefits of using generics in several contexts. We list here a sample of such claims.

In *Effective Java*, Bloch [3] asserts that when a programmer uses non-generic collections, she will not discover errors until run time. Even worse, the error is manifest as a `ClassCastException` when taking an item *out* of a collection, yet to correct the error, she must time-consumingly identify which object was wrongly inserted *into* the collection. By using generics, the type system shows the developer exactly where she inserted the incorrect object, reducing the time to fix the problem.

In their paper on automatically converting Java programs to use generic libraries, Donavan *et al.* [6] assert:

- In pre-generic Java, programmers thought of some classes in pseudo-generic terms and tried to use them in such a way. However, without a generic type system, they would

make inadvertent errors that would show up at runtime. The addition of generics to the type system moves these runtime errors to compile time type errors.

- The type system represents an explicit specification, and generics strengthen this specification. This is better for developers because they can use this strong specification to reason about the program better and are less likely to make mistakes. In addition, the compiler can enforce the specification.

- Prior to generics, programmers that wanted type safe containers would write their own home-grown data structures, increasing the amount of work and likelihood of error, compared to using data structures in libraries. Such structures also "introduce nonstandard and sometimes inconsistent abstractions that require extra effort for programmers to understand."

In his book on C++ templates, Vandevoorde [19] asserts that when the same operations need to be performed on different types, the programmer can implement the same behavior repeatedly for each type. However, if in doing so she writes and maintains many copies of similar code, she will make mistakes and tend to avoid complicated but better algorithms because they are more error prone. She must also deal with all of the difficulties associated with code clones such as making orchestrated changes to coupled clones [10] and perform maintenance more frequently [15].

Naftalin and Wadler [16] claim that generics work "synergistically" with other features of Java such as *for-each* for loops and autoboxing. They also claim that there are now fewer details for the programmer to remember. They also claim that generics can make design patterns more flexible by presenting an example of a visitor pattern that works on a tree with generic elements.

In summary, the claims made by previous authors are:

- Generics move runtime errors to compile time errors.

- Programmers no longer have to manually cast elements from pseudo-generic data structures or methods.

- Typed data collections such as `FileList`, create nonstandard and sometimes inconsistent abstractions.

- Generics prevent code duplication and errors resulting from maintaining multiple typed data collections.

- Generics enhance readability and specification.

- Generics lower cognitive load by requiring the programmer to remember fewer details.

3.2 Empirical Studies

There have been few empirical studies related to the use of generics in Java or parameterized types in object oriented languages in general. Here we discuss the few that exist.

In 2005, Basit *et al.* [1] performed two case studies examining how well generics in Java and templates in C++ allowed what they termed "clone unification." They found that 68% of the code in the Java Buffer library is duplicate and tried to reduce these clones through generification. About 40% of the duplicate code could be removed. They observed that type variation triggered many other non-type parametric differences among similar classes, hindering applications of generics. They also observed heavy cloning in the C++ Standard Template Library as well.

Fuhrer *et al.* [9] implemented refactoring tools that would replace raw references to standard library classes with pa-

rameterized types. In evaluating the refactoring tools on several Java programs, they were able to remove 48.6% of the casts and 91.2% of the compiler warnings.

We are not the first to examine how well features intended to aid programmers live up to their claims. Pankratius *et al.* performed an empirical study aimed at determining if transactional memory actually helped programmers write concurrent code [18]. He found some evidence that transactional memory (TM) did help; students using TM completed their programs much faster. However, they also spent a large amount of time tuning performance since TM performance was hard to predict.

These studies differ from our study in that they investigated generics or another language feature in an artificial or laboratory context, whereas we investigate generics in several natural contexts: open source software. As a result, these studies investigate the ideal impact of generics, while our study investigates their real impact.

4. INVESTIGATION

Our investigation begins with understanding how developers use generics in programs. Are some features of generics widely used and others never touched? Next, we examine claims made about generics and see if the purported benefits of generics are realized in practice. Finally, how does adoption play out — how soon does it occur, what happens to the old code, who buys in?

We start with a data characterization by measuring how widespread generics are among our selected projects and their developers. Then, we examine in detail how that usage varies across the features of generics.

4.1 Investigated Claims

Many claims have been made by language designers and researchers. One claim was that generics reduce the number of runtime exceptions [3]. Ideally, we would like to know how many runtime exceptions each version of a program could throw, but computing this is infeasible due to the state space explosion problem, compounded by the thousands of different versions of many open source projects. Instead, we restate the problem as how the number of casts in a program changes, reasoning that each cast represents a certain probability that a runtime exception will occur:

Hypothesis 1 - When generics are introduced into a codebase, the number of type casts in that codebase will be reduced.

We also investigated the claim of code reduction:

Hypothesis 2 - Introduction of user-defined generics classes reduce code-duplication.

4.2 Adoption Research Questions

Although a wealth of prior literature has examined how open source software (OSS) projects make decisions, assign and accomplish tasks, and organize themselves (e.g. [17, 14, 8]), the nature of adoption of new language features such as Java generics is not clear.

