# Usage Patterns of the Java Standard API

Homan Ma†, Robert Amor‡, Ewan Tempero‡
Department of Computer Science
University of Auckland
Auckland, New Zealand
†hma007@ec.auckland.ac.nz
‡{trebor,ewan}@cs.auckland.ac.nz

## Abstract

*The Java Standard API has grown enormously since Java's beginnings, now consisting of over 3,000 classes and 20,000 methods. The intent of this API is to provide high quality components that can be easily reused and so increase the Java developer's productivity — but does it? In this paper, we present a study that begins to answer this question. Specifically we take a corpus-based approach to help determine the "typical" usage of the Standard API. We find that, in an extensive corpus of open-source software, only about 50% of the classes in the Standard API are used at all, and around 21% of the methods are used. We discuss the implications this has for future development of both the API itself, and for tools to support the API.*

*Keywords:* Java Standard API, Software Repositories, Reuse

## 1. Introduction

Repositories of reusable software components have been proposed and built for several decades now, although their success has been limited. Reasons given for the lack of success include the components are not of sufficient quality, and lack of appropriate tool support [7, 8, 20]. The Java Standard API [17] is interesting in that it is quite large, but has almost no tool support beyond a browser for its hierarchical structure. Attempts to provide a better search facility have not had convincing success (e.g., [19, 10, 13]). A reasonable question to consider is, does the Java Standard API need any tool support? Perhaps it is successful enough that devoting resources to such endeavours is unnecessary. In this paper, we attempt to answer this question, and present the first study into the usage patterns of the Java Standard API.

The basis of our study is a corpus of open-source Java software we have been collecting at the University of Auckland. The original intent of this corpus was to provide material for research into design quality. However, we have quickly found many other uses for it, including the study presented in this paper. The corpus provides a view of what might be considered "typical" Java development, and so give us insight as to typical usage patterns of the Java Standard API.

The paper is organised as follows. In the next section, we discuss relevant background material and related work. In section 3, we discuss how we carried out our study, giving details of our corpus and the tools we developed. Section 4 presents the results we have obtained, and we discuss these results in section 5. Finally, we present our conclusions.

## 2. Related Work

There has been considerable research into software reuse in general (e.g., [12, 16, 4]), and the development of software repositories in particular [7, 8, 20]. As far as we are aware, there have been no other studies examining how the Java Standard API is used, or indeed any code library or reuse repository.

There is, however, a growing trend of using corpora of software for various studies. Knuth was one of the first to carry out empirical studies using a corpus [11]. He presented static analysis of over 400 FORTRAN programs and dynamic analysis of about 25 programs. His main motivation was compiler design, with the concern that compilers may not be optimised for the typical case as no-one knew what the typical case was. His analysis was at the statement level, counting such things as the number of occurrences of an `IF` statement, or the number of executions of a given statement.

More recently, Collberg et al. have carried out a study of 1132 Java programs [2]. These were gathered by searching for `jar` files with Google and removing any that were invalid. Their main goal was the development of tools for the protection of software from piracy, tampering, and reverse engineering. Like Knuth, they argued that their tools could benefit by knowing the typical and extreme values of various aspects of software. Consequently, their interest is in the low-level details of the code with a view toward future tool support or language design.

Gil and Maman analysed a corpus of 14 Java applications for the presence of *micro patterns*, patterns at the code level that represent low-level design choices [5]. They found that 3 out of 4 classes matched one of the 27 micro patterns in their catalogue, and just over half of the classes are catalogued by just 5 patterns.

Do et al. describe the Software-artifact Infrastructure Repository (SIR), an infrastructure developed to support controlled experimentation with software testing and regression testing techniques [3]. This includes a corpus consisting of multiple versions of several C and Java applications, together with test suites and fault data. SIR is closest in nature to our corpus as it is being developed to support a range of studies, rather than gathered together for a specific study.

Our own corpus has seen use in other studies. Some have involved investigating the presence of dependency cycles, the original motivation for developing our corpus [15, 14]. Others have involved investigating the use of "dependency injection" [18] and for a study into the existence of power-law distributions of relationships [1].

The study presented here has more in common with corpora-based applied linguistics, where corpora of (for example) documents (both written and spoken) in a given language are used to investigate how that language is used. Our corpus is what Hunston describes as a reference corpus [9].

