Empir Software Eng (2013) 18:1047–1089
DOI 10.1007/s10664-012-9236-6

Adoption and use of Java generics

Chris Parnin · Christian Bird · Emerson Murphy-Hill

Published online: 6 December 2012
© Springer Science+Business Media New York 2012

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 how generics have been adopted and used in practice. 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 40 popular open source Java programs, traversing more than 650 million lines of code in the process. We evaluate five hypotheses and research questions about how Java developers use generics. For example, our results suggest that generics sometimes reduce the number of type casts and that generics are usually adopted by a single champion in a project, rather than all committers. We also offer insights into why some features may be adopted sooner and others features may be held back.

Keywords Generics · Annotations · Java · Languages · Post-mortem analysis

Communicated by Arie van Deursen, Tao Xie, and Thomas Zimmermann

C. Parnin
College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA
e-mail: chris.parnin@gatech.edu

C. Bird
Microsoft Research, Redmond, WA 98052, USA
e-mail: cbird@microsoft.com

E. Murphy-Hill (✉)
Department of Computer Science,
North Carolina State University, Raleigh, NC 27695, USA
e-mail: emerson@csc.ncsu.edu

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 (Markstrum 2010). 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 as Java version 5 in 2004. 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 seven 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.

This paper extends our prior MSR 2011 paper (Parnin et al. 2011), where we made the following contributions:

- 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 (Sections 5–7); and
- We discuss the implications of the adoption and usage patterns of generics (Section 9).

In the prior paper, we examined our research questions and hypotheses from the perspective of *established projects*, projects which started before generics. This perspective was unique in that it allowed us to observe the impact of a new feature on an existing code base. In the present paper, we contrast our prior results with the adoption patterns of *recent projects*, projects which started after generics and may offer different perspectives. Second, we also wanted to compare the adoption of Java generics with an another feature, Java annotations, that were released in conjunction with generics in the Java 5 release. By examining annotations, an arguably less risky and simpler feature, we have the ability to tease apart some of the factors that influence adoption; for instance, was Java Virtual Machine compatibility the main barrier to adoption, or was it something else?

In this paper, we add the following new contributions:

- We explore 20 new open source projects that were initiated after the introduction of generics and
- We contrast our findings about generics with data on another language feature, Java annotations.

2 Language Feature Overview

In this section we briefly describe the motivation and use of Java generics and annotations. 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 and annotations may safely skip this section.

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 `Files`, 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** (Bracha 2012) (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** (Bracha 2012), 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** (Bracha 2005). 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** (Bracha 2005), 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 head(List<A> l){
  return l.get(0);
}
```

In this code, the programmer can pass to the `head` method a generic list containing any type.

2.3 Motivation for Annotations

Programmers sometimes want their software to give information to the tools that run over that software. For example, a program might want to tell a compiler that a certain method is deprecated and should no longer be called (Java Language Guide 2012) or a class might want to tell its environment that it represents a web service (The Advantages of the Java EE 5 Platform 2012). Prior to Java 5, such mechanisms to communicate with tools were *ad hoc*. For example, before Java 5, the `Deprecated` tag in a JavaDoc comment indicates whether a method is deprecated, while an external descriptor file indicates that a class is web service.

2.4 Programming with Annotations

With Java 5, the annotation language feature was introduced as a unified syntax for programs to issue directives to tools. To use an annotation, the programmer puts an @ symbol followed by an annotation name just before a program element (such as a class or method), and, if the annotation has values, sets those values in curly brackets. For instance, to tell the compiler that the `head` method is deprecated, the programmer can write the following:

```
@Deprecated
<A> A head(List<A> l){
```

When a program is compiled, the compiler warns the programmer about any code that references this method. If the programmer wants to mark a class as a web service, she can write the following:

```
@WebService
public class MyWebService{
```

The `@Deprecated` annotation is an example of an annotation recognized by the Java 5 compiler. Two other annotations are recognized by default by the compiler: the `@Override` annotation, used for indicating that a method overrides a method in a superclass, and the `@SuppressWarnings` annotation, used for telling the compiler not to generate certain warnings when compiling (The Java Tutorials 2012). The `@WebService` annotation is an example of an annotation defined in a specific API. Often these types of annotations are discovered and inspected via reflection and used for purposes such as automatically generating wrapper code or configuring framework properties. Users can define their own custom annotations as well, although a discussion of how this is done is beyond the scope of this paper.

3 Related Work

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

3.1 Claims Regarding Generics

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] 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 (2008) 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, Donovan et al. (2004) 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 and Josuttis (2003) 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 (Geiger et al. 2006) and perform maintenance more frequently (Monden et al. 2002).

Naftalin and Wadler (2006) 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.

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

- Typed data collections such as `FileList`, create non-standard 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 of Generics

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. (2005) 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. (2005) implemented refactoring tools that would replace raw references to standard library classes with parameterized 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 (Pankratius et al. 2009). They 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.

3.3 Empirical Studies of Annotations

In this paper we contrast the adoption of Java generics with adoption of Java annotations. While many researchers have introduced new types of annotations, such as for extended type checking (Flanagan et al. 2002) and pluggable types (Papi et al. 2008), little work has studied the use of annotations in existing programs. The most relevant empirical research that we know of is Shi and colleagues' study of how API documentation changes over time (Shi et al. 2011). Specifically, the authors looked at how Java API annotations are changed in five real-world libraries in order to understand how API documentation evolves. In contrast, the study presented in

this paper analyzes a wider variety of Java annotations in order to understand how language features are adopted.

Other research has investigated how annotation-like source code constructs are used. For example, Liebig and colleagues studied the use of C preprocessor directives to understand whether those directives align with the source code they accompany (Liebig et al. 2011). As another example, Storey and colleagues studied how developers tag their code with task markers (such as "TODO") to understand how developers manage tasks (Storey et al. 2008). In contrast to these studies, the current papers seeks to study the use of annotations as a means to understand language feature adoption.

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

One of the claims regarding generics (identified previously) is that they reduce the number of runtime exceptions (Bloch 2008). Ideally, we would like to know how many `ClassCastExceptions` a program threw before generics were introduced, then compare that to the number thrown after generics were introduced. If the claim is true, the number of thrown `ClassCastExceptions` should be reduced. To investigate the feasibility of this type of analysis, we manually searched the bug repositories of three large projects (JDT, the Spring Framework, and OpenSSO) for valid bug reports containing `ClassCastExceptions`. Overall, we found very few bug reports regarding `ClassCastExceptions`: in JDT, only about 10 `ClassCastException` bugs were reported per year; in the Spring Framework, only about 13 per year, and in OpenSSO, only about 5 per year. In smaller projects, the number of reported `ClassCastExceptions` is likely much smaller. We hypothesize that the problem is not so much that `ClassCastExceptions` occur infrequently, but that they are usually introduced and fixed before the software is released. Because of the low number of bug reports about `ClassCastExceptions`, we reasoned that this was not a feasible approach to perform a temporal, statistical analysis to investigate the claim about generics reducing runtime exceptions. We also rule out dynamic approaches where we would run each version of a program due to the state space explosion problem, which is compounded by the thousands of different versions of many open source projects.

However, Bloch, in his remarks about runtime exception, continues with a related claim that casts would also be reduced by the introduction of generics (Bloch 2008). Researchers consider casts to be a code smell (Van Emden and Moonen 2002),

indicating poor code structure and a catalyst for runtime exceptions. We reason that evidence of reducing casts also gives evidence of reducing probability of runtime exceptions by a non-zero amount. Thus, we investigate:

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

We also investigated Donavan's claim that without a mechanism such as generics, it would be necessary for programmers to introduce code duplication in order to achieve type safety. Donavan argued that developers would be forced to create data structures for every type of data they wanted to store. If we assume that Donavan's claim is valid, then we can measure the worse-case cost for achieving type-safety via the method proposed by Donavan. Specially, we can estimate the amount of duplication and bugs that would arise from having to maintain the duplicated type-safe version of classes. There are several reasons why this is an worse-case estimate: e.g., developers may find ways to factor out commonalities in non-type safe code. But, taken more generally, these measures provide a simple way of quantifying the value of generics by observing if types are instantiated with more than one parameter.

Hypothesis 2 Manually maintaining type-safe code would be costly due to maintaining a high number of clones.

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. Ducheneaut 2005; Mockus et al. 2002; O'Mahony and Ferraro 2007), the nature of adoption of new language features such as Java generics or annotations is not clear.

Our first research question investigates if there will be a concerted effort to convert old code to use the new generic language feature. 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? In other words:

Research Question 1 Will there be large-scale efforts to convert old code using raw types to use generics?

Our second research question centers around how project members embrace new language features such as Java generics and annotations. 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 a new feature 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 new

features rather than acceptance by individual members. We would also expect our research question to have consistent answers for both generics and annotations:

Research Question 2 Will project members broadly use new language features after introduction into the project?

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. Additionally, developers are often constrained by the platforms they are intended to deploy on. We expect that the choice to use new language features such as generics or annotations will in part depend on the tool support available and platform support for those features.

Research Question 3 What factors influence adoption of new language features?

4.3 Projects Studied

To test our hypotheses and evaluate our research questions, we automatically analyzed 40 open source software projects.

For the first 20, we analyzed the top "most used" projects according to ohloh.net, selecting 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 can be seen in Table 1.

Table 1 20 open source projects that were established before Java generics existed

Project name	Devs	Age	Start	End	LOC
Ant	38	10	1/13/2000	11/15/2010	85,736
Azureus	29	6	7/7/2003	4/01/2010	130,440
CheckStyle	5	6	6/22/2001	12/13/2007	174,611
Commons Collections	27	9	4/14/2001	10/22/2010	235,487
Eclipse-cs	6	7	5/21/2003	6/30/2010	592,214
Eclipse-JDT (JDT)	69	9	5/2/2001	11/19/2010	45,979
FindBugs	29	7	3/24/2003	10/25/2010	27,894
FreeMind	4	8	8/1/2000	7/17/2009	175,042
Hibernate	23	4	11/29/2001	2/27/2006	3,125,097
JEdit	94	10	1/16/2000	6/30/2010	52,031
Jetty	13	10	8/6/1998	5/15/2009	90,862
JUnit	6	8	12/23/2000	1/27/2009	154,984
Log4j	14	9	12/14/2000	8/18/2010	164,710
Lucene	35	8	9/18/2001	3/23/2010	71,168
Maven	29	6	1/3/2004	11/16/2010	417,803
the Spring Framework	27	3	6/17/2005	4/13/2009	292,379
Squirrel-SQL	17	8	11/13/2001	10/5/2010	81,889
Subclipse	16	7	6/20/2003	11/09/2010	39,532
Weka	25	8	4/20/1999	12/17/2007	35,419
Xerces	28	11	11/9/1999	11/14/2010	21,520

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.

For the final 20 projects, we decided to use a different sampling methodology for two reasons. First, after examining the first 20 projects, we realized that the type of sampled projects tended to be skewed toward developer tools. Second, some of the first 20 projects appeared not to use generics to be backward compatible with clients who used Java environments that are not generics-compliant. To address these two limitations of the first data set, we sampled projects using two criteria. First, with each of the 20 categories of projects listed on sourceforge.net, we analyzed one Java project that was tagged on Ohloh with that category name. The categories are mobile, internet, text editors, religion and philosophy, scientific and engineering, social sciences, other, formats and protocols, database, security, printing, terminals, office and business, system, education, games and entertainment, desktop environments, communications, and multimedia. Second, we chose projects whose first commit appeared well after 2004, and tried to exclude projects whose first commit appeared to be a repository migration. The 20 selected projects shown in Table 2.

In analyzing the history of these projects, we analyzed 104,069,124 lines of code.

Throughout this paper, we will focus our discussion on three of the 40 projects: JEdit, Squirrel-SQL, and MiGen. We chose these specific projects because they are a fairly representative cross section of the 40 projects. JEdit, a text editor for programming, began development in 2000 and is the most mature project of the three. Squirrel-SQL, a graphical user interface for exploring databases, began development in 2001. MiGen, an educational program for teachers of mathematics, is the least mature of the three projects, beginning in 2007.

Table 2 20 open source projects that were started after Java generics

Project name	Devs	Age	Start	End	LOC
BBSSH	1	1	1/19/2010	8/17/2011	42,127
Ehcache	24	5	3/26/2006	8/12/2011	166,808
encuestame	3	1	4/22/2009	2/18/2011	73,520
flowgame	5	1	5/09/2009	3/14/2011	10,284
Hummingbird	7	<1	3/3/2010	8/24/2011	16,178
ice4j	4	1	1/29/2010	8/8/2011	42,444
libgdx	18	1	3/6/2010	8/30/2011	166,887
Makagiga	1	5	2/25/2006	8/12/2011	253,187
MiGen	9	3	10/19/2007	8/31/2011	207,663
MOBAC	3	2	9/2/2008	8/13/2011	51,971
OpenSSO	97	4	11/1/2005	3/4/2010	241,062
PathVisio	15	5	1/30/2006	8/12/2011	99,273
Posterita	10	3	11/21/2005	3/31/2009	166,156
Red5	19	5	1/9/2006	8/29/2011	97,455
Religion Search	2	1	10/11/2010	12/17/2011	5,912
SCSReader	3	<1	7/3/2010	6/5/2011	3,858
SMSLib	6	3	1/31/2008	8/6/2011	28,448
VietOCR	1	3	7/27/2008	8/1/2011	10,912
XBUP	1	4	10/7/2006	8/11/2011	104,600
Zero Kelvin Desktop	3	1	12/29/2006	12/12/2008	13,506

Although we focus on these three projects throughout this paper, we also relate these results to the other 37 projects. To distinguish our two sets of projects, we refer to the first set of projects as the *established* projects and the second set of projects as the *recent* projects.

4.4 Methodology

To analyze the 40 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[2] (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 Section 7.1.

To identify changes in use of generics within a project, we use an approach similar to APFEL, by Zimmermann (2006). 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.

In an effort to present a precise description of our data collection, we present a formal definition. This description can be safely passed over by the uninterested

[2]Due to potential changes as the paper evolves, the complete data set will be on the PROMISE site by the final version of the paper and the correct URL to that data set will appear in that version of the paper.

reader and is not required to understand our results. 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.

$$\text{Types}_\text{r} : \left(M \times F \times R \times T_r\right) \to \mathbb{Z}$$

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

$$\text{Types}_\text{g} : \left(M \times F \times R \times T_g\right) \to \mathbb{Z}$$

Finally, let $\text{Elide} : T_g \to T_r$ be a function that maps a parameterized type to its corresponding raw type. For example Elide(`List<String>`) = `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

$$\begin{aligned} \exists i > 0 : \text{Types}_\text{r}(m, f, r-1, t_r) &= \text{Types}_\text{r}(m, f, r, t_r) + i \\ \wedge\, \text{Types}_\text{g}(m, f, r-1, t_g) &= \text{Types}_\text{g}(m, f, r, t_g) - i \\ \wedge\, \text{Elide}(t_g) &= t_r \end{aligned}$$

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

To give insight into our collected data, we characterize several facets about our data. Specifically, we break down the use of generics and annotations by established and recent projects, developers, parameterization behavior, and advanced features usage such as wildcards. Finally, we relate some observations that arose from our examination of the data.

5.1 Projects

Did projects adopt generics or annotations? Specifically, we examined the latest snapshot of each project in our data and then noted the number of instances of parameterized types, raw types, and annotations. For generics, we equate the presence of parameterized types as adoption of generics and the presence of raw types as non-adoption. For annotations, we counted the number of annotations in the project. Note, these measures only provide a very broad view of adoption.

Established Projects Figure 1 compares the number of raw types, parameterized types, and annotations in the established projects. 13 projects out of 20 made more use of raw types than generics, with 4 of those not using generics or annotations at all. JEDIT and SQUIRREL-SQL made prominent use of generics, whereas the SPRING FRAMEWORK and FINDBUGS made prominent use of annotations.

Recent Projects Figure 2 compares the number of raw types, parameterized types, and annotations in the recent projects. A different story emerges. Only 2 out of 20 projects had more raw types than generics. All projects used generics and all but one used annotations. There were 4 projects that did not have any raw types: FLOWGAME, ICE4J, RELIGION SEARCH, and SCSREADER.

While it is unsurprising that established projects continued to use raw types, we were surprised that raw types are still used in some recent projects. To get an idea

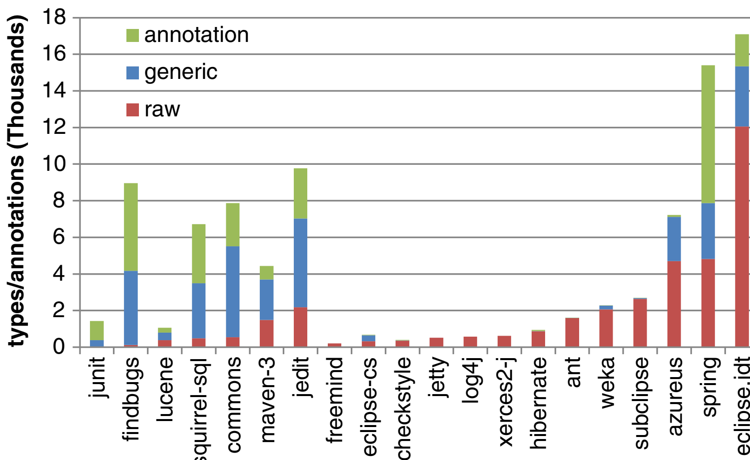


Fig. 1 Annotation, parameterized type, and raw type counts in 20 established projects

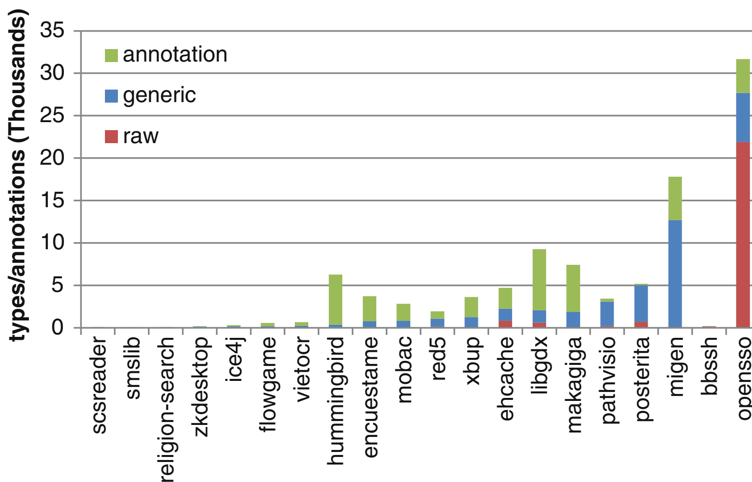


Fig. 2 Annotation, parameterized type, and raw type counts in 20 recent projects

why, we manually inspected a few raw types from the EHCACHE and MIGEN projects. In a few cases in EHCACHE, use of raw types made sense, such as when a generic type parameter made no difference in the program. For instance, we observed that in a custom implementation of a dictionary, two dictionary entries were compared for equality; in this case, the type of those entries made no difference, since equality is defined for all `Objects`. In these cases, developers could have used the wildcard type with generics, but for some reason, chose not to do so. In most cases, we could discern no particular reason for usage of raw types over generics in EHCACHE. For instance, in one class we observed the fully generic code:

```
List<Thread> requestThreads = new ArrayList<Thread>();
```

But then a few lines later, we observed generics mixed with raw types:

```
List<ThreadInformation> threads = new ArrayList();
```

In MIGEN, the few raw types that did exist appeared to be either in test code or scrupulously commented. In one inline comment, a developer noted that he did not generify a raw type because he did not have time; in another, a developer noted that he tried to generify a collection but the generic version caused unexpected runtime behavior.

Overall, without systematic inspection and interviewing the developers, we can only speculate on why some projects adopted generics and annotations and other did not. We plan on conducting such inspection and interviews as part of future work.

5.2 Developers

Did developers widely embrace generics? How did this compare with annotations? We examined commits with creation or modification of parameterized types, generic type declarations, generic method declarations, or annotations.

Established Projects In the established projects, 538 developers made 678,551 commits. Of those developers, 71 made generic declarations (13 %), 128 specified annotations (24 %), and 141 used parameterized types (26 %). Naturally, some developers commit more than others, which may give them more opportunity to use generics. Only 272 developers had more than 100 commits, averaging 2467 commits. Within this group of more frequent committers, 66 used generic declarations (24 %), 99 used annotations (36 %), and 105 used parameterized types (38 %).

Recent Projects In the recent projects, 232 developers made 197,744 commits. Of those developers, 47 used generic declarations (20 %), 138 used annotations (59 %), and 142 used parameterized types (61 %). Of the 102 more frequent committers in the recent projects, with an average 1906 commits, 43 used generic declarations (42 %), 83 used annotations (81 %), and 87 used parameterized types (85 %).

The data suggests there were several forces shaping use of new features in Java by developers. In both established and recent projects, a small minority of developers (perhaps with more authority or involvement) used generic declarations. In most projects, a single member of a project (perhaps having an architect role) clearly introduces a disproportionate amount of the generic declarations (see, for example, Fig. 7). In established projects, developers demonstrated a modest use of generics and annotations. Potentially, inexperience with the new features, or difficulty in migrating existing code to fit in with the new features hampered adoption. In more recent projects, these factors may have been ameliorated, as a larger percentage of developers have started to use generics and annotations in their code.

In general, we observed that developers generally adopt usage of both features, although there were a handful of developers that only adopted use of either annotations or generics exclusively.

5.3 Features Breakdown

We characterize how different aspects of a feature were used to identify any differences between established and recent projects and between usage of aspects of those features. In both cases, these differences give insight into adoption factors, such as the difficulty in learning aspects of a new feature and whether those differences persist over time. We focus mostly on generics, simply because there are many more aspects of generics to investigate in comparison with annotations.

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 about 70 % of parameterized types across all of the codebases that we examined. The most popular parameterized types across all projects were `Lists`, followed by `Maps`. Table 3 illustrates this finding by showing use of the top 14 parameterized types in the Squirrel-SQL project.

In comparison, Table 4 illustrates how annotations were used in Squirrel-SQL, showing a similar usage distribution to generics. Annotations from the standard Java library, such as *Override* and *Before*, are the only annotations used by the majority of the 40 projects analyzed. Otherwise, in addition to unit testing, annotations were used for a variety of domain- and project-specific cases.

Table 3 Number of parameterizations of several generic types in SQUIRREL-SQL

Type	Parameterizations
List<String>	351
ArrayList<String>	221
HashMap<String,String>	157
List<ITableInfo>	96
Class<?>	91
Collection<String[]>	77
List<ArtifactStatus>	61
Vector<String>	58
List<ObjectTreeNode>	55
List<TableColumnInfo>	55
Iterator<String>	40
List<Object[]>	33
ArrayList<MappedClassInfo>	28

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 3 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 in SQUIRREL-SQL. We observed similar patterns in other projects with generics, with `Collections` of `Strings` being the predominant parameterized type in half of projects studied. This trend tended to be stronger in the established projects, which predominantly used `String` parameters in 78 % of projects with generics, compared to recent projects in only 22 %. The second most popular parameter was `?` as an argument to the `Class` parameterized type, the most popular parameterized type in 14 % projects.

Overall, the most common usage of generics was to parameterize a collection of strings.

5.3.3 Generic Types versus Methods

We compared the number of user-defined generic types and methods across the established and recent projects.

Table 4 Number of uses of annotations in SQUIRREL-SQL

Annotation	Use count
Override	1935
Test	636
Before	274
SuppressWarnings	196
After	158
Ignore	16
GUITest	4
Deprecated	3
TestExecutionListeners	2
RunWith	2
ContextConfiguration	2

Established Projects In the established projects, 979 generic methods and 1684 generic types existed during the lifetime of the projects. Out of the projects that used generics, 4 projects had fewer than 10 generic types, and 4 had more than 100 generic types. This trend was not necessarily a function of size; for example, FindBugs made extensive use of generic types (116) in comparison to JEdit (39) even though FindBugs is roughly half the size of JEdit. Figure 3 shows box plots depicting the number of type and method declarations across all projects. In all but 4 established projects there were more generic classes than generic methods, an almost 2-to-1 ratio.

Recent Projects In the recent projects, 666 generic methods and 1234 generic types existed during the lifetime of the projects. Seven projects had fewer than 10 generic types, and 2 had more than 100 generic types. Only 3 projects had more generic methods than generic types, again matching the near 2-to-1 ratio also seen in the established projects. Overall, there were little differences between the established and recent projects.

A final observation we found was that introduction of generic types lagged behind the introduction of parameterized types, a tendency followed by most of the established projects that we studied. Exceptions include an early adoptor of generics, FindBugs, which began using generic types and parameterized types at about the same time, and Ant and Subclipse, which never used *any* generic types. However, we did not observe this trend as strongly in recent projects. This lag suggests that **adoption may grow in stages as developers become more comfortable with the new feature**.

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. Figure 4 shows box plots depicting the number of parameterizations of each user-defined type.