Our first research question centers around how project members embrace new language features such as Java generics. Do they do it together, or do some members still hold out? Even though "benevolent dictatorships" exist in OSS, nearly every open source project's decision-making process

is governed in at least a semi-democratic fashion. Since the decision to use generics has implications directly on the codebase itself (e.g., it may require using a newer JDK or modify popular method signatures impacting all call sites), we expect that there will be project-wide acceptance of generics rather than acceptance by individual members:

Research Question 1 - <mark>Will project members broadly use generics after introduction into the project?</mark>

In addition to a broad consensus for use of generics, a second research question investigates if there will be a concerted effort to replace old code to use the new features. Are the new features compelling enough to fix old code that may contain problems that would be fixed by generics or at least to maintain consistency?

Research Question 2 - <mark>Will there be large-scale efforts to convert old code using raw types to use generics?</mark>

Finally, Java integrated development environments (IDEs) such as Eclipse, Netbeans, and IntelliJ IDEA all support features such as syntax highlighting and semantic analysis to provide auto completion and identify type errors interactively. These tools enable developers to be more productive, but not all IDEs supported generics when they were first introduced. We expect that the choice to use new language features such as generics will in part depend on the tool support available for those features.

Research Question 3 - <mark>Does generics support in the IDE influence adoption?</mark>

4.3 Projects Studied

To test the hypotheses regarding generics, we automatically analyzed 20 open source software projects. We analyzed the top "most used" projects according to `ohloh.net`, including only projects with significant amounts of Java code. We chose to select projects from `ohloh.net` because the site contains the most comprehensive list of open source projects of which we are aware. The 20 selected projects were ANT, AZUREUS, CHECKSTYLE, COMMONS COLLECTIONS, FREE-MIND, FINDBUGS, JETTY, JEDIT, JDT, JUNIT, ECLIPSE-CS, HIBERNATE, LOG4J, LUCENE, MAVEN, the SPRING FRAMEWORK, SQUIRREL-SQL, SUBCLIPSE, WEKA, and XERCES. In mining the full version histories of these 20 projects, we analyzed the full content of each version of each Java source file, a total of 548,982,841 lines.

Throughout this paper, we will focus our discussion on three of the 20 projects: JEDIT, ECLIPSE-CS, and SQUIRREL-SQL. We chose these specific projects because they are a fairly representative cross section of the 20 projects. JEDIT, a text editor for programming, began development in 2000 and is the largest and most mature project of the three. ECLIPSE-CS, which integrates the Checkstyle static analysis tool into the Eclipse Integrated Development Environment, began development in 2003 and is the smallest program of the three. SQUIRREL-SQL, a graphical user interface for exploring databases, began development in 2001.

Although we focus on these three projects throughout this paper, we also relate these results to the other 17 projects.

4.4 Methodology

To analyze the 20 projects in terms of our hypotheses, we chose an automated approach. Our approach involves several linked tools to perform the analysis on each project.

The first step in our analysis was to copy each project from a remote repository to a local machine. We did this to conserve network bandwidth and speed up the second step. We used `rsync` to copy projects stored in CVS and SVN, and `git-clone` for Git repositories.

The second step of our analysis was to check out every version of every file from the project's repository. Using a python script, we stored the different file revisions in an intermediate format.

Our third step comprised analyzing the generics usage in each revision. We performed this analysis using Eclipse's JDT to create an abstract syntax tree of each revision. From the abstract syntax tree, we extracted information relevant to generics, such as what kind of generic was used (type or method declaration, and parameterized type). We then populated a MySQL database with this information.

Finally, we analyzed the data in the database in a number of different ways, depending on what information we were trying to extract. We primarily used the `R` statistical package for analyzing and plotting data. Our data and tools are available in the PROMISE repositories (`http://promisedata.org`).

4.4.1 Identifying Generification

As part of our analysis, we identified instances in source code evolution where raw types were replaced by their generic counterparts (e.g. `List` to `List<String>`, hereafter referred to as corresponding types). We describe our approach in detail here and describe the results of using such analysis in subsection 7.1.

To identify changes in use of generics within a project, we use an approach similar to APFEL, by Zimmermann [20]. For each file in a project repository, we examined each pair of subsequent revisions of the file. For each method in each file (identified by name) we identify the number of uses of each raw and parameterized type in the method. If the count for a particular raw type decreases from one revision to the next and the count for the corresponding parameterized type increases by the same amount, we mark this as a generification.

More formally, let F denote the set of all files in a project repository and $R = \{1, 2, \ldots, n\}$ denote the set of all revisions in the repository. Thus, $f_r \in F \times R$ represents file f in revision r (or, put another way, immediately after revision r has been checked into the repository). Let M be the set of all method names in the source code in the repository and T_r be the set of all raw types and T_g be the set of all parameterized types in the source code. We now define two functions. Types_r takes a method m, file f, revision r, and raw type $t \in T_r$ and returns the number of uses of t in method m within revision r of file f.

$$\text{Types}_r : (M \times F \times R \times T_r) \to \mathbb{Z}$$

Similarly, Types_g provides the same functionality for a parameterized type $t \in T_g$.

$$\text{Types}_g : (M \times F \times R \times T_g) \to \mathbb{Z}$$

Finally, let $\text{Elide} : T_g \to T_r$ be a function that maps a parameterized type to its corresponding raw type. For example $\text{Elide}(\texttt{List<String>}) = \texttt{List}$. We record a generification of type $t_r \in T_r$ to type $t_g \in T_r$ in method $m \in M$ in revision

$r \in R$ of file $f \in F$ iff

$$\exists i > 0 : \text{Types}_r(m, f, r - 1, t_r) = \text{Types}_r(m, f, r, t_r) + i$$
$$\wedge\ \text{Types}_g(m, f, r - 1, t_g) = \text{Types}_g(m, f, r, t_g) - i$$
$$\wedge\ \text{Elide}(t_g) = t_r$$

We note that this approach is a heuristic and does not provide conclusive proof that a generification occurred. To assess this threat, we manually examined over 100 generifications identified by our algorithm and in all cases, the change represented a generification of a raw type.