## 3. Methodology

The main question to consider for this study is what exactly we should measure. A program consists of many entities. A Java entity can be a package, class type (normal or enum), interface type (normal or annotation type), member (class, interface, field, or method) of a reference type, type parameter (of a class, interface, method or constructor), parameter (to a method, constructor, or exception handler), or local variable. We classify the entities from the Java API being used within a Java source code file as types, classes, methods or packages, as described below and illustrated in Figure 1. API *types* consist of:

- Imported types. The type *Random* is imported from the type *java.util.Random* of the package *java.util*.
- Fields, class variables and instance variables of classes, and constants of interfaces. The type *int* is the type declared by the identifier *divisor*.

```
import java.util.Random;

class MiscMath {
  int divisor;
  MiscMath(int divisor) {
    this.divisor = divisor;
  }
  float ratio(long l) {
    try {
      l /= divisor;
    } catch (Exception e) {
      if (e instanceof ArithmeticException)
        l = Long.MAX_VALUE;
      else
        l = 0;
    }
    return (float)l;
  }

  double gausser() {
    Random r = new Random();
    double[] val = new double[2];
    val[0] = r.nextGaussian();
    val[1] = r.nextGaussian();
    return (val[0] + val[1]) / 2;
  }
}
```

**Figure 1. Sample Code (From Java Language Specification, First Edition)**

- Method parameters. The method parameter of the method *ratio* is declared to be of the type *long*.
- Method results. The return type of the method *ratio* is of the type *long* and the return type of the method *gausser* is of the type *double*.
- Constructor parameters. The parameter of the constructor for *MiscMath* is declared to be of the type *int*.
- Local variables. The local variables within the method *gausser* are declared of the types *Random* and *double[]*.
- Exception handler parameters. The exception handler parameter of the catch clause is declared to be of the type *Exception*.

API *classes* consist of:

- Class instance creations. A class instance creation expression that uses the type *Random* is used within the *gausser* method.
- Superclasses. The optional extends clause within the following class declaration *ModifiedLinkedList extends LinkedList* indicates that the API class

*java.util.LinkedList* is the superclass of the class *ModifiedLinkedList* being declared.

- Superinterfaces. The optional implements clause within the following class declaration *NewAction implements AccessibleAction* indicates that the API interface *javax.accessibility.AccessibleAction* is a direct superinterface of the class NewAction being declared.
- Static classes. These are classes that contain static fields and/or methods. Such as the *java.lang.Math* class.

API *methods* consist of:

- Instance methods. They require an instance of a class to be declared before they can be invoked. The code in Figure 1 demonstrates the method `nextGaussian()` of the class *java.util.Random* is invoked only after an instance of the Random class is instantiated.
- Class methods. Also referred to as static methods. e.g. *Math.random()*

Each time an API class or API method is identified within the source code, the corresponding API package that the class or method exists in can be regarded as being used.

## 3.1. Jepends Tool

The Jepends tools was original developed to analyse the source code of a system in order to identify classes that are possible refactoring candidates [15]. This tool has been modified to record the possible API uses (identified and listed above) found within the source code. Jepends takes the source code of an application and outputs three text files:

- Superclasses and Superinterfaces used.
- Class and Type instances.
- Instance and Class method invocations.

Further analysis is done on this raw data to produce the results shown in the next section.

## 3.2. Software Corpus

As mentioned earlier, our study uses a corpus of Java software we have been developing. At the beginning of the study, we used 39 open source Java applications that were in the corpus. These are listed as `Set 1` in the Appendix. These applications were selected from a number of sources, corpora used in other published papers (e.g., [6, 5]), and popular (widely down-loaded) and actively developed open-source Java applications from various websites including developerWorks[1], SourceForge[2], Freshmeat[3], Java.net[4], Java-Source.net[5] and The Apache Software Foundation[6].