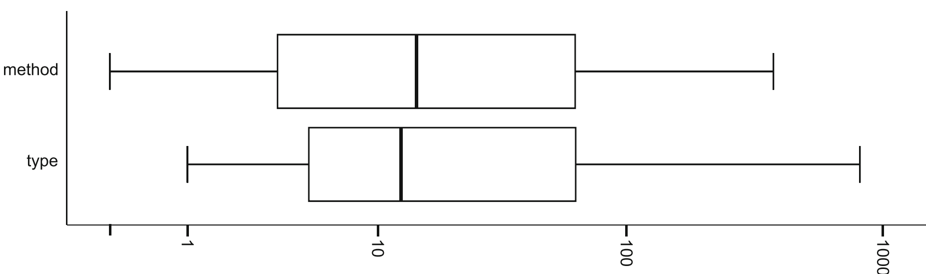


Fig. 3 *Box plots* displaying the number of method and type declarations in the projects under investigation

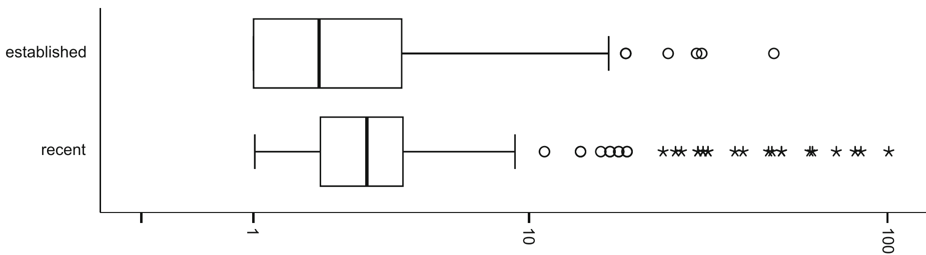


Fig. 4 *Box plots* displaying the number of parameterizations of each user-defined type in the established and recent projects

Established Projects In our established projects, 330 user-defined generic type declarations were instantiated in total 1123 times. Of those, 38 % had a single parameterization. The remaining 62 % ranged from 2 to 49 parameterizations (mean = 4.8). The distribution was very positively skewed such that 80 % of generic classes had fewer than 5 parameterizations.

Recent Projects In our recent projects, 332 user-defined generic type declarations were instantiated in total 2027 times. Of those, 23 % had a single parameterization. The remaining 77 % ranged from 2 to 100 parameterizations (mean = 7.5). Still, 76 % of generic classes had fewer than 5 parameterizations.

Overall, the lower portion of the distribution for both the established and recent projects were similar, differing on the tail-end in magnitude. This suggests that the **cost savings envisioned by the language designers may not have been fully realized in practice**.

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 multiple type arguments such as `Map<String,Double>`.

Established Projects As a percentage of all parameterized types for the established projects, each advanced use made up the following percentages: nesting (1 %), bounded types (4 %), wildcards (11 %), and multiple type arguments (22 %).

Recent Projects The break down was similar for the recent projects, as a percentage of all parameterized types each advanced use made up the following percentages: nesting (1 %), bounded types (2 %), wildcards (15 %), and multiple type arguments (14 %).

The consistent levels of usage between established and recent projects suggests that **there was an inherent difficulty or limited applicability in the more advanced features of generics, limiting their adoption**.

6 Investigating Claims

In this section, we examine Hypothesis 1 and Hypothesis 2. Here we do not specifically compare results for established projects against those for recent projects, as we did not find any substantial differences between the two project sets.

6.1 Generics Reduce Casts

An 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. However, comparing just the raw number of generics against the raw number of casts could be misleading, because an increase in generics may not actually cause a decrease in casts whenever new code containing parameterized types is added. To control for this, we calculated the density of program elements (parameterized types or casts) by dividing the number of program elements by Halstead's program length (Halstead 1977). Halstead's program length is the sum of the total number of operators (such as method calls) and the total number of operands (such as a variable). We used Halstead's program length here because it measures program size, but also disregards code formatting, whitespace and comments, making it preferrable to a simple lines-of-code metric. Thus, Halstead's program length allows us to more fairly compare projects that use different conventions for formatting, whitespace, and comments. This is important because, for example, Azureus has about half has many comments per line of code as Weka, according to ohloh.net.

Figure 5 plots the cast and parameterized type density for three projects. The x-axis represents time and the y-axis is the density of program elements. The number on the y-axis represents the number of program elements per unit program length. Red (top) lines represent the density of casts over time. Blue (bottom) lines represent the density of parameterized types over time. Because the density of parameterized types is small relative to that of casts, to improve the readability of the figure, the blue line is scaled by 10. Similar time series graphs are shown in the Appendix for all projects.

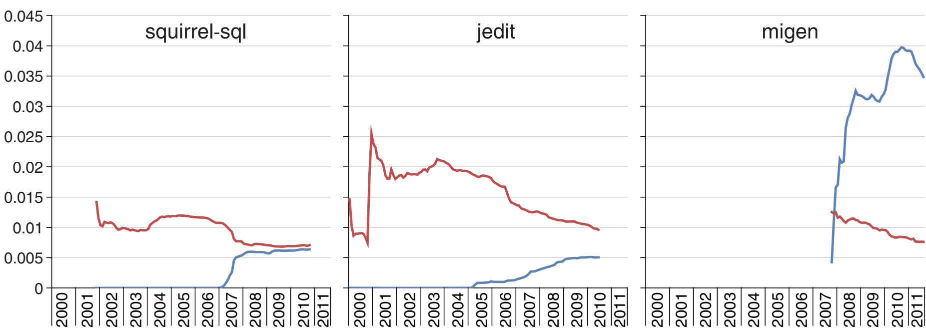


Fig. 5 Casts (*red*, *top line*) and parameterized type (*blue*, *bottom line*) density. Parameterized type density is scaled by a factor of ten to aid visual comparison

Overall, the graphs do suggest a relationship between the use of casts and the use of parameterized types. In SQUIRREL-SQL, an increase in generics in 2007 corresponds to a decrease in casts. The same is true about JEDIT from 2005 onward and over the lifetime of MIGEN. Ten other projects also distinctively showed this trend (ICE4J, ECLIPSE-CS, FLOWGAME, FINDBUGS, JUNIT, LUCENE, MAVEN, MOBAC, the SPRING FRAMEWORK, and PATHVISIO). Interestingly, a few projects showed the opposite trend (RELIGION SEARCH, LIBGDX, HUMMINGBIRD, COMMONS COLLECTIONS), where increases in generics tended to correspond to increases in casts. We speculate that this opposite trend may be due to changes that require use of generic types and APIs that require casting. For instance, using Java's reflection API, the object returned from `Class.forName(...)` will likely need to be cast.

In addition to a visual inspection, we used Spearman's rank correlation to examine the relationship between generics density and cast density over time. We also employed Benjamini–Hochberg p-value correction to mitigate false discovery (Benjamini and Hochberg 1995). Only RELIGION SEARCH did not show a statistically significant correlation ($p > .05$). Of the remaining 35 projects that used generics, we found that: 6 projects showed a strong inverse correlation (above −0.84); 9 showed a moderate inverse correlation (between −0.4 and −0.8); and 8 showed a weak inverse correlation (between 0 and −0.4). However, 10 projects showed a weak positive correlation (between 0 and −0.4) while, surprisingly, MAKAGIGA (0.69) and ENCUESTAME (0.65) showed strong positive correlations, indicating that increased generics use coincided with more casts. Again, this positive correlation may be due to changes that require use of generic types and APIs that require casting.

On the whole, the data that we collected **supports** Hypothesis 1.

One limitation to this analysis is that we considered trends across all contributors. While this illustrates a more 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.

Another limitation is that our density function used in the cast analysis sometimes may not accurately measure the effect of parameterized types on casts. For example, if a program contains generics, and then a large class is deleted that contains many casts and no generics, the density of generics in the program goes up while the density of casts goes down. Our analysis would mis-interpret this change as the addition of generics causing the removal of casts. Further study with more sophisticated metrics are needed to mitigate this threat.