One limitation of this approach is that we will miss "implicit" parameterized types. Consider the following two method signatures:

void printList(List<String> l)
List<String> getList()

Our analysis will identify both methods as using generics. However, if these two method calls are nested in a separate method:

a. printList (b. getList ())

then no parameterized type appears in the AST and we do not count it as a use of generics. Tackling this problem would require a static analysis beyond the bounds of an individual source file, heavily decreasing performance at the scale of our analysis (hundreds of millions LOC). We do not believe this impacts our results, as in our experience, few methods contain implicit parameterized types without type declarations.

5. DATA CHARACTERIZATION

5.1 Projects

Did projects adopt generics? Specifically, we equate the presence of parameterized types as adoption of generics and the presence of raw types as non-adoption. We counted the number of parameterizations and raw types at the latest point of development for all projects.

Figure 1 compares the number of raw types and parameterized types. Eight projects had more parameterized types than raw types while twelve projects used more raw types than parameterized. JEDIT and SQUIRREL-SQL made prominent use of generics.
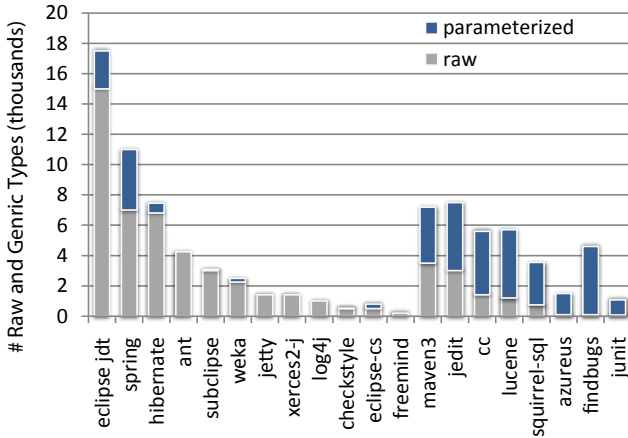

Figure 1: Parameterized and raw type counts in 20 projects.

Five projects ignored generics. Without interviewing the developers, we can only speculate on why. In section 6, we examine if the claims researchers made failed to hold in practice, and could contribute to lower adoption.

5.2 Developers

Did developers widely embrace generics? We examined commits with creation or modification of parameterized types, generic type declarations, or generic method declarations. In total, 532 developers made 598,855 commits to the projects. Of those developers, 75 developers created or modified generic declarations (14%) and 150 developers used parameterized types (28%). For these developers, the average number of commits involving generic declarations was 27 commits and 554 commits associated with parameterized types. Naturally, some developers commit more than others, which may give them more opportunity to use generics. Only 263 developers had more than 100 commits, averaging 2247 commits. Within this group of more frequent committers, 72 created or modified generic declarations (27%) and 112 used parameterized types (42%).

The data suggests only a select few of developers (perhaps with more authority or involvement) would create generic declarations followed by a modest embrace of generics by the most frequently committing developers. In later sections, we examine in more detail whether certain developers are choosing to ignore generics in favor of raw types (Section 7.1) and whether there is a concerted effort to migrate those raw types to use generic types instead (Section 7.2).

5.3 Features Breakdown

5.3.1 Common Parameterized Types

We classified parameterized types as either user-defined or from the standard Java Collections (`java.util`) based on name signatures. We found that on the whole, use of Collections types accounts for over 90% of parameterized types across all of the codebases that we examined. In every project, `List` objects were the most common, followed by `Map` objects. Table 1 illustrates this finding by showing use of parameterized types in the SQUIRREL-SQL project.

Type	Parameterizations
List<String>	1066
ArrayList<String>	682
HashMap<String,String>	554
List<ObjectTreeNode>	376
List<ITableInfo>	322
Class<?>	314
List<TableColumnInfo>	304
Vector<String>	234
List<ArtifactStatus>	196
Collection<String[]>	166
List<Object[]>	132
Iterator<String>	124
ArrayList<MappedClassInfo>	114
Set<String>	102

Table 1: Number of parameterizations of different generic types in SQUIRREL-SQL

5.3.2 Common Arguments

We also investigated which type arguments were used most frequently. Again, there was a very clear dominant usage pattern. `Strings` were by far the most common arguments. Table 1 shows the number of parameterized types of each kind of type argument in SQUIRREL-SQL for the most commonly used types. In fact, it appears that `Lists` and `Maps` of `Strings` account for approximately one quarter of parameterized types. We observed similar patterns in all projects with generics, with `Lists` of `Strings` always topping the list at almost twice the usage of the next commonly used parameterized type.

5.3.3 Generic Types versus Methods

We compared the number of user-defined generic types and methods. In total, 411 generic methods and 1127 generic types existed across all projects during the time of study. Out of the 15 projects that used generics, 6 had fewer than 10 generic types, and 3 projects had more than 100. This trend was not necessarily a function of size; for example, FINDBUGS made extensive use of generic types (88) in comparison to JEDIT (33) even though FINDBUGS is vastly smaller.

