An Empirical Study of Unused Design Decisions in Open Source Java Software

Ewan Tempero
Department of Computer Science
The University of Auckland
New Zealand
e.tempero-at-cs.auckland.ac.nz

Abstract

A recent study on how inheritance is used in open source Java software revealed a surprising number of interfaces that were neither implemented nor extended. While innocent explanations for this exist (the interfaces are part of frameworks that only clients of the frameworks implement), it does raise the question of how much "dead code" exists in applications. Dead code usually refers to code within a function that cannot be executed, but unused interfaces, and more generally unused public methods, represent dead code at the "design" level, and so can potentially have a significant impact on future maintenance costs. This paper presents a large empirical study on existence of design decisions that are unused. This study examined 100 open source Java applications. The results show a significant level of unused design decisions.

1. Introduction

It is well-known that maintenance costs represent most of the life-time costs of software and so it is important to identify opportunities to reduce this cost. One opportunity is to avoid code that provides no useful service. Such code can cause an unnecessary maintenance burden if the maintainers are not aware it has no use. Any effort that is spent on it is essentially wasted effort [3]. Furthermore, if it is externally visible, then changes to it potentially have a large impact, so more care and effort is needed to maintain it. This suggests that it would be useful to have means to identify such code so that it can at least be marked as being of no use, or even removed. However identifying such code is in itself a cost, and if there is not in fact that much non-useful code then establishing that fact is itself wasted effort. The question that this paper tries to answer is, how much non-useful code is there?

The study described in this paper was motivated by a minor finding in a previous study on the use of inheritance in open source Java software [5]. In that study we observed a number of interfaces that were not implemented. This was surprising, since the purpose of interfaces is to be implemented. One possible explanation was that these interfaces were in fact simply not used. Another possibility was that these interfaces represented entry points to frameworks so that, while the frameworks themselves might not implement the interfaces, any clients of the frameworks would be expected to do so. In thinking about how to establish which was the actual explanation, we realised that interfaces in the second category, while they may not be *implemented*, would certainly be *used* within the framework. That is, if a framework has an interface that declared a method, then we would expect to see a call to that method somewhere in the framework. If we had interfaces for which we could find no such calls, that would indicate that those interfaces were in fact not providing a useful service.

Of course its not just interfaces that may be unused; any of the types declared in a program may not in fact be used. Furthermore, it may be that some methods (or more generally, *members*) of a given type are used, but not others. The principle described above can be applied to classes (and other Java types) and can report on just individual members. Thus, any members of any types for which no use can be found must be considered non-useful code. Given our initial motivation, our interest is primarily in members that are *externally visible*, that is, could be used from any other part of the application. Such members are visible at the design level and so must be treated with care, with careful consideration needed before they are changed or removed.

In order to begin to answer the question posed in the first paragraph, we carried out an empirical study. We examined 100 open source Java applications and identified, using very conservative criteria, members we considered to be unused. We found a surprisingly large number of such members.

The paper is organised as follows. In section 2 we summarise the related work on the issue of non-useful, or "dead" code. Section 3 then discusses in detail the criteria we use to identify unused members. We present the results in section 4 and discuss them in section 5. Finally, we present conclusions and discuss future work in section 6.

2. Background

The general problem of identifying unused code has been studied in a number of different contexts. An important topic in compiler optimisation is *dead code elimination*. Different treatments (e.g. [1]) of dead code elimination provide slightly different definitions of it, but the most general form of this deals with code within a procedure that provides no useful purpose. Examples include: code that has no effect (e.g., two writes to a location with no reads of that location between them, the first write has no effect), code that provides no functionality (e.g., a location that is never read), or code that is unreachable (e.g., a return occurs before the code). In this context, some forms of dead code are introduced by code generation, rather than introduced by the programmer, or some are disallowed by the language (e.g. Java forbids certain classes of unreachable code). However in all cases this topic refers to unusable code that is not visible in the interface of the types, and so is not where our interests lie.

The context that is perhaps the most relevant to our work is the reduction in the size of applications deployed across a network, in particular the work applied to Java by Tip et al. [6] They use various program transformation techniques to identify and remove unused members of Java types in applications. They were able to achieve a significant reduction in size of the applications they studied.