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 measured 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 Section 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 40 projects, we found a large number of clones would need to be created for a small number of types. We observed parameterization of 1152 types, but actually found about 46 % of these types (532) 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 parameterizations (all were Java collection classes), we found a total of 8686 different parameterizations. To accommodate all the parameterizations of these ten classes, 8676 clones would need to be created, or about 868 clones per class. But the number of parameterizations dropped drastically for the remaining 1142 classes; 5275 clones would need to be created, or about 4.6 clones per class. Interestingly, we only found 13 parameterizations of `Pair` types across all projects. We speculate that a generic version of `Pair` is less useful than we initially expected.

Next, we analyzed the user-defined generic class from each project that had the most parameterizations, for the purpose of estimating the impact of code duplication. In total, we analyzed 12 user-defined generic classes from the established projects and 12 from the recent projects. The generic classes had a total of 347 parameterizations. The mean code size of the classes was 176 lines of code and the classes were changed a total of 244 times (mean 10). We estimate, as a result from these 24 generic classes alone, an estimated 109,816 lines of duplicated code were prevented. With our error estimation, 195 errors would have been prevented based on our metric and an error constant of 7.4/100000 (1/100 errors per commit, and 7.4/1000 errors per LOC (Humphrey 1995)). However, the number of errors prevented varied significantly between generic classes; of the 24 total generic classes, we estimate that 16 of them prevented no bugs at all.

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 classes.

Using a Wilcoxon signed-ranks test, we observed that there were no significant differences between the set of 20 established projects and the set of 20 recent projects. More specifically, there were no significant differences in terms of either group's 12 user-defined generic types in any of the following metrics: lines of code, duplication prevented, or errors prevented. This suggests that projects that "grew up with generics" did not benefit from generics' duplication prevention any more than established projects.

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. Finally, we did not exclude generics that were introduced for testing purposes, such as in JDT, where some generics are used to test Eclipse's Java language tools. As a consequence, projects that used generics for testing may not be representative of the average Java project.

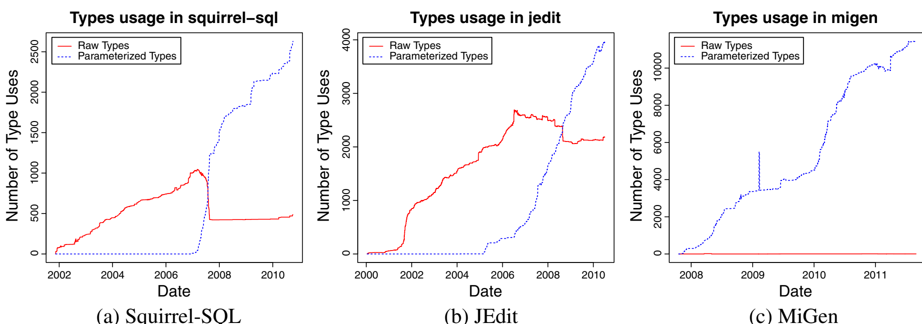
7 Factors for Adoption

Risk, legacy code, backward compatibility, developer politics, feature complexity, and learning; these are several factors that may influence adoption. By comparing differences in adoption by established and recent projects of generics and annotations we attempt to tease apart some of these factors.

7.1 Do Developers Change Old Code to Use New Features?

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 1. Conversely, for this research question, we did not examine annotations, as there was no corresponding old feature to "upgrade". 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 1, we examined if and how old code is modified after generics adoption. Figure 6 depicts a gross comparison by showing the growth in raw types (solid red) and generic types (dashed



(a) Squirrel-SQL (b) JEdit (c) MiGen

Fig. 6 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

blue) over time for the three projects of interest (see Appendix for graphs of all projects). 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 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 Section 4.4.1.

Established Projects Consider 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 Fig. 6 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 of 4360 introduced raw types converted to use generics (11.9 %). 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**.

Recent Projects For recent projects, we see even fewer and smaller migration efforts. In the Red5 project, 134 of the 416 total raw types that were added were eventually converted to parameterized types (yielding a final total of 1082 parameterized types). No other recent projects had more than 100 raw types converted to parameterized types, and seven projects had no migrations at all.

The reasons behind the lack of migration in the established and recent projects may actually be different. For projects that had a substantial code base when generics were added to the language, raw types were already heavily used and thus developers decided not to modify that code, taking an “if it’s not broken, then don’t fix it” mentality. In contrast, most projects that started after 2005 used generics from the start and did not use raw types extensively to begin with (notables exceptions were Ehcache and BBSSH). In these projects, we did not see migrations because there was not a large set of raw types that *could* be converted to use generics.

7.2 Who Buys-In?

Research Question 2 relates to *who* uses a new feature in the projects that adopt them. We expect that since most large projects depend on the principle of community consensus, the decision to use a new feature would be made as a group and would not be dominated by one developer. We separately analyzed developer’s use of generics and annotations. We also looked for any differences between established and recent

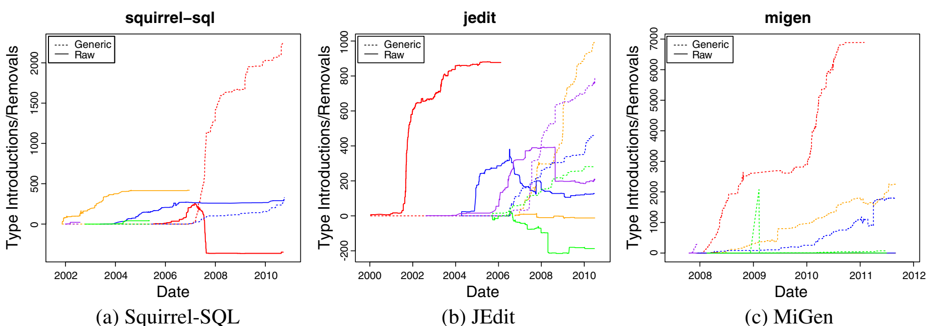
projects, where the *newness* of a feature may affect the dynamics of how community consensus occurs.

To answer Research Question 2, we first examined the introduction and removal of a feature by developers over time. We performed a Fisher's exact test (Dowdy et al. 2004) 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, Benjamini and Hochberg 1995) to determine if any one contributor uses a feature 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.

To illustrate these results, we make use of several graphs detailing different author's usage of a feature in a project. Figure 7 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, Fig. 7a shows that in SQUIRREL-SQL, one contributor began introducing parameterized types in early 2007 while concurrently removing raw types. The Appendix contains similar graphs of all projects.

Contributors' Use of Generics The most common pattern that we observed across projects was one contributor introducing the majority of generics. This pattern is illustrated in SQUIRREL-SQL (Fig. 7a) and similar phenomena were observed in ECLIPSE-CS, JDT, HIBERNATE, AZUREUS, LUCENE, WEKA, and COMMONS COLLECTIONS. In established projects, one contributor dominates all others in their use of parameterized types to a statistically significant degree ($\alpha = .05$).

In recent projects, we hypothesized that there may be different phenomena at work since there was no pre-existing non-generic code base that would make the decision to use generics a debated topic. Therefore, we expected broad community usage of generics. However, even in these newer projects, there was still a clear



(a) Squirrel-SQL (b) JEdit (c) MiGen

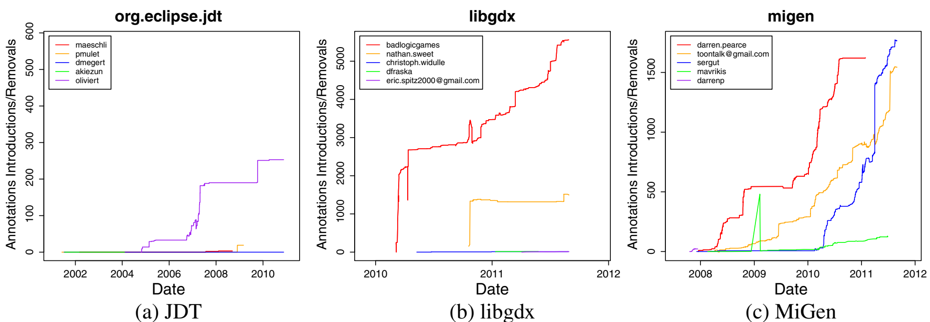
Fig. 7 Contributors' introduction and removal of type uses over time for the five most active contributors in each project. *Solid lines* indicate use of raw types (types such as `List` that provide an opportunity for generification) and *dashed lines*, parameterized types. Each *color* represents a different contributor