In every project there were more generic classes than generic methods, an average of about a 3-to-1 ratio. We were surprised by this; a large selling point of generics was the ability to create generic operations over data. Rather it seems, generics were used as placeholders and not as abstractions.

The number of generic declarations lagged the number of parameterizations, a tendency followed by most of the projects that we studied. Exceptions include FINDBUGS, which began using declarations and parameterizations at about the same time, and ANT and SUBCLIPSE, which never used any declarations. This lag suggests that adoption grows in stages as developers become more comfortable with the new feature. We examine adoption lag in section 7.3.

5.3.4 Unique parameterizations

For generics to be advantageous, each type declaration must be parameterized by multiple types, otherwise a simple non-generic solution would suffice. But, for example, a generic type may be parameterized many times throughout the code but only have one unique parameter (*e.g.,* `String`). In practice, how many unique parameterizations are made of type declarations? Is the number small or are generics preventing thousands of clones from being created?

From our data, we counted user-defined type declarations and their parameterizations. In total, 334 user-defined generic type declarations were instantiated. Of those, 33% had a single parameterization. The remaining 67% ranged from 2 to 39 parameterizations (mean=4.5). Overall, the distribution was very positively skewed such that 80% of generic classes had fewer than 5 parameterizations. Does this support the cost savings envisioned by the language designers? We investigate further in Section 6.2.

5.3.5 Advanced Parameterizations

We examined several advanced uses of parameterization, including **wildcard types**, such as `List<?>`, where the type argument matches any type; bounded types, such as `List<? extends Integer>`, where the argument matches a certain set of types; nesting, such as `List<List<String> >`; and mutiple type arguments such as `Map<String,Double>`.

As a percentage of all parameterized types, each advanced use made up the following percentages: nesting (<1%), bounded types (3%), wildcards (10%), multiple type arguments (20%).

6. INVESTIGATING CLAIMS

6.1 Generics reduce casts

One primary argument for introducing generics is that they reduce the number of runtime exceptions because they reduce the need to cast (Hypothesis 1). Thus, it is reasonable to expect that the addition of generics will reduce casts.

To test Hypothesis 1, we examined our data to determine if an increase in generics leads to a decrease in casts. Figure 2 plots the number of casts against the number of parameterized types for three projects. The x-axis represents time and the y-axis is the ratio of program elements (parameterized types or casts) to Halstead's program length; this essentially normalizes the number of program elements for program size. Red (top) lines represent the normalized number of casts over time. Blue (bottom) lines represent the normalized number of parameterized types (this ratio is multiplied by a factor of 10, because the number of parameterized types is small relative to the number of casts).

Overall, the graphs do not suggest a strong relationship between the use of casts and the use of parameterized types, for several reasons:

- The number of casts fluctuates significantly in the initial phase of all three projects (a trait shared by most of the 20 projects that we investigated), even before the introduction of generics. It would appear that some software development practice has a larger effect on casts than parameterized types.

- After the initial turmoil, all three projects show a gradual decline in the number of casts (a trait shared by about half of the projects that we investigated), *even before developers start to introduce generics.* This suggests that some other software development practice reduces the number of casts in a project over time.

However, the figures do suggest that the introduction of generics may, in some circumstances, reduce the number of casts. Specifically, notice that ECLIPSE-CS and SQUIRREL-SQL both exhibit sharp increases in the number of generics. ECLIPSE-CS, and to a lesser extent SQUIRREL-SQL, simultaneously decrease the number of casts in the program. This suggests that some projects will see a decrease in the number of casts when generics are introduced. However, this effect seems to be limited; a total of 6 out of 15 projects that used generics ever exhibit this sharp inverse relationship.

In addition to a visual inspection, we used a spearman rank correlation to examine the relationship between generics use and use of casts. We also employed Benjamini-Hochberg p-value correction to mitigate false discovery [2]. Of the statistically significant results ($p < 0.05$), we found that six of the projects showed a moderate inverse correlation (between -0.4 and -0.7) and only one project, SQUIRREL-SQL, showed a strong relationship (-0.84). Surprisingly, the SPRING FRAMEWORK actually showed a positive correlation (0.49), indicating that increased generics use coincided with more casts.

Overall, the data that we collected **does not support** Hypothesis 1.

The main limitation to this analysis is that we considered trends across all contributors. While this illustrates a more
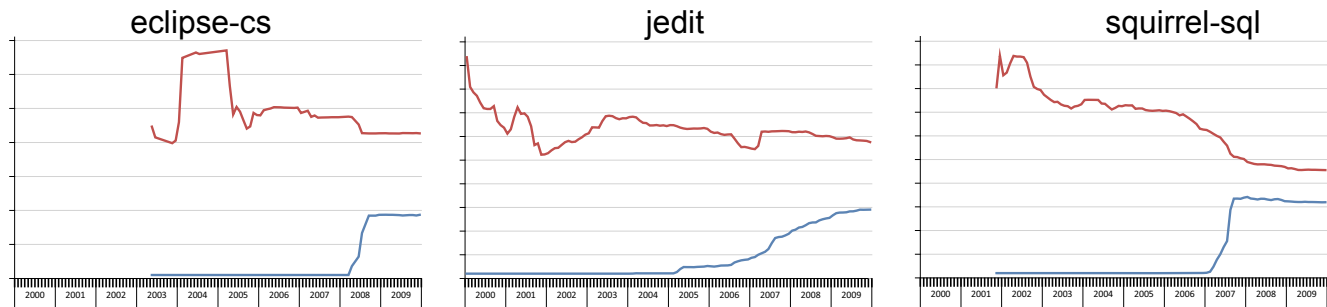
Figure 2: Number of casts (red, top line) versus number of parameterized types (blue, bottom line), normalized by program length.

project-wide trend, it may be that if we considered only the trends of generics and casts for developers who embraced generics, there would be a stronger relationship.