[1]`http://www-128.ibm.com/developerworks/views/java/downloads.jsp`

[2]`http://sourceforge.net/`

**Table 1. API usage**

| | Package | | Class | | Method | |
|---|---|---|---|---|---|---|
| Set 1 | 87 | 88.8% | 1396 | 51.2% | 4114 | 20.8% |
| Set 2 | 75 | 76.5% | 984 | 36.1% | 2473 | 12.5% |
| All | 88 | 89.8% | 1436 | 52.7% | 4150 | 21.0% |

`Set 1` was gathered with no regard to likely usage of the Java API. After obtaining the results for `Set 1`, we decided to extend the corpus in a deliberate attempt to improve the coverage of the Java API. The applications were chosen from Java-Source.net. The basis for choosing a reliable application from each software category was to get a broader spectrum of functionality. Care was taken to make sure that the latest stable release of each application was used. The reason behind this is that the latest stable release would contain the most functionality and so the most API usage. The 37 new applications are listed as `Set 2` in the Appendix.

## 3.3. Java API Reference

The Java 1.4.2 API was used as a reference. Java 1.4.2 was used rather than Java 5.0 as most of the open source applications used were originally compiled with Java 1.4.2 and therefore would not contain new Java 5.0 functionality. The API package and class list were taking directly from the Java 2 Platform, Standard Edition, v 1.4.2 API Specification documentation and the API method list was created by using Java reflection together with the API class list collected (making sure that the compiler was set at 1.4.2). This gave us all of the packages, classes, and methods described in the public documentation for a Java developer, and so we felt best represents the “size” of the Java Standard API. The results are:

| | |
|---|---|
| API Packages | 98 |
| API Classes | 2,723 |
| API Methods | 19,785 |

## 4. Results

In this section, we summarise our results, with discussion left until section 5.

Table 1 shows the API usage divided up into the API package, API class and API method use for each of `Set 1` and `Set 2` individually, and then the combined corpus. `Set 2` applications usage is slightly less than that of `Set`

[3]`http://freshmeat.net/`

[4]`http://community.java.net/projects/`

[5]`http://java-source.net/`

[6]`http://apache.org/`

| Top 20 Packages | | Top 20 Classes | | Top 20 Methods | | |
|---|---|---|---|---|---|---|
| Name | Frequency | Name | Frequency | Class | Method | Frequency |
| java.lang | 682742 | String | 225817 | String | equals | 72754 |
| java.util | 96132 | Object | 49709 | StringBuffer | append | 62074 |
| java.io | 72660 | Exception | 18427 | String | length | 39693 |
| java.awt | 66772 | List | 12261 | Iterator | next | 36516 |
| java.sql | 42100 | Iterator | 10224 | Iterator | hasNext | 31656 |
| javax.swing | 26330 | File | 8970 | StringBuffer | toString | 30034 |
| java.beans | 16170 | Class | 8029 | String | substring | 25129 |
| javax.swing.text | 14718 | StringBuffer | 7982 | List | size | 18119 |
| java.net | 13840 | Map | 7228 | String | indexOf | 17447 |
| javax.swing.text | 13348 | Throwable | 6068 | List | get | 16952 |
| java.awt.event | 12290 | Vector | 5558 | PrintWriter | println | 15016 |
| java.lang.reflect | 10624 | IOException | 5507 | Hashtable | put | 13344 |
| javax.naming | 8414 | Integer | 5388 | List | add | 12541 |
| java.util.logging | 7846 | SQLException | 5086 | String | charAt | 11715 |
| java.awt.geom | 6082 | Connection | 4626 | Map | get | 11351 |
| javax.naming.directory | 5144 | Element | 4470 | String | startsWith | 11268 |
| java.security | 4422 | ArrayList | 4339 | Vector | size | 11046 |
| java.math | 4106 | Properties | 4083 | ArrayList | size | 10902 |
| java.text | 3968 | Collection | 3825 | HashMap | put | 10387 |
| javax.swing.event | 3022 | URL | 3513 | Integer | intValue | 9993 |

**Table 2. Top 20 packages, classes, and methods used**

`1`. The combined usage shows only a slight increase over `Set 1`.

## 4.1. Entity Frequency

As expected, `java.lang` is the top package used as it provides the fundamentals of the Java programming language. Figure 2 shows the frequency of package usage ordered along the $x$-axis according to decreasing frequency. Each point $(x, y)$ represents the number of classes ($y$) that use exactly $x$ packages. The second graph is the `log-log` version of the first graph. We provide `log-log` graphs as we are also interested as to whether these distributions obey a power-law, which would be indicated by a straight line on the `log-log` graph.