champion that accounted for most generics use in all but two projects (the contrary projects were PathVisio and SCSReader).

There were some outliers. JEdit (Fig. 7b) represents a less common pattern in that all of the active contributors began using generics at the same time (towards the end of 2006). This is more representative of the Spring Framework, JUnit, and Maven. 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.

Contributors Use of Annotations As a contrast to our generic buy-in results, we also examined individual contributors' adoption of annotations. Consistent with our results of analysis individuals' adoption of generics, we found that the majority of the projects that used annotations had a clear "champion" that used them more than the rest of the contributors to a statistically significant degree. Figure 8 shows the adoption graphs for the most active contributors in three projects. The graphs for the JDT and libgdx projects are representative of the vast majority of projects, as there is an obvious contributor that accounts for most annotations. The graph for MiGen is uncharacteristic, as there were a number of contributors that all actively added annotations to the codebase at roughly the same rate and time interval. Annotation graphs for all projects are shown in the Appendix.

The reader may notice that each of these graphs shows occurrences of steep increases over short time periods (e.g., the user sergut in MiGen in the early part of 2011). Interestingly, we also observed abrupt introductions of hundreds and sometimes even thousands of annotations in very short time periods. For instance, one contributor in the Lucene project added 2,182 annotations (across multiple files) in just one commit and one contributor added 16,019 to Hummingbird in just two days! We speculate that this level of activity may be indicative of use of an automatic technique for adding annotations. While not quite as extreme, we observed "bursts" of annotation introduction (usually on the order of hundreds in a short time period) in all projects that actually used annotations (36 out of 40) except for Makagiga,



(a) JDT (b) libgdx (c) MiGen

Fig. 8 Contributors' introduction and removal of annotations over time for the most active contributors in each project. Each *color* represents a different contributor

which showed a fairly constant monotonic increase, and WEKA and ANT, which did not use annotations extensively.

Overall, the data and our analysis indicates that **features 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 new features as well as the obstacles (both technological and social) that they encountered.

7.3 What Factors Affect Adoption?

Is backward compability the dominating concern, or do other factors such as risk, learning, or tool support play a role as well? In legacy codebases, are less risky features adopted earlier than more risky features? Do these trends disappear in more recent projects?

To evaluate Research Question 3, we focused on the factors of compatibility and IDE support. We separately analyzed established and recent projects where applicable to identify consistent trends.

7.3.1 Compatibility or Other Factors

To evaluate the factor of compatibility, we examined the difference between adoption dates of annotations and adoption dates of generics. Our reasoning is that if concerns of compatibility was the primary factor holding back adoption, then we should observe near simultaneous adoption of both features once the concerns had been removed. Alternatively, if we observe large differences in adoption dates between the features, then some other factors may had held back adoption of a particular feature.

Non-Simultaneous Adoption in Most Established Projects We examined the dates of the first annotation and generic used in the established projects. Although we did find a few projects that introduced annotations and generics simulanteously, the majority of projects staggered adoption, often by years. Specifically, we found 4 projects adopted generics before annotations, ranging from months to years while 7 projects adopted annotations before generics, ranging from several days to years (for SUBCLIPSE annotations appeared 5 years before the first generic). Interestingly, LOG4J introduced annotations in 2007, but never introduced generics. There were 5 projects that first used annotations and generics on the same day. Overall, established projects staggered adoption between features by an average of 296 days. Figure 9 shows box plots depicting the number of days between adoption of generics and annotations.

Near-Simultaneous Adoption in Recent Projects Interestingly, the trend seen in established projects does not hold for recent projects. Instead, the recent projects were much quicker to use both features in a near-simultaneous fashion. We found 6 projects used generics before annotations, ranging from days to months, while 14 projects use annotations and generics on the same day. There was a near-simultaneous adoption of annotations and generics (an average 53 days lag), suggesting that projects used the available features in a major language upgrade together.

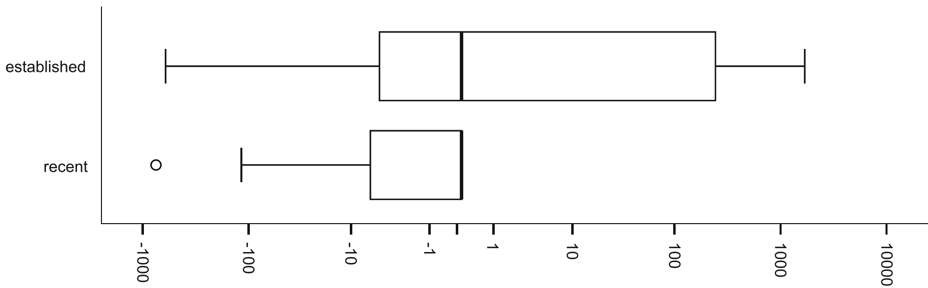


Fig. 9 *Box plots* displaying the number of days between when generics were introduced and when annotations were introduced in established and recent projects. *Negative values* indicate that annotations were introduced before generics

This delay of 53 days is significantly shorter than the 296 days experienced by established projects ($p < .02$ by an unpaired 2 tailed t-test).

If compatibility was the sole important factor, we might have expected more simultaneous adoption. Still, we do believe compatibility plays a role. For example, we did see examples of people holding back code (e.g., `List/*<String>*/`) and a few projects adding in both features on the same day. Further, there is evidence of delay between the release of Java 5 and adoption of either annotations or generics. We found an average adoption lag of 500 days after the official release by the established projects. However, other factors delay adoption even further (an average 296 days).

Overall, from the data that we collected to answer if compatibility is the sole factor in Research Question 3, the results indicate that **compatibility is an important, but not sole factor in adoption and other factors such as legacy code may contribute to even further delays**.

7.3.2 IDE Support

To evaluate IDE support, we first had to determine which projects used which IDEs and were active prior to IDE support (the 20 established projects). 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 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 established 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.[3]

> 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. However, we found no trend related to when generics were adopted. For instance, 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 6 shows that Squirrel-SQL and JEdit introduced generics into the code at a rapid rate once the decision to use them was made. 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. MiGen, a recent project, is shown in Fig. 6 to illustrate a recent project where no migration took place.