There seems to have been little work *measuring* unused code, that is, characterising the degree to which unused code actually exists, and the few empirical studies that exist relating to unused code are intended to establish the validity of a solution to something relating to unused code, rather than characterising the unused code itself.

For example, Tip et al. carried out an empirical study on 13 Java programs. As they were interested in the overall size of the application they did not directly report the unused members although this information is available from their results. They also identified all unused members, not just the externally visible ones. However their results suggest there could be a significant number of such members. Our study directly addresses this question.

Pearse and Oman looked at how certain metrics for maintainability reflect maintenance activities [3]. Of interest to us is that one of the activities they report on is removing unused code. They only reported on the number of C modules that were completely unused (that is if part of a module was unused it was not counted). They identified 29 out 820 such modules.

3. Methodology

The question we want to answer is, for every type created for the application, how much of it is used?

This is not a simple question to answer. For a given member of a Java type there are potentially four different contexts from which that member can be used. It can be used from a type in another package; it can be used from another type of the same package; it can be used from a subtype; it can be used by the type it belongs to. The visibility modifiers of Java can be used to prevent some of these uses, but for a public member all are possible. To simplify matters, we will concentrate only on public members accessed from other types, although we will allow those other types to be from both the same or different packages.

Another complication occurs due to inheritance. A method may be invoked by another type on an object of type A, but in fact that method was not declared in type A but in one of its ancestors, B say. Should we consider this to be a use of the member only on A, only or B, or both? To help clarify the discussion, we introduce the some definitions, both to make specific the terms we have already been using, and to introduce new terms.

Type A class or interface as defined by Java. (We ignore enums and annotations for this study.)

Member A non-static field or method as defined by Java. If the difference is important, we will use "method" or "field".

Member declaration The textual description of a member. The member is said to be "declared in" the type whose declaration contains the member declaration.

Member of a type A member that is available on objects of the type. Such members may have been declared in the type or inherited. Members from ancestors that are overridden do *not* belong to the type where the overriding member is declared as they cannot be directly accessed from outside the type.

External member A member whose declaration has the "public" or no (default) modifier as defined by Java.

Context of access to member The relationship of the code where the member is accessed to where the member is declared.

Unrelated access to member The context of access to the member is that the type where the access takes place is unrelated to the type where the member is declared, that is, the access is not from the type where the member is declared or from a subtype.

Protocol of a type The set of externally accessible members of a type. This borrows the Smalltalk[2] terminology to represent what can be done to objects of a type.

```
class B {                class A extends B {
  void b1(){}              void a1(){}
  void b2(){}              void b1(){}
}                        }

interface J {            interface I extends J {
  void j1();               void i1();
}                        }

class C implements I {
  void i1(){}
  void j1(){}
}
```

Figure 1. Types with differently-constructed protocols (Assume public accessibility for all members).

```
class Eg1 {
  void eg1Method(I anI) {
    anI.i1(); // I#i1(), C#i1()
    anI.j1(); // I#j1(), J#j1(), C#j1()
  }
}
```

Figure 2. Protocol use in the presence of inheritance.

```
class Eg2 {
  void eg2MethodA(A anA) {
    anA.b1(); // A#b1()
    anA.b2(); // B#b2()
    anA.a1(); // A#a1()
  }
  void eg2MethodB(B aB) {
    aB.b1(); // B#b1(), A#b1()
    aB.b2(); // B#b2()
  }
}
```

Figure 3. Use of inherited members.

Figure 1 shows several type declarations. The class B has two members, b1() and b2(), both of which have been declared in B. Class A has three members: a1(), which is declared in A, b1(), which is also declared in A, and b2(), which is declared in B and inherited by A. Because we will need to be specific where members are declared and what protocols they belong to, we will identify members with where they are declared using <type>'#'<member> (e.g. B#b1() refers to the declaration of b1() in B), and, where it is important to be specific about the protocol being referred to, we will use '['<type>']'<type>#<member> to specify what member belongs to what type (e.g. [A]B#b2() refers the member B#b2() in the protocol of A).