The top 20 classes used also has no surprises for the most frequently used class (String). It is worth noting that if *types* were considered, then the second most frequently used type would be `int` (214025) and the third would be `boolean` (53387). Figure 3 shows the frequency of class usage ordered along the $x$-axis according to decreasing frequency and the `log-log` version.

Figure 4 shows the frequency of method usage and the `log-log` version.

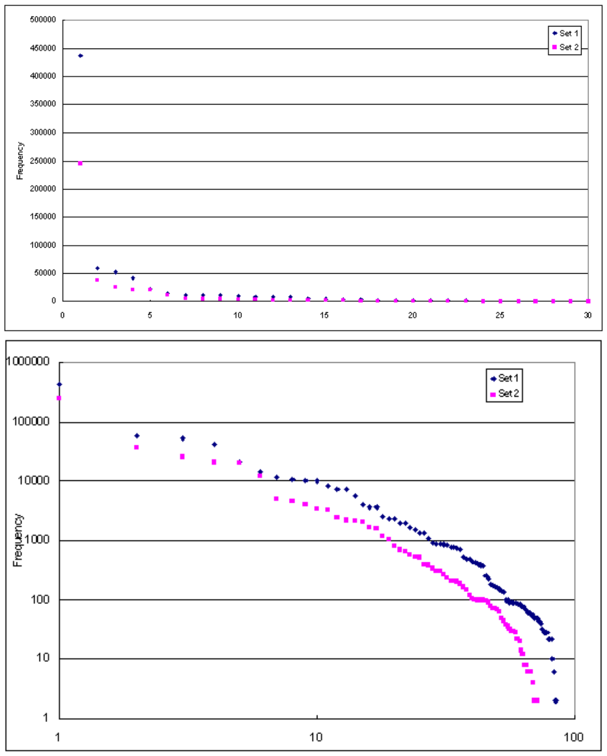

**Figure 2. Frequency of package use**

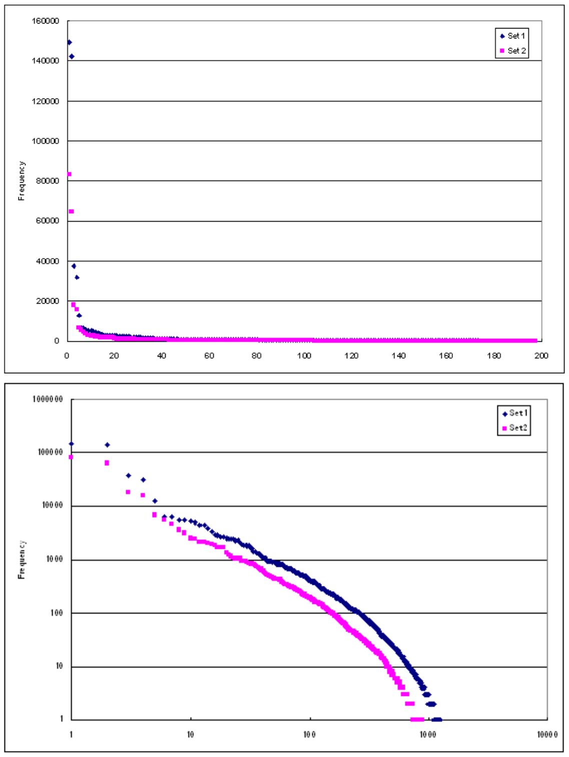

**Figure 3. Frequency of class use**

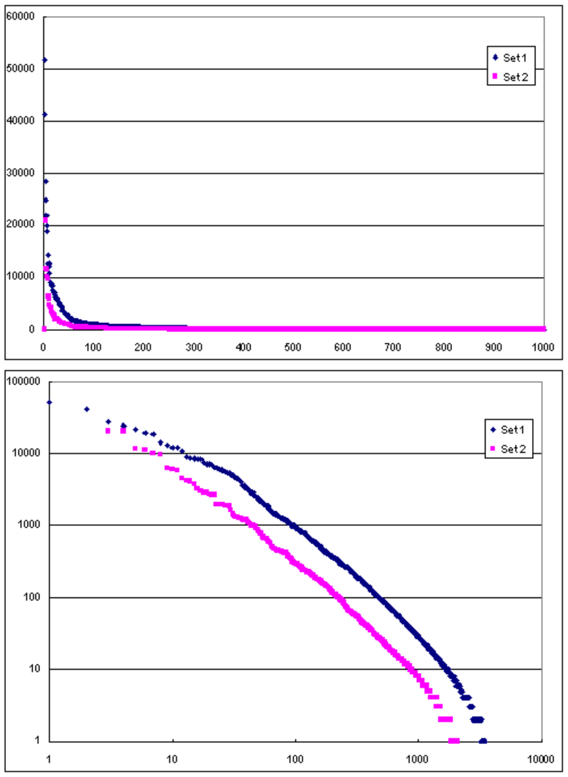

**Figure 4. Frequency of method use**

## 4.2. Cumulative API Usage versus Application size

Figure 5 shows the cumulative API usage as new applications are added in increasing order of size. The $y$ value of a point is calculated by taking the total number of API uses of the applications before, and then adding to that number the number of new API uses that the new application introduces. Unlike the earlier graphs, API usage is not the frequency of use, it is whether or not a particular API package, API class or API method is used at all. The intent of these graphs is to get a sense as to how representative our corpus is with respect to API usage.

## 4.3. Unused API Entities

The above results present what of the API is used in the corpus, but perhaps of more interest is what is not used.

The packages from which no classes are used in the full corpus are: `java/awt/dnd/peer` `java/awt/im` `java/awt/im/spi` `java/nio/charset/spi` `java/rmi/activation` `javax/imageio/plugins/jpeg` `javax/sound/midi/spi` `javax/sound/sampled/spi` `javax/swing/colorchooser` `javax/swing/plaf/multi`. A number of these packages related to the Service Provider Interface introduced in Java 1.3.

Table 3 attempts to provide a compact representation of the unused classes per package. In the last entry, the packages listed before the number in parentheses have that number of unused classes. We have given the number of classes unused, rather than a percentage of the package, as we feel this gives a better sense of how much code that is in the API has not been used.

Table 4 shows classes ranked by how many of their methods were unused in the corpus. A number of the classes involved are from the various user-interface related pack-