6.2 Generics prevent code duplication

Another claim regarding generics is that a generic type `Pair<S,T>` would prevent the need for countless clones of classes such as `StringIntPair` and `StringDoublePair` if a developer wanted to create a type-safe container. But in practice, how many clones would actually be needed? How many duplicated lines of code and bugs would be introduced from having to maintain these clones?

To test Hypothesis 2, we measure the number of unique parameterizations for all parameterized types to determine the number of clones. Further, we take our previous measures of unique parameterizations of just user-defined generics (shown in subsection 5.3.4), and use the lines of code and number of revisions in the source repository to estimate the impact of code duplication. Total lines of duplicated code are calculated by taking the number of unique parameters (P), lines of code (LOC) and applying this formula: $D = LOC * (P-1)$. This estimates the amount of additional code needed to provide implementations of non-generic code for each type parameter, P. Next, we take the total duplicated lines (D), the number of revisions (R), and an error constant (K) to estimate the potential faults in the code in this manner: $E = D * R * K$. This is a rough estimate that assumes a relatively uniform bug rate across lines of code.

From our data, we found a large number of clones would need to be created for a small number of types. We observed usage of 610 generic classes, but actually found about 50% of these types (306) only had exactly one type argument ever used throughout the project's history, suggesting that needless or premature generification of objects occurs fairly frequently. From the top ten generic classes having the most variation (all were Java collection classes), we found a total of 4087 variations. To accommodate all the variations of these ten classes, 4077 clones would need to be created, or about 407 clones per class. But the number of variations dropped drastically for the remaining 294 classes; 1707 clones would need to be created, or about 5.8 clones per class. Interestingly, we only found 6 variations for `Pair` across all projects.

Next, we analyzed the top 27 parameterized user-defined types to estimate the impact of code duplication. The generic classes had a total of 365 parameter variations. The mean code size of the types was 378 LOC and the types were changed a total of 775 times (mean 28). We estimate, as a result from these 27 generic types alone, an estimated 107,985 lines of duplicated code were prevented. With our error estimation, 400 errors would have been prevented based

on our metric and an error constant of 7.4/10000 (1/100 errors per commit, and 7.4/1000 errors per LOC [11]). On average, 28 bugs were prevented by a generic type.

Overall, this **supports Hypothesis 2**; however, the impact may not have been as extensive as expected. The benefit of preventing code duplication is largely confined to a few highly used types.

There are limitations to our results. We may over-estimate the code duplication if inheritance could have shared non-generic methods. We may under-estimate the number of unique parameterizations, as some generic types are intended for client use and were not used in the code we analyzed, for example the library COMMONS COLLECTIONS; there were 674 generic classes that were never parameterized. Further, we excluded 119 generic types from analysis that had only one unique parameter which themselves were other generic parameters. This might be common, for example, with a `GenericHashKey` that might be used by other generic types.

7. JAVA GENERIC ADOPTION

7.1 What happens to old code?

Since generics supposedly offer an elegant solution to a common problem, we investigated how pre-existing code is affected by projects' adoption of generics in an effort to answer Research Question 2. Is old code modified to use generics when a project decides to begin using generics? There are competing forces at play when considering whether to modify existing code to use generics. Assuming that new code uses generics extensively, modifying existing code to use generics can make such code stylistically consistent with new code. In addition, this avoids a mismatch in type signatures that define the interfaces between new and old code. In contrast, the argument against modifying old code to use generics is that it requires additional effort on code that already "works" and it is unlikely that such changes will be completely bug-free.

To address this question as presented in Research Question 2, we examined if and how old code is modified after generics adoption. Figure 3 depicts a gross comparison by showing the growth in raw types (solid red) and generic types (dashed blue) over time for the three projects of interest. Note that raw types are types used in the system for which a corresponding generic type exists, such as List. A drop in raw types that is coincident with an increase in parameterized types (e.g. in mid 2007 in SQUIRREL-SQL, which we manually verified by inspection as a large generification effort) indicate evidence of possible generification. Changes in types may not themselves be evidence of actual
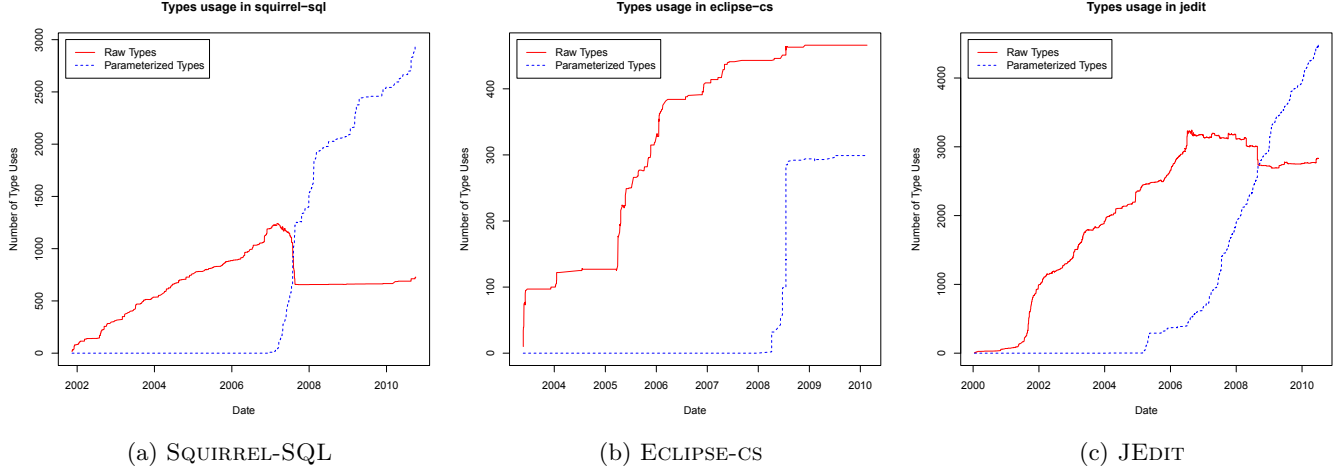
7