Overall, the data that we collected to answer the factor of IDE support in Research Question 3, the results 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. Finally, although we did not investigate tool support for annotations, we did observe several instances were annotations appeared to be introduced via tool support.

[3] http://www.theserverside.com/news/thread.tss?thread_id=37183

8 Limitations

There are several threats to validity in this study.

External Validity The projects we have sampled are all open-source projects, and they may not be representative of all software development projects. For example, certain industries, such as the defense industry, have stricter standards and slower timelines in supporting new versions of software, such as language runtimes, which may amplify or alter the conclusions of the study.

Even within open-source projects, the number of projects and the type of categories we have selected from may not be sufficient to draw conclusions for all open-source projects. Although, the data we have examined has highlighted several significant results, future research should confirm these findings at a larger scale within the open-source community.

General Validity The conclusions of this study are particular to adoption of language features in Java and may not hold for other languages. For example, a parallel adoption story of generics exists in C#—generics were also introduced in a new version of the language; however, subtle differences in the design and deployment of C# generics may have resulted in a different adoption story.

Further, the conclusions about the language features we have examined—Java generics and annotations—may not extend to other newly introduced language features such as Java closures. Future research needs to draw parallels between differences in adoption of language features and channel differences as insight into future design of language features.

Construct Validity Several conclusions in our study rely on complex analysis techniques. Limitations in those analysis techniques may have caused some results to be underestimated. For example, the migration analysis relies on the assumption that a raw type is migrated to a generic type if the fully qualified name of the method remains the same across revisions. This assumption may fail to count migrations that occurred during structural changes such as a file or signature rename. Note, that this assumption is not used for the other analyses, which tracks features at a project-wide level per revision.

In other cases, an analysis may only offer one perspective on the data when multiple perspectives might be needed. For example, one limitation of the cast analysis is that it is coarse-grained, examining the general relationship of casts and generics. However, the analysis is not sufficient for understanding why that relationship exists. In future work, a more fine-grained analysis can identify individual casts that were removed due introductions of generic functionality and compare that with other contexts for removal.

9 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 established projects and developers we studied *did not* use generics; for those that did, use was consistently narrow.

Empirically, we have found the usage of generics are almost entirely accounted by standard library collections, dwarfing the usage of user-defined generic types and methods. Additionally, given all the advanced parameterization options, their actual use appeared sparingly. We also found several places where the concept of generics were prematurely generified. Generics assumes that there are multiple candidates for parameterization. Instead, in practice we see that half of generic classes are only instantiated with one type, the other half with just a handful—only a very small generic classes are instantiated with numerous types.

Overall, the patterns of usage could indicate that a language enhancement as large scale and sweeping as generics may have been more than what was really needed. The disparity of different usage patterns presents an interesting conundrum for the language designer—should language features be added to address exceptional cases? Were there simpler solutions that language designers could have considered? For instance, had the language designers of Java generics instead opted to introduce a single `StringList` class, then they would have succeeded in satisfying a significant portion of Java generic usage. Are there more concise and incremental methods of introducing language features that language designers may consider?

Validating the many claims surrounding the benefits of generics remains a challenge. Our data only scratches the surface. Although we found merit to Danovan's claim that manually maintaining code clones would be costly, we found the impact to be limited to a few generic classes that are instantiated many times. And while our data indicates that generics reduce casts in most projects, a few projects showed the opposite trend. In future studies, we would like to investigate in more detail the underlying reasons and other unanswered claims. For example, developers may still be required to use casts in certain situations such as an implementation of `.equals()` or interfacing with older libraries.

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.

The adoption of generics by established projects may have been encumbered by issues other than backward compatibility. Some features may be more difficult to introduce than others. Projects with legacy code at risk may have found it more difficult to introduce generics than annotations. For example, introducing a generic type in a method signature may have an unintended consequence of changing many more method signatures than the programmer had signed up for. In contrast, an annotation can be easily added to a method with little impact. As evidence, we did see more projects adopt annotations over generics sooner. We also saw that very few projects made the effort to migrate old code to take advantage of generics. But certainly other features such as developer familiarity or prior exposure with features may have played a role as well. Interestingly, these issues do seem to recede with time, as we see more recent projects quickly embrace both new features at nearly the same time.

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 occurrences 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.

10 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.

Acknowledgements Thanks to NCSU students Brad Herrin, Donghoon Kim, Michael Kolbas, and Chris Suich, who contributed code to our analysis framework. Thanks to Jonathan Aldrich, Andrew Black, Prem Devanbu, Mike Ernst, Ron Garcia, Gail Murphy, Zhendong Su, and Thomas Zimmerman, who provided valuable advice.

Errata

In the MSR paper on which this paper is based (Parnin et al. 2011), we made three mistakes that have been corrected in this article. Because of these corrections, the results in this paper supersede the results from the MSR paper. We highlight the corrections here.

First, our time series analysis of casts versus generics undercounted the number of casts and generics. The time series appears in Fig. 5, along with a corrected correlation analysis (Section 6.1). This change reverses our original conclusion, which originally stated that generics do not have a strong influence on casts in a project.