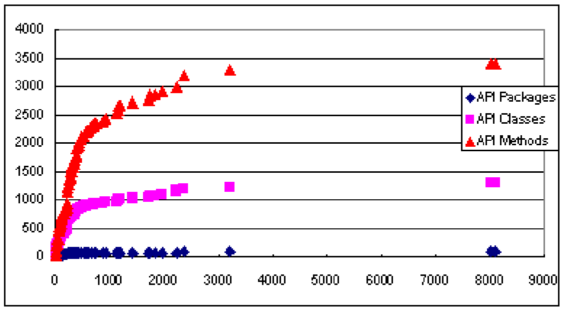

**Figure 5. Cumulative usage**

| | |
|---|---|
| 1 | java/awt/print, java/lang/reflect, java/nio/charset, java/nio/charset/spi, java/rmi/registry, java/util, java/util/jar, javax/security/auth/x500, javax/swing/filechooser, javax/swing/table, javax/swing/text/rtf, javax/xml/transform/dom, org/omg/CORBA/ORBPackage, org/omg/PortableServer/CurrentPackage, org/omg/PortableServer/POAManagerPackage, org/omg/stub/java/rmi |
| 2 | java/rmi/dgc, java/security/acl, javax/crypto/interfaces, javax/crypto/spec, javax/security/auth, javax/sql, javax/swing/border, javax/swing/event, org/ietf/jgss, org/omg/DynamicAny/DynAnyFactoryPackage, org/omg/IOP/CodecFactoryPackage, org/omg/Messaging, org/omg/SendingContext, org/xml/sax/helpers |
| 3 | java/awt/datatransfer, java/awt/im/spi, java/awt/image/renderable, java/sql, java/util/logging, java/util/zip, javax/imageio, javax/imageio/event, javax/naming/event, org/omg/CORBA/DynAnyPackage, org/omg/DynamicAny/DynAnyPackage, org/omg/IOP/CodecPackage |
| 4 | java/awt/im, java/nio/channels/spi, java/security/interfaces, javax/imageio/metadata, javax/imageio/plugins/jpeg, javax/security/auth/callback, javax/security/auth/kerberos, javax/security/cert, javax/sound/midi/spi, javax/sound/sampled/spi, javax/swing/colorchooser, javax/swing/text/html/parser, org/omg/PortableInterceptor/ORBInitInfoPackage |
| 5 | java/awt/color, java/nio, java/text, javax/imageio/spi, javax/naming/spi, javax/print/event, javax/rmi/CORBA, javax/swing/tree, javax/swing/undo |
| 6 | java/awt/dnd, java/security/spec, java/util/prefs, org/omg/CORBA/portable, org/omg/CosNaming/NamingContextExtPackage |
| 7 | java/awt/geom, java/lang, java/rmi, javax/naming/directory |
| 8 | java/beans, java/io, javax/imageio/stream |
| 10 | java/awt/font, javax/naming/ldap |
| >10 | java/awt/event, java/net, javax/crypto, javax/net/ssl (11); java/rmi/server (12); java/awt/image, java/security, javax/accessibility, javax/naming (13); org/omg/CosNaming/NamingContextPackage (14); java/rmi/activation, javax/print, org/omg/PortableServer/POAPackage (16); java/beans/beancontext, javax/swing/text/html (19); javax/sound/sampled (20); javax/print/attribute, org/omg/PortableInterceptor (21); java/nio/channels, javax/sound/midi, org/omg/CosNaming, org/omg/IOP (24); java/security/cert (28); javax/swing/plaf/multi (31); javax/swing (32); java/awt (34); javax/swing/plaf/metal (36); javax/swing/plaf (38); org/omg/PortableServer (43); org/omg/DynamicAny (48); javax/swing/plaf/basic (53); javax/swing/text (58); javax/print/attribute/standard (66); org/omg/CORBA (127); |

**Table 3. Number of unused classes per package**

ages. We speculate that the unused methods in these classes are in fact used by frameworks (SQL, Swing, CORBA), and so are not expected to be directly invoked by developer code. Notable exceptions are `java.util.Arrays` and `java.lang.Character`. Below the top 20, framework related classes still feature heavily.

## 5. Discussion

From the usage results gathered from our corpus of open-source software, in total only about 52% of the classes found within the Java 1.4.2 API are used, and around 21% of the methods are used. This is lower than one might expect. This raises questions about the representativeness of our corpus. It was these low results that led us to expand our corpus, the addition of new applications (`Set 2`) distinct from the initial corpus (`Set 1`). To prove the representativeness of `Set 1`, the new applications added were chosen from a broad range of functionality domains. Yet, there are no obvious differences between the results for `Set 1` and `Set 2`. In particular the combined results only slightly increase the usage. If the original corpus was not representative, we would expect to see marked differences once the applications of `Set 2` were considered. The cumulative graph in Figure 5 is consistent with this. The addition of the two largest applications changes the usage only a small amount. While this is not conclusive, it is very suggestive that our corpus is quite representative of the typical usage of the Java Standard API.

By and large, the most frequently used entities in the API are largely as one might expect. One surprise is how high `Exception` appears on the list of frequently used classes. We speculate this is due to a number of `try` blocks catching the most general exception rather than something more specific. `StringBuffer`'s position also seems somewhat surprising, until one realises that the "`String`" concatenation operator "+" is in fact translated by the compiler to a `StringBuffer` operation (and hence the position of `append` in the method list top 20). `PrintWriter.println` is in fact most likely a consequence of calls to `System.out.println`.