(a) SQUIRREL-SQL (b) ECLIPSE-CS (c) JEDIT

Figure 3: Migration efforts in switching old style collections was mostly limited in projects: old code remains. Solid lines indicate use of raw types (types such as `List` that provide an opportunity for generification) and dashed lines, generic types.

generification, however. We therefore determined generifications in a more principled way. Specifically, we identified raw types in the code as candidates for parameterization. We then examined what proportion of these candidates actually were removed and replaced by their generic counterparts by using the approach described in subsection 4.4.1.

In SQUIRREL-SQL, a total of 1411 raw types were introduced into the codebase over the life of the project (note that some were removed before others were added, so the maximum shown in Figure 3 is 1240). Of these, 574 (40.7%) were converted to use generics over a five month period starting when they were adopted in early 2007 (we identified these using the approach described in section 4.4.1). In contrast, JEDIT had 517 of a total 4360 introduced raw types converted to use generics (11.9%) and ECLIPSE-CS had only 30 of 497 converted (6%). Of the other projects studied, only COMMONS COLLECTIONS (28%) and LUCENE (33.4%) had more than 10% of their existing raw types generified. In aggregate, only 3 of the 15 projects that use generics converted more than 12% of their raw types and none of them converted more than half of their raw types use. We therefore conclude that although we do see a few large-scale migration efforts, **most projects do not show a large scale conversion of raw to parameterized types**.

7.2 Who buys-in?

Research Question 1 relates to *who* uses generics in the projects that adopt them. We expect that since most large projects depend on the principle of community consensus, the decision to use generics would be made as a group and would not be dominated by one developer.

To answer Research Question 1, we examined the introduction and removal of parameterized types by developers over time. Figure 4 shows the introduction (and removal) of parameterized types by contributor for the five most active contributors to each project. A solid line represents the number of raw types, which are candidates for generification, and a dashed line, parameterized types. Pairs of lines that are the same color denote the same contributor. A downward sloping solid line indicates that a contributor removed raw types. For instance, 4-a shows that in SQUIRREL-SQL, one contributor began introducing parameterized types in early 2007 while concurrently removing raw types.

The most common pattern that we observed across projects was one contributor introducing the majority of generics. This pattern is illustrated in SQUIRREL-SQL and ECLIPSE-CS (4-a and 4-b), and similar phenomena were observed in JDT, HIBERNATE, AZUREUS, LUCENE, WEKA, and COMMONS COLLECTIONS. We performed a Fisher's exact test [7] of introduction of raw and parameterized types comparing the top contributor with each of the other contributors in turn (using Benjamini-Hochberg p-value correction to mitigate false discovery [2]) to determine if any one contributor uses generics on average much more than the others. This test examines the *ratio* of raw types to parameterized types rather than the total volume, so that the difference of overall activity is controlled for. We found that in all cases, one contributor dominates all others in their use of parameterized types to a statistically significant degree.

JEDIT (4-c) represents a less common pattern in that all of the active contributors begin using generics at the same time (towards the end of 2006). This is more representative of the SPRING FRAMEWORK, JUNIT, and MAVEN. Interestingly, although our graph of JEDIT shows that most contributors began using parameterized types, a Fisher's exact test showed that one contributor (shown in yellow) still used parameterized types more often than raw types compared to all other contributors to a statistically significant degree.

Lastly, FINDBUGS (not shown) is an outlier as the two main contributors began using generics from the very beginning of recorded repository history and parameterized types were used almost exclusively where possible; we found almost no use of raw types in FINDBUGS at all.

Overall, the data and our analysis indicates that **generics are usually introduced by one or two contributors who "champion" their use and broad adoption by the project community is uncommon**.

In further work, we plan to investigate and contact these early adopters to identify *why* and *how* they began introducing generics as well as the obstacles (both technological and social) that they encountered.