So for the example, the protocol of A is { [A]A#a1(), [A]A#b1(), [A]B#b2() } and the protocol for I is {[I]I#i1(), [I]J#j1()}. Note that B#b1() is *not* in A's protocol as it has been overridden by A#b1(), meaning B#b1() is not an externally accessible member of A.

Note that we do *not* include members that B inherits from java.lang.Object. As we will discuss below, we do not include external libraries in our analysis.

We can now define "using a type" as "accessing a member in the protocol of the type from an *unrelated context*", and the question of "how much a type is used" becomes "what members of the protocol are used." Before describing possible metrics to use to answer this question, we must first address complications that arise due to the nature of object-oriented design, specifically those relating to inheritance.

Inheritance provides polymorphism, and polymorphism means that what actually gets executed can only be determined at run-time. Because we want to measure code statically, we must necessarily be conservative in our measurements. As our focus is on what is *not* used, we must err on the side of considering any member that *could* be used.

For example, consider the code shown in figure 2. The method eg1Method() has two callsites where methods are invoked on an object declared to be of type I. This suggests that the protocol being used is that of I, and so we should record uses of [I]I#i1() and [I]J#j1(). However, due to inheritance, there are two other protocol's involved.

Due to the polymorphic nature of the invocation at the first callsite of eg1Method(), any subtype of I could have been passed to eg1Method(). The only subtype that exists in the example is C. It is possible that the only place the methods of C are actually executed is when instances of C are passed to eg1Method(), so if we only considered callsites in this method to apply to I we would under-represent how much of C's protocol is used. So in this case we must also consider that callsite to be a use of [C]C#i1(). For a specific situation, we might be able to determine that there is no way an instance of C would be passed to eg1Method() (e.g. through some form of dataflow analysis), but, because we can't be sure we could detect all cases, for this study we will be most conservative and always treat such callsites to be a use of [C]C#i1().

The second callsite in eg1Method() presents a different problem. The member being referred to is J#j1(). Again it is possible that the only reference made to this member is when it belongs to I's protocol, that is, there is no use of [J]J#j1() anywhere. So again not recording a use of [J]J#j1() when there is a use of [I]J#j1() would seem to

```
class Eg3 {
  void eg3Method(C aC) {
    aC.i1(); // C#i1()
    aC.j1(); // C#j1()
  }
}
```

Figure 4. Use of members implemented not inherited.

under-represent the use of J's protocol.

Note that the problem of the second callsite in figure 2 is not caused by polymorphism as was for the first callsite, but because J#j1() was inherited by I. It is not the case that members are considered used just because they look like members in a descendant type. Consider the code in figure 3. For the first callsite of eg2MethodA(), only A#b1() could be executed and so only [A]A#b1() can be considered to be used here. For a similar-looking callsite in eg2MethodB, both A#b1() and B#b1() could be accessed, so we consider both [A]A#b1() and [B]B#b1() to be used. In this case, there are two different members involved so maybe it is reasonable. However for the second callsite in eg2MethodA, only one member is involved (B#b2()), but we must still consider both [A]B#b2() and [B]B#b2() to be used, because both protocols could be available at that callsite. For the second callsite of eg2MethodB, however, [A]B#b2() can never be accessed in A's protocol.

Another issue is the distinction between **extends** and **implements** in Java. When one type **extends** another, true "inheritance" is possible — it is possible that the extending type actually gets something for free from its ancestor (in the examples, I gets J#j1() and A gets B#b2() but *not* B#b1()). However **implements** only incurs an obligation on the "inheriting" type, namely to provide the specified members. This distinction is illustrated in figure 4.

In eg3Method(C) the protocols for neither I nor J are being used. This is because, while C implements those interfaces (I directly and J indirectly), that fact is not relevant to the callsites shown. If these two interfaces were removed from the application, it would not affect eg3Method(C) at all, and so it cannot be the case that the protocols of I and J could be considered used in this case.

Another way of thinking of this is that the declarations of the methods in C required in order to implement I "override" the declarations of I. While this is not strictly true in the Java sense of "override", it does provide a consistent explanation for why both (for example) [B]B#b1() is not used in eg2MethodA and [I]I#i1() is not used in eg3Method — in both cases they have been "overridden" by declarations more specific in the context for the callsites.