It is difficult to characterise the unused parts of the API.

|  |  |
|---|---|
| java.sql.DatabaseMetaData | 164 |
| java.sql.ResultSet | 139 |
| java.awt.Component | 133 |
| java.sql.CallableStatement | 79 |
| java.util.Arrays | 76 |
| java.lang.Character | 75 |
| java.awt.AWTEventMulticaster | 73 |
| javax.swing.JComponent | 72 |
| javax.swing.JTree | 65 |
| javax.swing.JTable | 61 |
| org.omg.DynamicAny._DynUnionStub | 60 |
| org.omg.DynamicAny._DynValueStub | 60 |
| javax.imageio.metadata.IIOMetadataNode | 59 |
| javax.sql.RowSet | 58 |
| org.omg.DynamicAny._DynSequenceStub | 57 |
| org.omg.DynamicAny._DynStructStub | 57 |
| javax.swing.AbstractButton | 55 |
| org.omg.DynamicAny._DynArrayStub | 55 |
| org.omg.DynamicAny._DynEnumStub | 55 |
| javax.swing.DebugGraphics | 54 |

**Table 4. Top 20 classes ordered by number of unused public methods**

Generally there is a sense that a large amount of code in the API is not providing benefit to Java developers. The results for the API itself might explain why some of the code was written in the first place, however we suspect that the growth of the API is also an issue. The very first Java API, released in 1996 had approximately 200 reusable Java classes grouped into 8 software packages. With the latest release of Java 5.0, this number has reached approximately 3000 grouped into 166 packages, with about 250 new classes compared to the Java 1.4.2 release. New library extensions in Java 5.0 included extensions of the *lang, util* packages, networking features, support for more security standards (SASL, OCSP, TSP), internationalisation and the list continues. Following this trend, the next version Java 6.0 (mustang) will introduce even more functionality.

It is possible that many of the classes with low usage are due to their relatively recent introduction, meaning they are not that well know. The `java.nio` and `java.*.spi` packages are perhaps good examples of this. It would be interesting to study whether developers' uses of the API are lagging behind the Java API releases, that is that use of new features only appears after further API releases occur.

### 5.1. API Usage by API

The corpus we studied contains applications from a number of domains. One "application" we initially did not consider was the API itself. Examination of this turns out to be very interesting. The usage is as follows:

|  |  |  |
|---|---|---|
| Package | 97 | 95.6% |
| Classes | 1980 | 72.7% |
| Methods | 14468 | 73.1% |

This is a significant difference to the results for our corpus, and suggests that a significant role of the API is in fact to support the API itself, rather than to provide components for use outside of the API. This is consistent with our observations that many of the classes with large numbers of unused methods are part of various frameworks.

Figures 2 3 and 4 do not appear to show power-law distributions. As how many packages, classes, or methods used in a class is likely to be roughly related to the size of the class, this is consistent with other studies that indicate that size-related frequency distributions tend not to be power-laws [1].

### 5.2. Threats to Validity

There are some limitations to our study that may impact our conclusions. We have presented our results in terms of Java 1.4.2, although we are aware that some more recent releases may in fact contain some Java 5.0 constructs. Manual investigation suggests this is minimal.

We claim our corpus is fairly representative, but we have only considered open source applications. Commercially-developed applications may exhibit different characteristics.

We do not count the use of fields from API classes. A common usage of this type is when the fields represent constants.

## 6. Conclusions

We have carried out a study into how the Java Standard API is typically used in a corpus of 76 open source Java applications. About 50% of the Java classes and about 80% of the Java methods are not used at all. Some of this may be explained by the use of frameworks. It appears that a significant part of the published API appears not relevant to most developers. This raises questions as to whether some part of the API could be separated as "infrastructure", leading to a smaller and simpler view of the API used by a Java developer. We intend to further explore the usage patterns of the Java API. This includes increasing the corpus and examining further the use of infrastructure classes. We think it will also be illustrative to study how quickly additions to the API start being used. Ultimately we hope to be able to characterise the effectiveness of the API.

## References

[1] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of Java software. In W. Cook, editor, *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, U.S.A, Oct. 2006. 26/157.

[2] C. Collberg, G. Myles, and M. Stepp. An empirical study of Java bytecode programs. Technical Report TR04-11, Department of Computer Science, Univeristy of Arizona, 2004.

[3] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[4] W. B. Frakes and C. J. Fox. Sixteen questions about software reuse. *Communications of the ACM*, 38(6):75–ff., 1995.

[5] J. Y. Gil and I. Maman. Micro patterns in java code. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, pages 97–116, New York, NY, USA, 2005. ACM Press.

[6] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 241–255, New York, NY, USA, 2001. ACM Press.

[7] S. Henninger. Using iterative refinement to find reusable software. *IEEE Software*, 11(5):48–59, 1994.