7.3 How soon adopted?

We next turn to the question of how long it has taken software projects to adopt generics use and if there is a

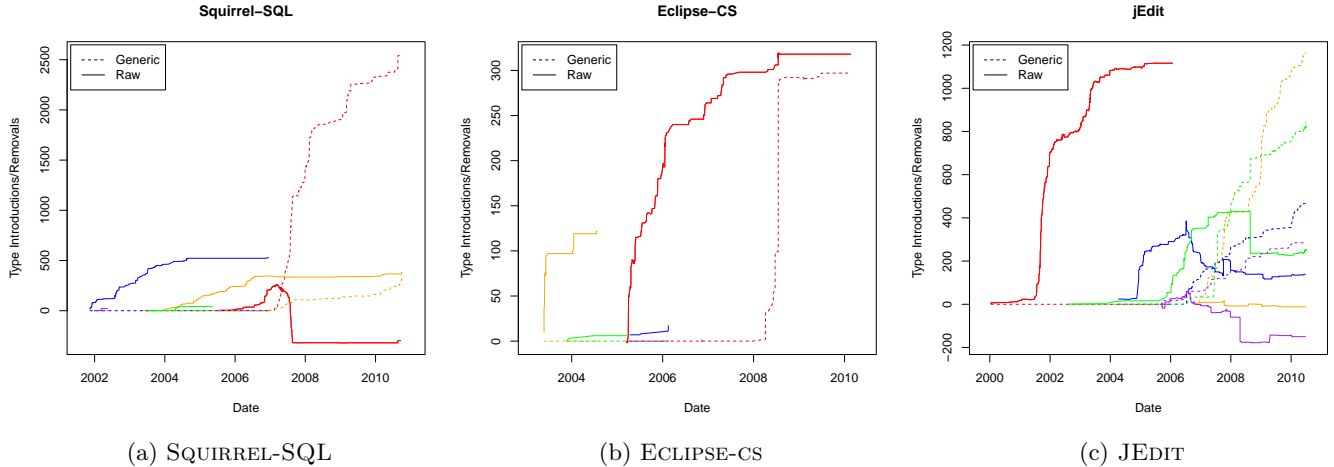(a) SQUIRREL-SQL   (b) ECLIPSE-CS   (c) JEDIT

Figure 4: Contributors' introduction and removal of type uses over time for the five most active contributors in each project. Solid lines indicate use of raw types (types such as `List` that provide an opportunity for generification) and dashed lines, parameterized types. Each color represents a different contributor.

relationship with tool support (including IDEs such as Eclipse and NetBeans) of generics. In order to relate time to key events that may affect adoption, we chose dates of generics feature introduction in Java and IDE support for generics.

To evaluate Research Question 3, we first had to determine which projects used which IDEs. We found evidence that IDEs were used for development for most of the projects that we studied. This evidence existed in the form of files created by IDEs (`.project` files in the case of Eclipse) or discussions on mailing lists. Eclipse was the most predominant IDE that we found evidence for, used by developers in AZUREUS, CHECKSTYLE, ECLIPSE-CS, FINDBUGS, JETTY, JUNIT, JDT, the SPRING FRAMEWORK, SQUIRREL-SQL, SUBCLIPSE, WEKA, and XERCES.

Although Java 1.5 with generics was released in September of 2004, Eclipse did not support generics until the public release of version 3.1 in late June, 2005. NetBeans supported generics at the same time that they were introduced, making a study of the effects of support for this IDE difficult if not impossible. We therefore examined each of the eight projects that use Eclipse as an IDE to determine if they adopted generics prior to the 3.1 release. Of these projects, CHECKSTYLE, JUNIT, JDT and FINDBUGS started using generics prior to generics support in Eclipse. The other four projects waited until after generics support appeared in Eclipse and did not switch until sometime in 2006 or later (SUBCLIPSE did not begin using generics until 2010). We examined the developer mailing lists at the time that generics support was added to Eclipse and also at the time that they began using generics and found no discussion of generics support in Eclipse as a factor in decision-making. Although these eight projects technically adopted generics after Eclipse support for them, the fact that adoption did not occur for at least six months after such support along with an absence of evidence on the developer mailing lists, leads us to believe that IDE support may not be critical.

The following quote from Jason King in a discussion of generics support in Eclipse provides one way to reconcile the perceived importance of tool support with our findings.[2]

Our team adopted Java 5.0 back in November 2004 and incrementally adopted the [Eclipse] 3.1 milestone builds as they came out throughout the first 6 months of this year. We found the product to be remarkably stable from an early stage, with few serious bugs.

As the entire team was learning the Java 5 features, we started manually coding in generics (and enums, varargs, annotations etc). A few times we complained that autocompletion and refactoring would help, but the absence didn't stop us. When a new [Eclipse] milestone came out our pairing sessions were really fun as we discovered new features appearing in the IDE.

Although tool support does not appear to be critical, we also looked at time of adoption to identify other possible factors affecting uptake time. Interestingly, we found no trend related to when generics were adopted. As examples of the observed variation, JEDIT started using them in 2004, SQUIRREL-SQL in 2006, ECLIPSE-CS in 2008, and SUBCLIPSE in 2010. FINDBUGS is again an anomaly as it used generics *before generics were officially released*! The only statement we *can* confidently make is that there was *not* strong adoption of generics immediately following their introduction into Java.

We also saw wide variation in the *rate* of generics adoption within the codebases. Figure 3 shows that SQUIRREL-SQL, ECLIPSE-CS, and JEDIT introduced generics into the code at a rapid rate once the decision to use them was made. In contrast, a number of projects, LUCENE, HIBERNATE, AZUREUS, CHECKSTYLE, and JUNIT show a lull in generics use for months or even years following first generics use.