The examples above indicate that a single callsite could correspond to uses of more than one protocol, and that the protocols involved could be both higher up in the inheritance hierarchy than the context specified or lower down. Protocols higher in the hierarchy are involved when they provide a member that is inherited in the context of the callsite, whereas protocols lower in the hierarchy are always involved.

3.1. Population Measured

Our empirical study as made on the contents of the Qualitas Corpus [4], a set of open source Java applications we have been using for similar studies. The version we used was released on 3 June 2008 and consists of 100 applications. The complete list, including the specific versions analysed, are available from the Qualitas Corpus website.

The measurements were performed on the bytecode representation of the applications. While this is not a true representation of the source code, it does provide a good reflection of the source code view of use of public members. If a method method(**int**) is invoked on a variable declared of type T, then the bytecode will contain both method(**int**) and T (although, as noted above, the method(**int**) may not have been declared in T). Issues relating to such things as typecasts will have already been resolved by the compiler and recorded in the bytecode.

Some applications use third-party libraries and these libraries are not always available in the corpus, so we measured only those types that were considered part of the application, that is, no third-party types are measured. Since we were not measuring "external" code, we decided to also not measure uses of the Java Standard API. There was also a practical reason for this decision as discussed below.

3.2. Practical Measurement

As noted above, we do not measure any types considered external to the application. This effects what can be measured. For example, consider the code in figure 5. It shows a class, CompareA, that has been implemented to make use of the Java Collections Framework. It implements the method CompareA#compare(A,A) as required by the Comparator interface. The method compare(A,A) is called by the implementation of java.util.Collections.sort().

If we do not measure the implementation of sort then we never see its use of compare(A,A), and so that method appears unused, which is clearly incorrect. The approach we took for this study was to assume any member that might be called from a framework that is not part of the application was actually called. Since frameworks work through type relationships described through inheritance, we identified types that might be used in this way as those types with ancestors from external types. This meant that we had

```
import java.util.*;

public class CompareA
    implements Comparator<A> {
    public int compare(A o1, A o2) {
        // omitted
    }
    public void method() {
        // omitted
    }
}

  ...
  public void sortA(List<A> list) {
    Collections.sort(list, new CompareA());
  }
```

Figure 5. Framework use of application method.

to treat any member of such classes as potentially called by a framework. So in the example, we measure the method CompareA#method() as also potentially called by the Collections Framework, even though we know that is not really the case.

The alternative was to measure all external libraries along with the applications that use them. As noted above, we did not always have such libraries. Furthermore, the size of the Standard API implementation is significantly bigger than almost all the applications. Its contribution would dominate the measurements if we included it, and even if we analysed but did not include its measurements, the time taken to measure it as well as the application was impractical. This is an issue that frequently crops up in empirical studies of Java applications and a more satisfactory solution is needed. We do, however, include jre in our study as a separate application.

A consequence for our study of these decisions was that the results we report are likely to significantly under-represent the unused code.

4. Results

We measured 100 applications, gathering the number of used and unused members measured as described in the previous section for each class and interface declared, more than 60,000 types in all. We only have space for a representative sample.

Table 1 shows one summary of the data. It reflects how many types have 2 or more members that were unused. In choosing "2 or more" we are in some sense allowing types 1 unused member to be considered the same as no unused

Application	0+1	>1	$\%>1$	Max
fitjava	11	48	81.4%	40
sablecc	80	203	71.7%	270
jruby	702	1335	65.5%	26
freecs	46	77	62.6%	13
javacc	47	78	62.4%	129
jhotdraw	105	165	61.1%	40
checkstyle	127	195	60.6%	26
ant	461	656	58.7%	30
jasml	20	26	56.5%	17
jre	3507	4225	54.6%	273
roller	258	134	34.2%	41
jpf	98	50	33.8%	30
c_jdbc	362	176	32.7%	32
eclipse	13022	6298	32.6%	422
weka	1228	579	32.0%	60
oscache	45	21	31.8%	7
xerces	533	246	31.6%	52
ganttproject	470	63	11.8%	22
jsXe	85	11	11.5%	33
squirrel_sql	1490	186	11.1%	21
colt	508	63	11.0%	17
ireport	1099	130	10.6%	40
galleon	663	64	8.8%	43
jgraphpad	269	24	8.2%	23
nekohtml	42	3	6.7%	4
rssowl	676	36	5.1%	313
jmoney	182	7	3.7%	9