Second, we originally miscounted the number of generic language features due to two bugs in our analysis software. The corrected numbers and graphs appear throughout this paper. The corrected numbers and graphs do not change our original conclusions because the shape of the data remains nearly identical.

Third, our original example of a generic method declaration in Section 2.2 was not correctly typed code. The new example is correctly typed.

Appendix

In this Appendix, we show extended figures for all projects.

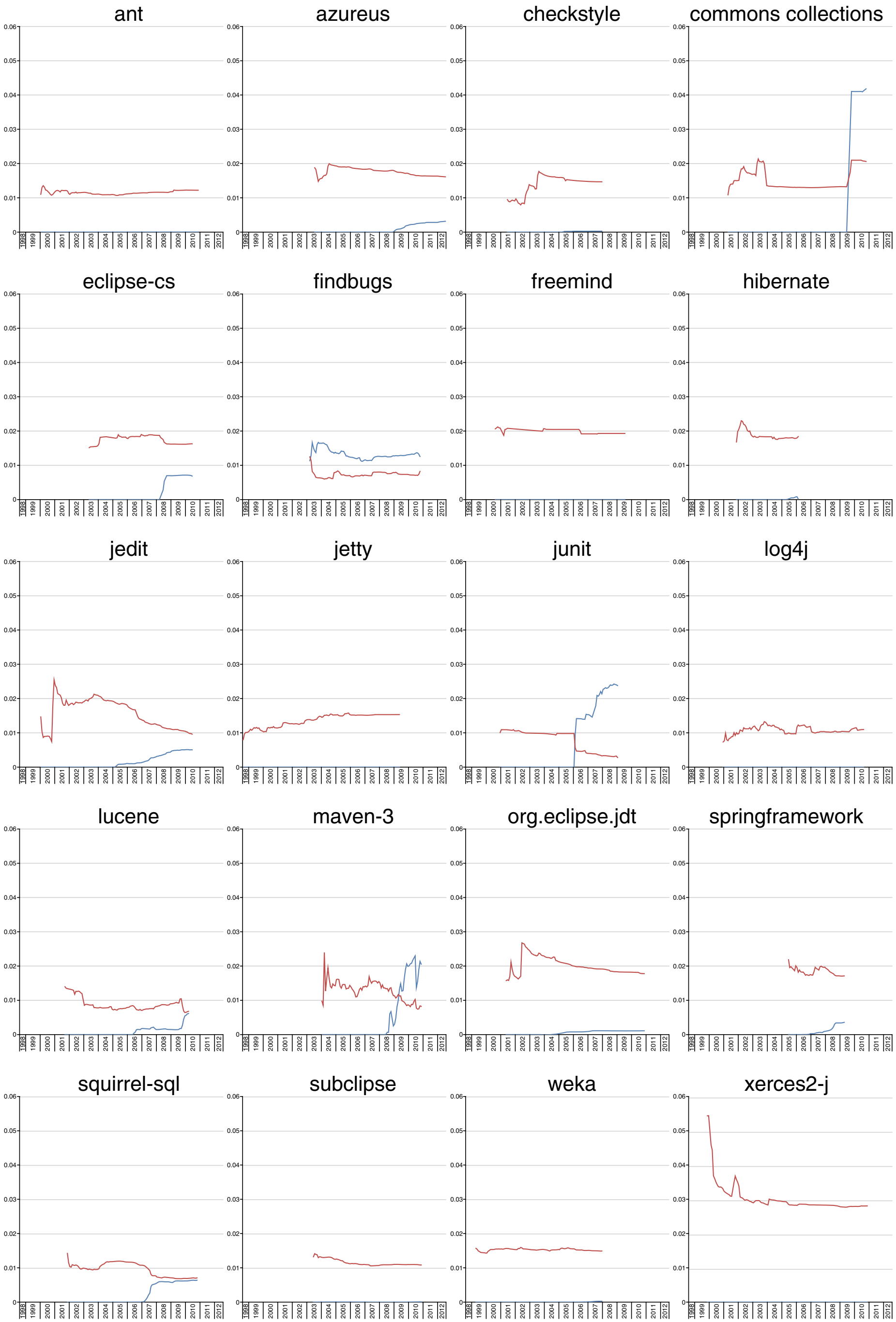


Fig. 10 Cast versus generic density in established projects

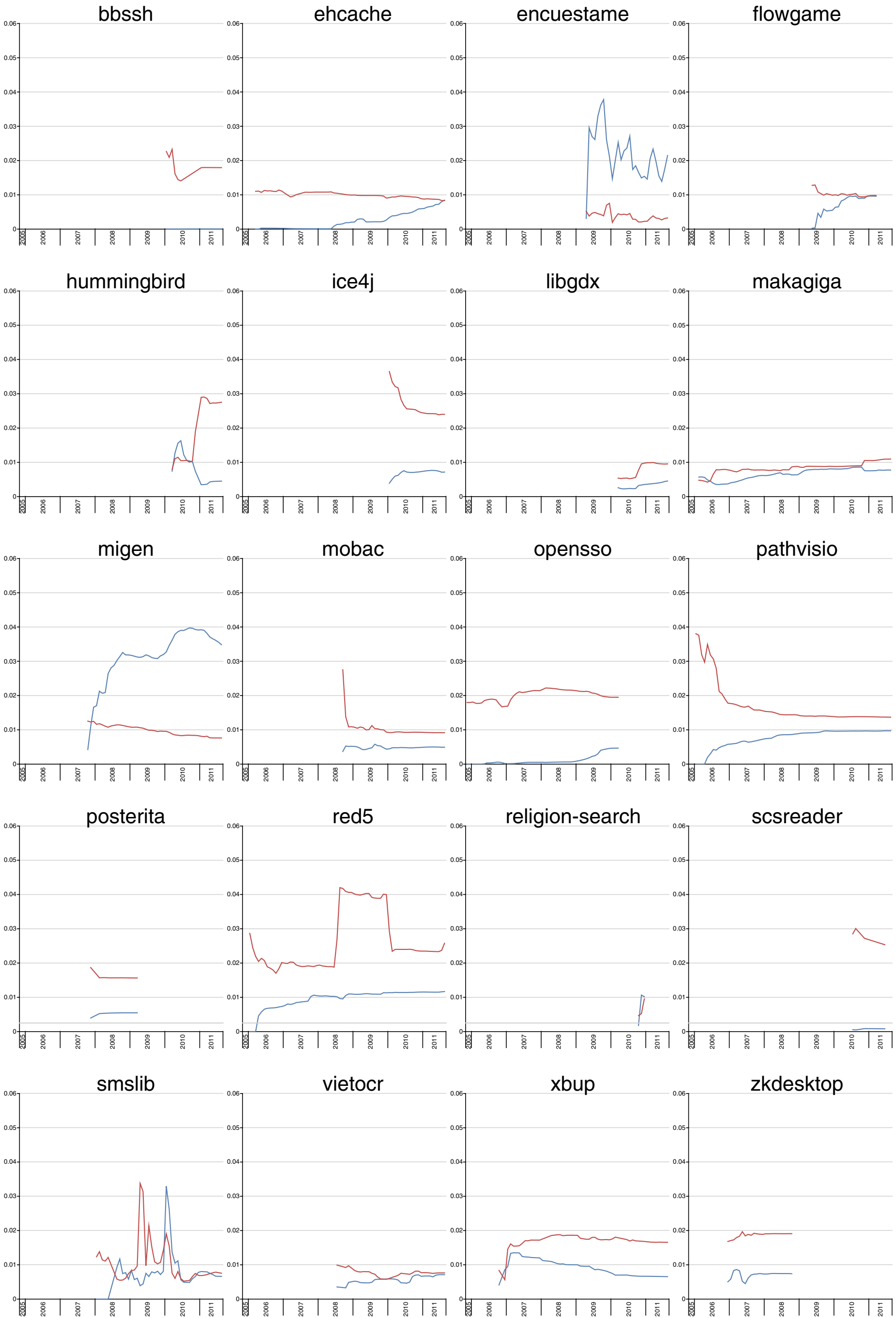


Fig. 11 Cast versus generic density in recent projects

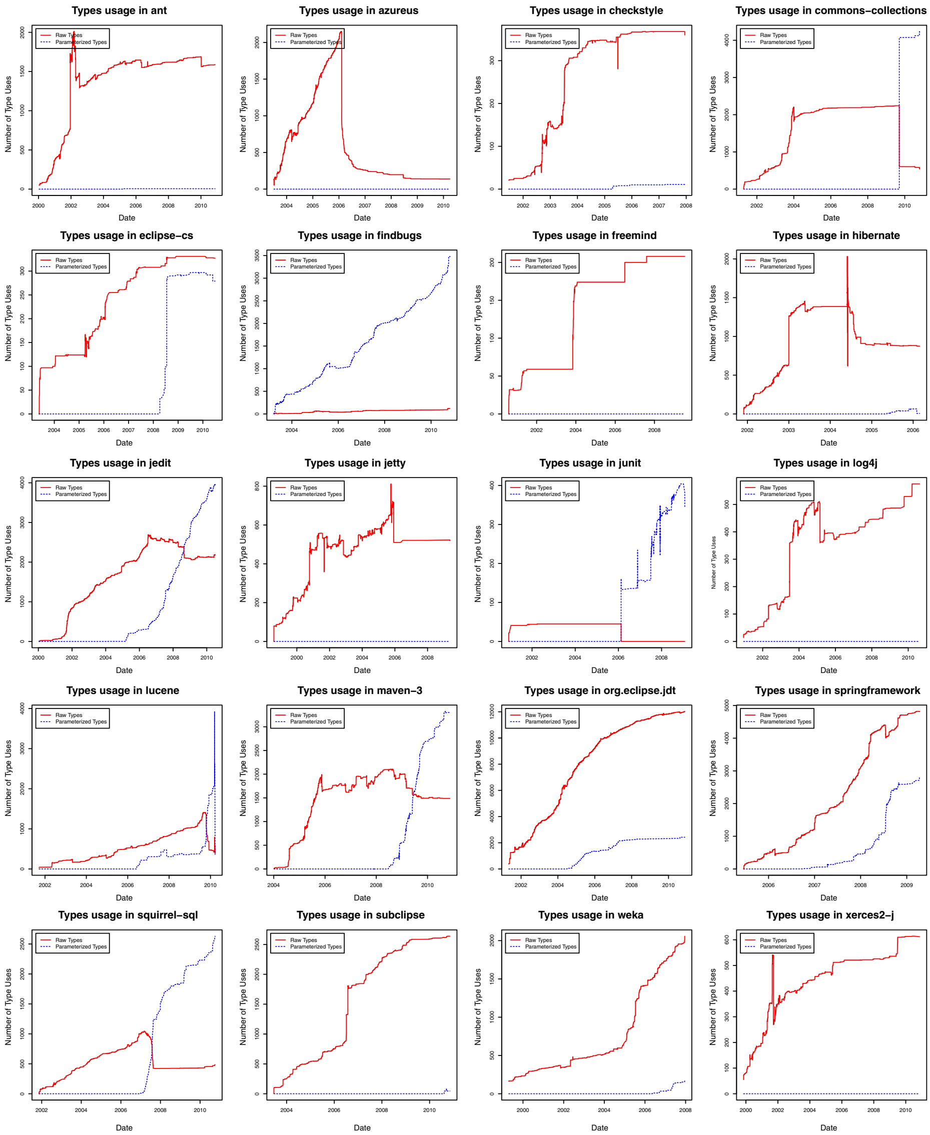


Fig. 12 Usage of raw and generic types in established projects

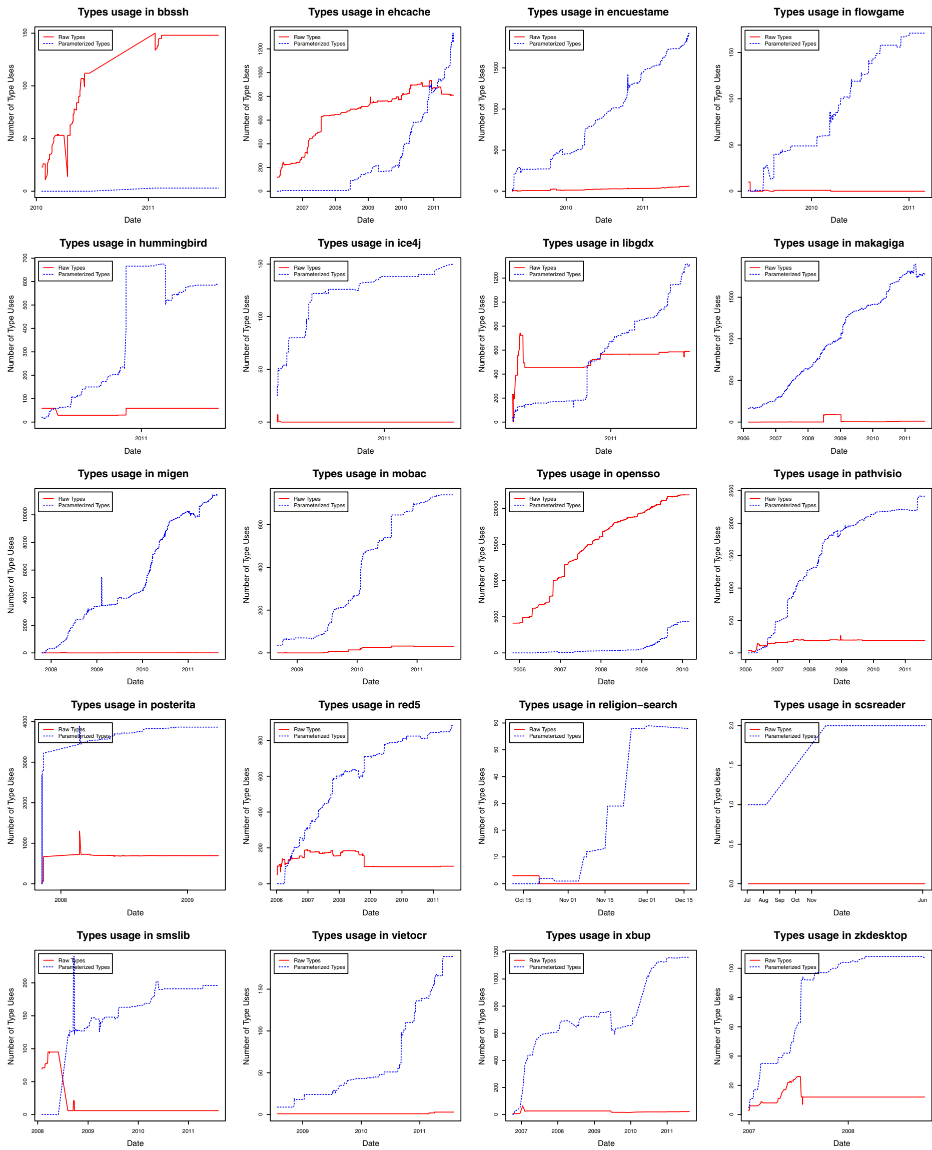


Fig. 13 Usage of raw and generic types in recent projects

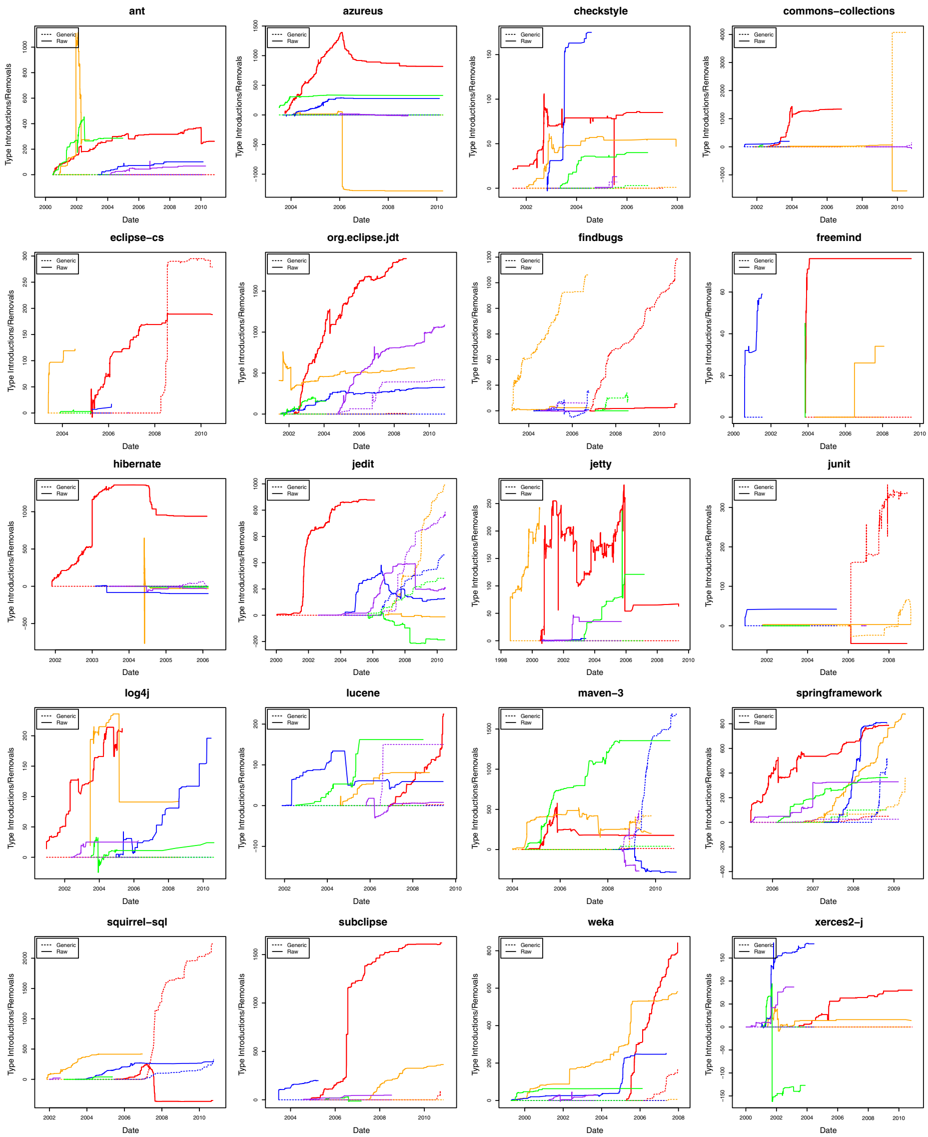


Fig. 14 Contributors' introduction and removal of parameterized types over time in established projects

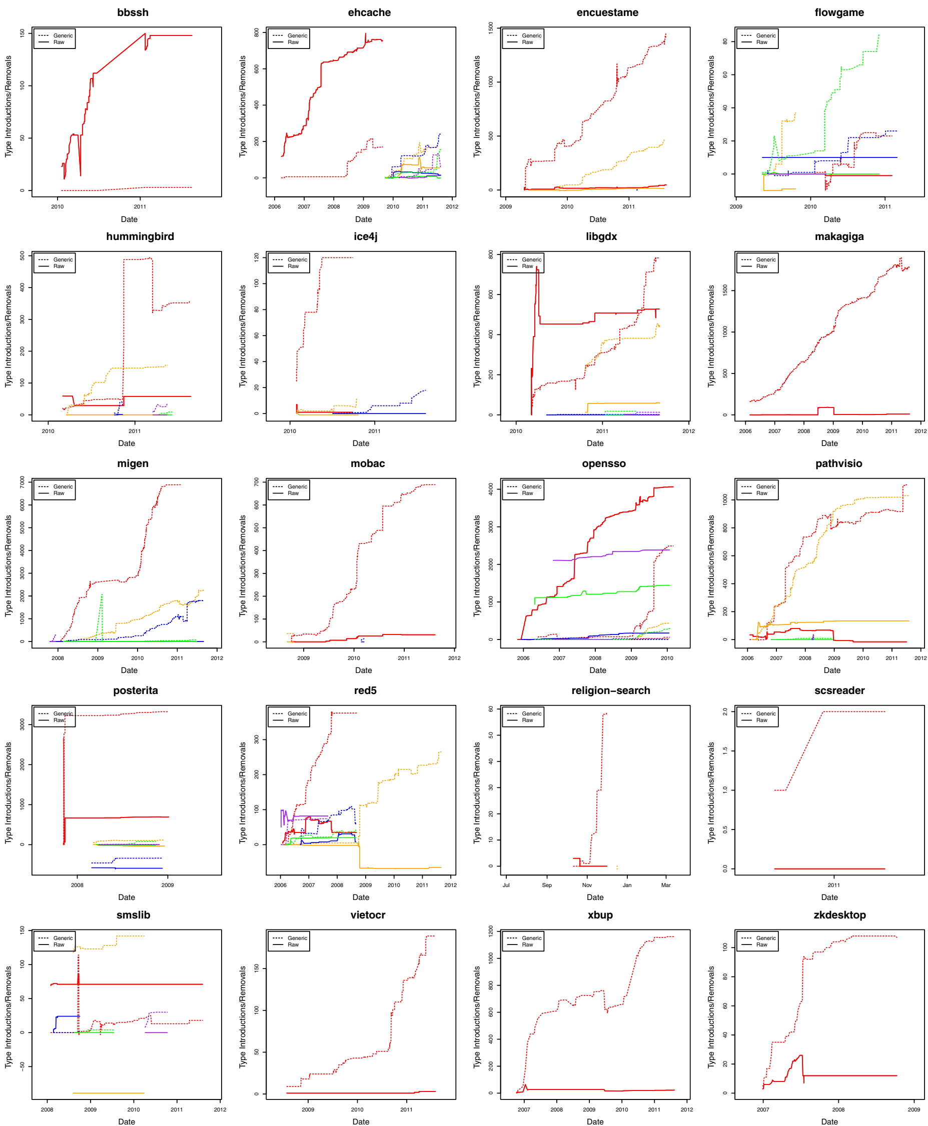


Fig. 15 Contributors' introduction and removal of parameterized types over time in recent projects

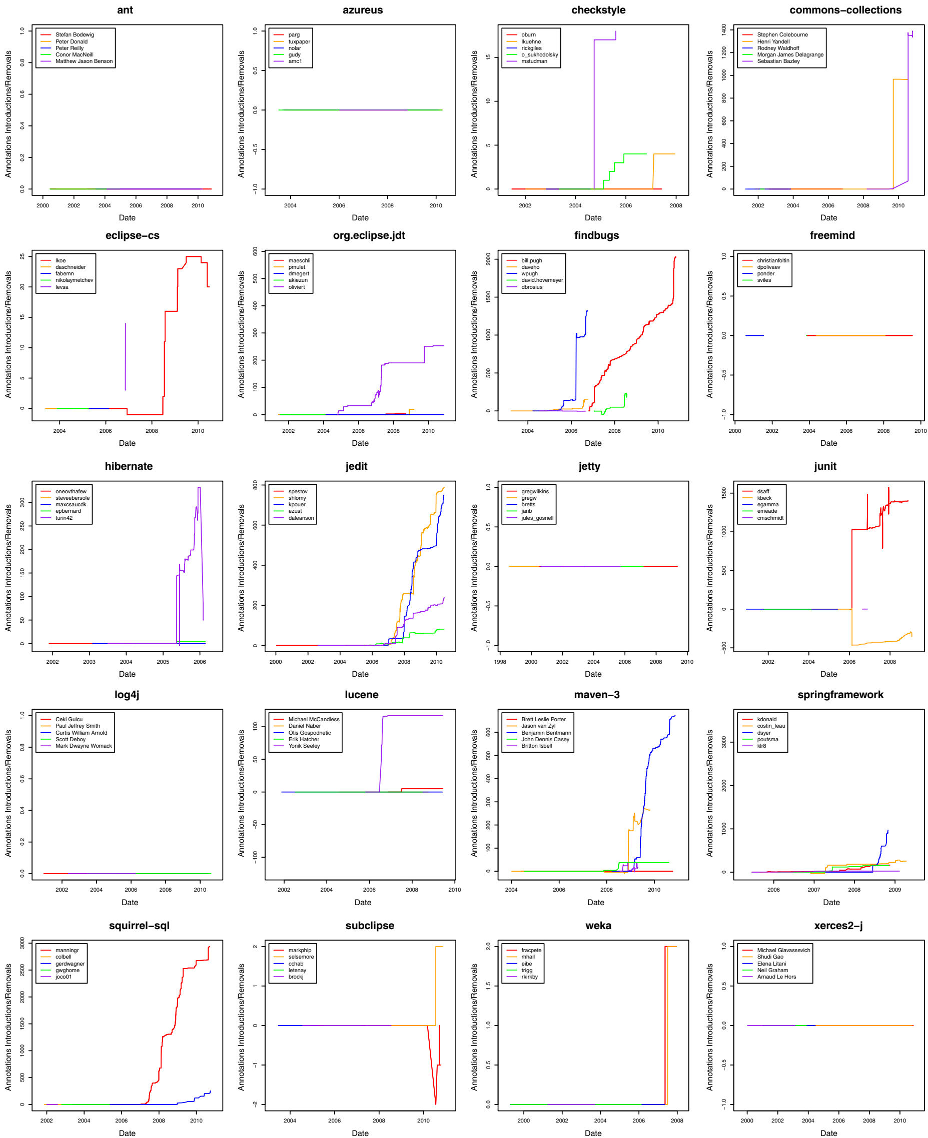


Fig. 16 Contributors' introduction and removal of annotations over time in established projects

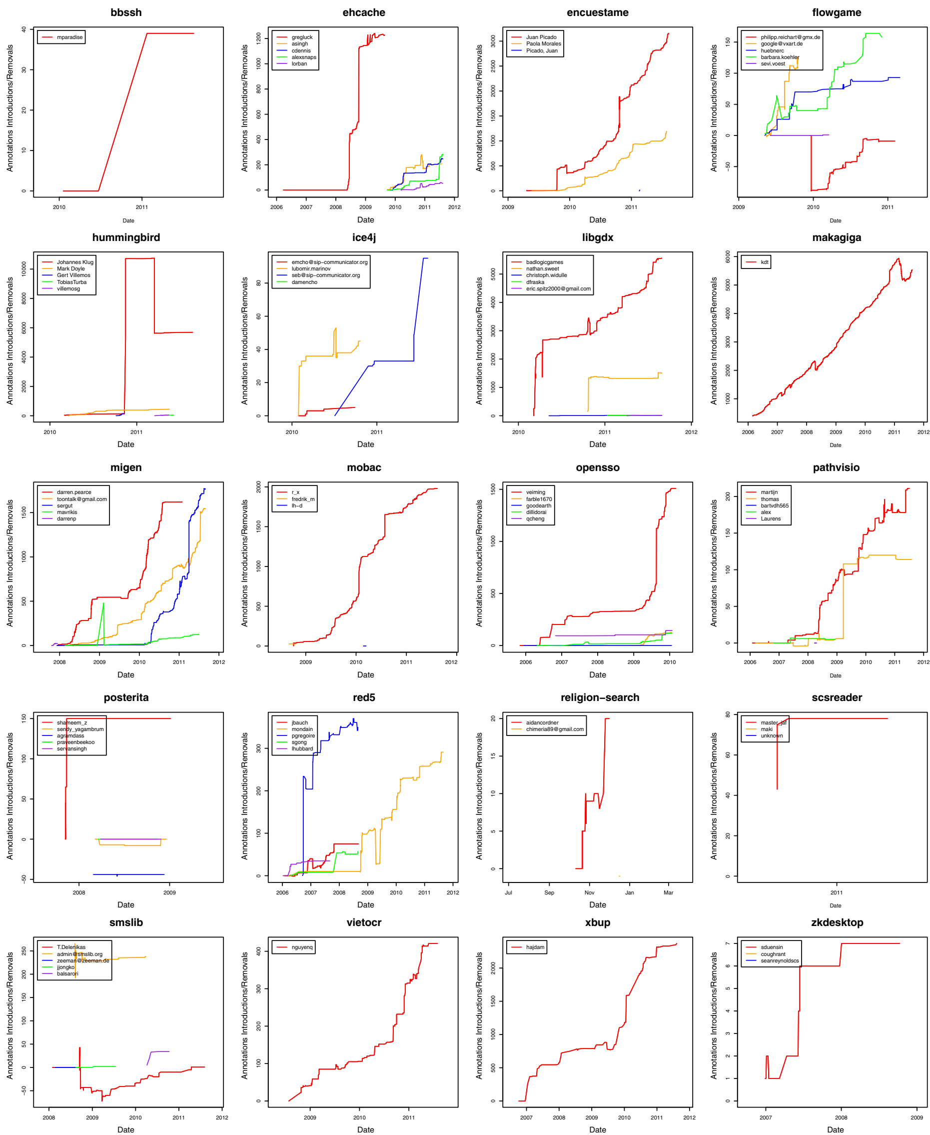


Fig. 17 Contributors' introduction and removal of annotations over time in recent projects

References

Basit H, Rajapakse D, Jarzabek S (2005) An empirical study on limits of clone unification using generics. In: Proceedings of the 17th international conference on software engineering and knowledge engineering, pp 109–114

Benjamini Y, Hochberg Y (1995) Controlling the false discovery rate: a practical and powerful approach to multiple testing. J R Stat Soc B 57(1):289–300

Bloch J (2008) Effective Java, 2nd edn. Prentice-Hall PTR
Bracha G (2005) Generics in the java programming language. Web. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf. Accessed 1 Mar 2012
Bracha G (2012) Lesson: generics. Web. http://download.oracle.com/javase/tutorial/extra/generics/index.html. Accessed 1 Mar 2012
Donovan A, Kieżun A, Tschantz, MS, Ernst MD (2004) 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
Dowdy S, Wearden S, Chilko D, (2004) Statistics for research, 3rd edn. Wiley, New York
Ducheneaut N (2005) Socialization in an open source software community: a socio-technical analysis. Comput Support Coop Work 14(4):323–368
Flanagan C, Leino KRM, Lillibridge M, Nelson G, Saxe JB, Stata R (2002) Extended static checking for java. SIGPLAN Not 37:234–245
Fuhrer R, Tip F, Kieżun A, Dolby J, Keller M (2005) Efficiently refactoring java applications to use generic libraries. In: Eurpoean conference on object oriented programming, pp 71–96
Geiger R, Fluri B, Gall H, Pinzger M (2006) Relation of code clones and change couplings. FASE 3922:411–425
Halstead MH (1977) Elements of software science (operating and programming systems series). Elsevier Science Inc., New York, NY, USA
Humphrey WS (1995) A discipline for software engineering. Addison-Wesley Longman Publishing
Java Language Guide: Annotations (2012). Web. http://download.oracle.com/javase/1.5.0/docs/guide/language/annotations.html. Accessed 1 Mar 2012
Liebig J, Kästner C, Apel S (2011) Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In: Proceedings of the tenth international conference on aspect-oriented software development, AOSD ’11. ACM, New York, NY, USA, pp 191–202
Markstrum S (2010) Staking claims: a history of programming language design claims and evidence. In: Proceedings of the workshop on the evaluation and usability of programming languages and tools
Mockus A, Fielding R, Herbsleb J (2002) Two case studies of open source software development: apache and mozilla. ACM Trans Softw Eng Methodol 11(3):309–346
Monden A, Nakae D, Kamiya T, Sato S, Matsumoto K (2002) Software quality analysis by code clones in industrial legacy software. In: Proceedings of the 8th international symposium on software metrics
Naftalin M, Wadler P (2006) Java generics and collections. O’Reilly Media, Inc
O’Mahony S, Ferraro F (2007) The emergence of governance in an open source community. Acad Manage J 50(5):1079–1106
Pankratius V, Adl-Tabatabai A, Otto F (2009) Does transactional memory keep its promises?: results from an empirical study. Technical Report 2009-12, Universität Karlsruhe, Fakultät für Informatik
Papi MM, Ali M, Correa Jr TL, Perkins JH, Ernst MD (2008) Practical pluggable types for java. In: Proceedings of the 2008 international symposium on software testing and analysis, ISSTA ’08. ACM, New York, NY, USA, pp 201–212
Parnin C, Bird C, Murphy-Hill E (2011) Java generics adoption: how new features are introduced, championed, or ignored. In: Proceedings of the 8th working conference on mining software repositories, MSR ’11. ACM, New York, NY, USA, pp 3–12
Shi L, Zhong H, Xie T, Li M (2011) An empirical study on evolution of api documentation. In: Proceedings of the 14th international conference on fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software, FASE’11/ETAPS’11. Springer, Berlin, Heidelberg, pp 416–431
Storey M-A, Ryall J, Bull RI, Myers D, Singer J (2008) Todo or to bug: exploring how task annotations play a role in the work practices of software developers. In: Proceedings of the 30th international conference on software engineering, ICSE ’08. ACM, New York, NY, USA, pp 251–260
The Advantages of the Java EE 5 Platform: A Conversation with Distinguished Engineer Bill Shannon (2012) Web. http://java.sun.com/developer/technicalArticles/Interviews/shannon_qa.html. Accessed 1 Mar 2012
The Java Tutorials: Annotations (2012). Web. http://download.oracle.com/javase/tutorial/java/java00/annotations.html. Accessed 1 Mar 2012

Van Emden E, Moonen L (2002) Java quality assurance by detecting code smells. In: Proceedings of the ninth working conference on reverse engineering (WCRE'02), pp 97–106. IEEE Computer Society, Washington, DC, USA
Vandevoorde D, Josuttis N (2003) C++ templates: the complete guide. Addison-Wesley Professional
Zimmermann T (2006) Fine-grained processing of CVS archives with APFEL. In: Proceedings of the OOPSLA workshop on eclipse technology eXchange. ACM Press

Chris Parnin is a Phd Student at Georgia Tech. He walks the line between being a professional software developer and researching them. He specializes in empirical, cognitive studies, and user studies of software development. Contact him at chris.parnin@gatech.edu. http://www.cc.gatech.edu/~vector/.

Christian Bird is a researcher at Microsoft Research in Redmond, Washington. His interests are in empirical studies of software engineering, predominantly examining the problems encountered in large software development projects. He received his Ph.D. from U.C. Davis. Contact him at cbird@microsoft.com. http://research.microsoft.com/people/cbird.

Emerson Murphy-Hill is an assistant professor at North Carolina State University. His research interests include human-computer interaction and software tools. He holds a Ph.D. in Computer Science from Portland State University. Contact him at emerson@csc.ncsu.edu. http://www.csc.ncsu.edu/faculty/emerson.