Overall, the data that we collected to answer Research Question 3 indicate that **lack of IDE support for generics did not have an impact on its adoption**. This finding raises more questions than it answers. Deciding to use a new language feature is non-trivial and can have large consequences. If many projects adopted generics, but did so at vastly different times and rates, what factors affect the decision of when to begin using them? In the future, we plan to contact project developers, especially those that first began using generics, to identify these factors.

---

8. DISCUSSION AND FUTURE WORK

Overall, we were surprised by several of our findings about generics, which are at odds with our initial hypotheses. For instance, we were surprised that over half of the projects and developers we studied *did not* use generics; for those that did, use was consistently narrow. Empirically, we have found that generics are almost entirely used to either hold or traverse collections of objects in a type safe manner. Indeed, had the language designers of Java generics instead opted to introduce a single `StringList` class, then they would have succeeded in satisfying 25% of Java generic usage.

Although we found some merit to claims of reducing code duplication, we found the impact to be limited in scope. A surprisingly dubious argument is that casts are reduced with use of generics. In future studies, we would like to investigate in more detail the underlying reason.

This could well indicate that a language enhancement as large scale and sweeping as generics may have been more than what was really needed. Perhaps, language designers can take this to heart: in addition to the many difficulties inherent in adding generics to the Java language, designers had heated debates regarding the finer points of semantics. James Gosling, known as the father of the Java language, portrayed discussion of generics as a "food fight" and that "generics is the thing that caused Bill Joy and I to get as close to physical violence as we've ever gotten" [12].

While our results have painted a broad picture of how generics are used, different projects adopted generics at different times, and different people made use of generics in different ways. In the future we plan to better understand what are deciding factors or barriers for adopting new language features by contacting the developers to understand their thoughts and opinions of generics. We have measured use of generics by examining the frequency of their occurences within the source code, but there may be other measures of impact such as number of uses dynamically at run-time and we are investigating these measures. Further, we plan on manually inspecting less-frequently used aspects of generics to more qualitatively identify the value and impact of generics on the software.

9. CONCLUSION

We have explored how Java developers have used, and not used, Java generics over the past few years. We uncovered surprising generics usage trends, but also observed variation between projects and between developers. However, the results presented here illustrate only broad trends; future work will explain why these trends and variations exist.

While we expect that our retrospective results will, at this point, have little impact on Java generics, our results may help us adjust our expectations about the adoption of future language features. For example, based on our results, developers may not replace old code with new language features, so perhaps the introduction of a language feature alone is not enough to assure adoption. In future language-design wars, we hope that empirical data about how developers use language features may be an antidote to anecdotes.

10. REFERENCES

[1] H. Basit, D. Rajapakse, and S. Jarzabek. An empirical study on limits of clone unification using generics. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, pages 109–114, 2005.

[2] Y. Benjamini and Y. Hochberg. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995.

[3] J. Bloch. *Effective Java*. Prentice-Hall PTR, 2nd edition, 2008.

[4] G. Bracha. Lesson: Generics. Web. http://download.oracle.com/javase/tutorial/extra/generics/index.html.

[5] G. Bracha. Generics in the java programming language. Web, July 2005. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[6] A. Donovan, A. Kieżun, M. S. Tschantz, and M. D. Ernst. Converting java programs to use generic libraries. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 15–34, New York, NY, USA, 2004. ACM.

[7] S. Dowdy, S. Wearden, and D. Chilko. *Statistics for research*. John Wiley & Sons, third edition, 2004.

[8] N. Ducheneaut. Socialization in an Open Source Software Community: A Socio-Technical Analysis. *Computer Supported Cooperative Work (CSCW)*, 14(4):323–368, 2005.

[9] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. *Eurpoean Conference on Object Oriented Programming*, pages 71–96, 2005.

[10] R. Geiger, B. Fluri, H. Gall, and M. Pinzger. Relation of code clones and change couplings. *Fundamental Approaches to Software Engineering*, 3922:411–425, 2006.

[11] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[12] D. Intersimone. New additions to the Java language. Java One 2001 Keynote delivered by James Gosling. Web. http://edn.embarcadero.com/article/27440.

[13] S. Markstrum. Staking claims: A history of programming language design claims and evidence (a positional work-in-progress). In *Proceedings of the Workshop on the Evaluation and Usability of Programming Languages and Tools*, 2010.

[14] A. Mockus, R. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002.

[15] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software Quality Analysis by Code Clones in Industrial Legacy Software. In *Proceedings of the 8th International Symposium on Software Metrics*, page 87. IEEE Computer Society, 2002.

[16] M. Naftalin and P. Wadler. *Java generics and collections*. O'Reilly Media, Inc., 2006.

[17] S. O'Mahony and F. Ferraro. The emergence of governance in an open source community. *Academy of Management Journal*, 50(5):1079–1106, 2007.

[18] V. Pankratius, A. Adl-Tabatabai, and F. Otto. Does Transactional Memory Keep Its Promises?: Results from an Empirical Study. Technical Report 2009-12, Universität Karlsruhe, Fakultät für Informatik, 2009.

[19] D. Vandevoorde and N. Josuttis. *C++ templates: the Complete Guide*. Addison-Wesley Professional, 2003.

[20] T. Zimmermann. Fine-grained Processing of CVS Archives with APFEL. In *Proceedings of the 2006 OOPSLA*

Workshop on Eclipse Technology eXchange, New York, NY, USA, October 2006. ACM Press.