Table 1. Top 10, Bottom 10, and middle 7, by proportion of types with >1 unused members. $0+1$ is number of types with 0 or 1 unused members, >1 is number with more than 1, Max is the maximum number of unused members in a type.

members, that is we consider only 1 unused member to be "not bad" and 2 or more to be "bad". This choice is somewhat arbitrary but gives us a way to summarise the data. We give another breakdown below.

For example, fitjava has 11 types with 0 or 1 unused members and 48 with 2 or more, or 81.4% of its types have 2 or more unused members. It also has one type with 40 unused members. The table shows the top 10, bottom 10, and middle 7 (eclipse has the median value of 32.6%) ordered by proportion with 2 or more unused members. There are 64 applications in the range 20%–50%.

The largest **Max** value is for poi, a rather remarkable 497, that is, there is a class or interface in poi with 497 members that are not used anywhere. The next largest is eclipse with 422. The third largest is for drjava with 377. The minimum **Max** is 4 (nekohtml, 6.7% >1) and the median is 41

App	N	Used	Used%	Not	Not%
jparse	58	49	84.5%	0	0.0%
sunflow	182	119	65.4%	9	4.9%
trove	314	190	60.5%	7	2.2%
colt	524	316	60.3%	131	25.0%
squirrel_sql	698	413	59.2%	33	4.7%
rssowl	190	110	57.9%	3	1.6%
hibernate	1202	682	56.7%	27	2.2%
ganttproject	295	161	54.6%	23	7.8%
findbugs	648	352	54.3%	40	6.2%
azureus	3263	1685	51.6%	276	8.5%
jrefactory	161	25	15.5%	5	3.1%
hsqldb	199	29	14.6%	31	15.6%
exoportal	942	130	13.8%	113	12.0%
itext	549	73	13.3%	37	6.7%
picocontainer	69	9	13.0%	4	5.8%
sablecc	261	34	13.0%	21	8.0%
ireport	199	23	11.6%	18	9.0%
freecs	102	11	10.8%	5	4.9%
jchempaint	474	45	9.5%	71	15.0%
fitjava	50	1	2.0%	0	0.0%

Table 2. Top and Bottom 10 applications by proportion of types with all members used. N is number of types measured, Used is number of types where all members are used, and Not is number of types with no members used.

App	N	B	U	U/B%
jgrapht	142	45	45	100.0%
lucene	210	9	9	100.0%
jag	96	9	9	100.0%
jrat	133	1	1	100.0%
checkstyle	274	139	139	100.0%
colt	524	3	3	100.0%
jhotdraw	222	30	30	100.0%
joggplayer	87	13	13	100.0%
jparse	58	3	3	100.0%
jFin_DateMath	47	3	3	100.0%
jena	1285	220	174	79.1%
ant	949	84	65	77.4%
gt2	2182	234	181	77.4%
c_jdbc	331	65	50	76.9%
ivatagroupware	175	13	10	76.9%
aoi	687	253	192	75.9%
freecol	347	40	30	75.0%
rssowl	190	21	6	28.6%
jruby	1927	140	35	25.0%
hibernate	1202	216	39	18.1%
nakedobjects	1276	252	36	14.3%
htmlunit	151	97	5	5.2%
jasml	44	0	0	0.0%
junit	89	0	0	0.0%
cobertura	46	0	0	0.0%
nekohtml	22	0	0	0.0%
jmoney	12	3	0	0.0%

Table 3. Proportion of "big" types with "a lot" of unused members. B is the number of "big" types, U is the number of types with more than 20 members that have more than 20% unused members.

(roller , 34% $>$ 1).