[8] S. Henninger. An evolutionary approach to constructing effective software reuse repo sitories. *ACM Transactions on Software Engineering Methodology*, 6(2):111–140, 1997.

[9] S. Hunston. *Corpora in Applied Linguistics*. Cambridge University Press, 2002.

[10] L. R. Jensen. A reuse repository with automated synonym support and cluster generation. Master's thesis, Department of Computer Science at the Faculty of Science, Univ ersity of Aarhus, Denmark, 2004.

[11] D. E. Knuth. An empirical study of FORTRAN programs. *Software–Practice and Experience*, 1(2):105–133, 1971.

[12] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[13] M.-Y. Lin, R. Amor, and E. Tempero. A Java reuse repository for Eclipse using LSI. In *The Australian Software Engineering Conference*, Apr. 2006.

[14] H. Melton and E. Tempero. An empirical study of cycles among classes in Java. Technical Report UoA-SE-2006-1, Department of Computer Science, University of Auckland, 2006.

[15] H. Melton and E. Tempero. Identifying refactoring opportunities by identifying dependency cycles. In V. Estivill-Castro and G. Dobbie, editors, *Twenty-Ninth Australasian Computer Science Conference*, Hobart, Tasmania, Australia, Jan. 2006. Proceedings published as "Conferences in Research and Practice in Information Technology, Vol. 48".

[16] H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–561, June 1995.

[17] Sun Microsystems. Java se api documentation. `http://java.sun.com/reference/api/index.html`, 2006.

[18] H. Y. Yang, H. Melton, and E. Tempero. An empirical study into use of dependency injection in Java. Technical Report UoA-SE-2006-3, Department of Computer Science, University of Auckland, 2006.

[19] Y. Ye. *Supporting component-based software development with active component repository systems*. PhD thesis, University of Colorado, 2001.

[20] Y. Ye and G. Fischer. Promoting reuse with active reuse repository systems. In *6th International Conerence on Software Reuse*, pages 302–317, London, UK, 2000. Springer-Verlag.

## Appendix

This is the contents of the corpus used in this study.

**Set 1** aglets-2.0.2, ant-1.6.5, aoi-2.2, azureus-2.3.0.4, colt-1.2.0, columba-1.0, derby-10.1.1.0, drawn-vC, drawswf-1.2.9, fitjava-1.1, fitlibraryforfitnesse-20050923, galleon-1.8.0, ganttproject-1.11.1, geronimo-1.0-M5, gt2-2.2-rc3, hibernate-3.1-rc2, hsqldb-1.8.0.2, ireport-0.5.2, jag-5.0.1, jaga-1.0.b, javacc-3.2, jboss-4.0.3-SP1, jchempaint-2.0.12, jedit-4.2, jeppers-20050607, jext-5.0, jfreechart-1.0.0-rc1, jgraph-5.7.4.3, jhotdraw-6.0.1, jparse-0.96, jtopen-4.9, junit-3.8.1, lucene-1.4.3, megamek-2005.10.11, openoffice-2.0.0, pmd-3.3, rssowl-1.2, springframework-1.2.7, tomcat-5.0.28

**Set 2** antlr-2.7.6, argouml-0.20, aspectj-1.0.6, c_jdbc-2.0.2, compiere-250d, displaytag-1.1, exoportal-v1.0.2, findbugs-1.0.0-rc1, freecs-1.2.20060130, heritrix-1.8.0, hsqldb-1.8.0.4, htmlunit-1.8, infoglue-2.3Final, informa-0.6.5, itext-1.4, ivatagroupware-0.11.3, j_ftp-1.48, jfreechart-1.0.1, jrat-0.6, jspwiki-2.2.33, log4j-1.2.13, mvnforum-1.0-ga, nekohtml-0.9.5, openjms-0.7.7-alpha-3, oscache-2.3-full, proguard-3.6, quartz-1.5.2, quickserver-1.4.7, quilt-0.6-a-5, roller-2.1.1-incubating, scarab-1.0-b20, servicemix-3.0-SNAPSHOT, squirrel-sql-2.2final, struts-1.2.9, trove-1.1b5, webmail-0.7.10, xmojo-5.0.0