Table 1 gives a fairly simplistic view of the results. It does not, for example, take into account the sizes of individual classes. If a class has 20 members, with 2 members unused, then it will be recorded as > 1, just the same as a class with 3 measurable members and 2 unused. It's not clear which of these two cases might be considered "worse", but we would like to be able to distinguish them. Table 1 also does not give a good sense of how unused members are distributed amongst the types of an application. For example, it does not tell us how many types really have no unused members, or how many have no used members.

Table 2 shows the top 10 and bottom 10 applications by proportion of types that have all members used. It also shows, for those applications, the proportion of types with no members used. This table shows that there are applications that have a significant number of types that are completely used. Nevertheless, the 10th application, azureus, only has 52% such types, and has 276 types with no apparent use at all. Also, as this table only shows 20 out of 100 applications measured, types with all members used seems to be the exception rather than the rule.

For space reasons we will not show the results ordered by proportion of types with no members used. As might be expected from table 2, fitjava has the minimum value (despite also having the minimum for all members used). The maximum value is 29.9% for joggplayer, that is, 26 of the 87 types measured have no members used (15 types have all members used). The median is 6.9% for velocity (14 out of 203). In absolute numbers, eclipse is the largest (547/15213), javacc is the median (22/120), and fitjava the smallest. Only 31 applications have fewer than 10 types with no members used.

Table 3 shows another view of the data. In this case, we consider only "big" types, as measured by number of members, and identify what proportion of those types have "a lot" of unused members. For the table shown, we have chosen "big" to be 20 members, and "a lot" to be more than 20%. We feel types that meet this criteria have an interface for which a non-trivial proportion of it is unused. We show the top and bottom 10, and middle 7, applications. Note that

App	N	r
jfreechart	350	0.99
jre	7523	0.98
checkstyle	273	0.98
drjava	743	0.98
sablecc	260	0.97
poi	475	0.97
myfaces_core	338	0.96
jena	1284	0.96
javacc	119	0.95
fitjava	49	0.94
mvnforum	169	0.81
lucene	209	0.81
velocity	202	0.81
ant	948	0.80
azureus	3262	0.80
jmeter	414	0.79
xalan	793	0.78
proguard	340	0.57
pmd	367	0.53
jasml	43	0.51
displaytag	74	0.49
ganttproject	294	0.47
jmoney	11	0.44
cobertura	45	0.36
freecs	101	0.30
drawswf	168	0.19
nekohtml	21	0.08

Table 4. Pearson's coefficient for number of members in type against number of unused members, top 10, bottom 10, and middle 7

.

due to our choice of "big", some applications have no types that meet the criteria.

The data in table 3 suggests that big types tend to have a number of unused members. Table 4 investigates this possibility by showing the Pearson's coefficient for number of unused members per type against number of members in the type. For many applications the correlation is very strong although it is by no means universal. Whether there is a causal relationship is yet to be established. It is possible that the correlation exists simply because a type with a large number of members has more members to be forgotten about. Nevertheless this is another avenue to be explored.

5. Discussion

From the overall results given in the previous section there is a general sense that there is a lot of unused code out there. Despite a very generous interpretation of what it means for something to be "used", rather than finding there is little that is unused by that definition, it is more a case of wondering why there is all this apparently unwanted code.

Some of what we are seeing is almost certainly due to the fact that some of the "applications" are really frameworks, that is, code that by itself provides no directly useful functionality, but instead is meant to provide functionality in the development of another system. In such cases, we would expect that some of the types provided would be there solely for use by the client application and not by the framework itself.

Nevertheless we might expect only relatively few such types. If most types in a framework are meant to have some of their members used by the client, that seems to be a very complicated system to use. It would be a more usable framework if there were only a small number of client-accessible points. Looking at some frameworks it is not clear why some members are unused. For example, junit follows a common convention of naming subpackages not meant to be used externally as internal, and yet several classes in those packages have unused members.

It is worth noting that our presentation of "unused code" has been as "number of members", whereas it is unlikely that each member represents the same amount of cost of maintenance. If many of the unused members were of a trivial nature, then the number would be of less concern, however, a cursory examination indicates this is not the case.

Another point to consider is that in looking at the raw data we notice a number of unused members are in fact constructors. Constructors are always an odd case, as they are not really members in the standard sense, so perhaps we should have discounted them. However there are not so many that doing so would have changed our overall conclusions, and they do still represent a maintenance burden.

In our presentation we have not distinguished methods and fields, referring only to "members." Since we consider only externally visible members, we assume that mostly what we examine are methods, although we have not verified that assumption. One source of publicly visible fields is constants, however since we do not consider static members, we do not consider such members. So we expect that most of what we are seeing as being unused really does represent unused methods.

We could speculate as to why there is so much unused code but it would be just speculation as we do not have hard evidence to support our thinking. What is needed is an in-depth study to determine what the causes are. Our study indicates that such research could be worthwhile as identifying the causes could lead to improvements in productivity by avoiding so much unused code in the first place. One speculation we will make is that developers are generally aware that there is unused code, but the effort to remove it (making sure only unnecessary code is in fact removed) is too great. In fact IDEs such as Eclipse already report unused private members and can report what the call hierarchy is for a given method. This functionality could be exploited to provide the kinds of measurements we have used, and our study suggests that this could yield important efficiencies in software development.

5.1. Threats to Validity

The main threat to internal validity is the tool used measure the applications. The analysis it does is complex, and the entities being measured are very complex, so the possibility that cases have been missed or are incorrectly measured certainly exists. The probability of this threat was reduced through extensive automated tests and by sampling the reported unused members and manually checking that in fact there was no use of those members. Another check that was made was to measure the tool itself. As its design was very well understood, manual assessment as what was unused could be made with confidence (the results were both correct and humbling!).

Due to the conservative nature of the measuring instrument it is quite likely that the measurements reported here are considerably lower than the actual situation. As well as being generous as to what a member could be considered used, we also do not take into account call chains. If one method was only ever called by a method that we would consider unused, we would only record the latter as such, missing the fact that the former is also effectively unused.

A more sophisticated measuring instrument, perhaps based on techniques used by Tip et al. [6], would produce more accurate measurements. Had our results shown low levels of unused members, we would have needed more precise measurements, but given the high levels we have seen it seems unnecessary to be more precise.

The main threat to external validity is, as is always the case in these studies, the representativeness of the corpus. We are confident that the corpus is representative of successful open source Java applications due to its size and how it was developed. That, plus the fact that the results are much the same for all applications, gives us confidence that our conclusions apply to successful open source Java applications. Further studies are needed to determine whether our conclusions apply to commercial Java software, or software written in other languages.

6. Conclusions

We have studied the degree to which externally visible members of types in open source Java applications are unused. Our motivation for doing so is that these members represent the design of the application, and so if they are not being used then that suggests weaknesses in the design. We found that types with unused members to be very common, including types that appear to be completely unused. We believe our results under-represents the true degree of unused design decisions.

While lots of numbers is not the most exciting of results, we believe we have made two important contributions:

- We have made a large empirical study, probably the largest of its kind, on the degree of unused code. By doing we provide good evidence that there is a significant amount of unused code. We suggest that there is considerable work to be done in this area, both in refining our measurements and in characterising the cost of such code.

- We have provided another example of where actual measurement provides a more convincing picture of what is really going on with our software. While our results may not come as a surprise to some people, we can provide actual data to support our beliefs. We believe considerably more must be done in this area.

Our results justify the need for better tool support for tracking how members are used, such as for IDEs to indicate all members that are unused, not just private ones. Also, understanding the mechanisms that lead to unused members may be very beneficial to improving productivity.

Acknowledgements

This work was done while the author was a visiting researcher to the BESQ project at Blekinge Institute of Technology, Ronneby, Sweden, whose support he gratefully acknowledges.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman, and M. Lam. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[3] T. Pearse and P. Oman. Maintainability measurements on industrial source code maintenance activities. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 295, 1995.

[4] Qualitas Research Group. Qualitas corpus. release 20080603. `http://www.cs.auckland.ac.nz/~ewan/corpus/`, June 2008.

[5] E. Tempero, J. Noble, and H. Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *22nd European Conference on Object-Oriented Programming*, July 2008.

[6] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter. Practical extraction techniques for java. *ACM Trans. Program. Lang. Syst.*, 24(6):625–666, 2002.