2010 Asia Pacific Software Engineering Conference

The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies

Ewan Tempero*, Craig Anslow§, Jens Dietrich†, Ted Han*, Jing Li*,
Markus Lumpe‡, Hayden Melton*, James Noble§
*Department of Computer Science, The University of Auckland
Auckland, New Zealand. e.tempero@cs.auckland.ac.nz
† Massey University, School of Engineering and Advanced Technology
Palmerston North, New Zealand. j.b.dietrich@massey.ac.nz
‡ Faculty of Information & Communication Technologies, Swinburne University of Technology
Hawthorn, Australia. mlumpe@ict.swin.edu.au
§ School of Engineering and Computer Science, Victoria University of Wellington
Wellington, New Zealand. kjx@ecs.vuw.ac.nz

***Abstract*—In order to increase our ability to use measurement to support software development practise we need to do more analysis of code. However, empirical studies of code are expensive and their results are difficult to compare. We describe the Qualitas Corpus, a large curated collection of open source Java systems. The corpus reduces the cost of performing large empirical studies of code and supports comparison of measurements of the same artifacts. We discuss its design, organisation, and issues associated with its development.**

***Keywords*-Empirical studies; curated code corpus; experimental infrastructure**

I. Introduction

Measurement is fundamental to engineering, however its use in engineering software has been limited. While many software metrics have been proposed (e.g. [1]), few are regularly used in industry to support decision making. A key reason for this is that our understanding of the relationship between measurements we know how to make and quality attributes, such as modifiability, understandability, extensibility, reusability, and testability, that we care about is poor. This is particularly true with respect to theories regarding characteristics of software structure such as encapsulation, inheritance, coupling, and cohesion. Traditional engineering disciplines have had hundreds or thousands of years of experience of comparing measurements with quality outcomes, but central to this experience is the taking and sharing of measurements and outcomes. In contrast there have been few useful measurements of code. In this paper we describe the Qualitas Corpus, infrastructure that supports taking and sharing measurements of code artifacts.

Barriers to measuring code and understanding what the measurements mean include access to code to measure and the tools to do the measurement. The advent of open source software (OSS) has meant significantly more code is now accessible for measurement than in the past. This has led to an increase in interest in empirical studies of code. However, there is still a non trivial cost to gathering the artifacts from enough OSS projects to make a study useful. One of the main goals of the Qualitas Corpus is to substantially reduce the cost of performing large empirical studies of code.

However, just measuring code is not enough. We need models explaining the relationship between the measurements and the quality attributes, and we need experiments to validate those models. Validation does not come through a single experiment — experiments must be replicated. Replication requires at least understanding of the relationship between the artifacts used in the different experiments. In some forms of experiments, we want to use the same artifacts so as to be able to compare results in a meaningful way. This means we need to know in detail what artifacts are used in any experiment, meaning an ad hoc collection of code whose contents is unknown is not sufficient. What is needed is a *curated* collection of code artifacts. A second goal of the Qualitas Corpus is to support comparison of measurements of the same artifacts, that is, to provide a reference corpus for empirical studies of code.

The contributions of this paper are:

- We present arguments for the provision of a reference corpus of code for empirical studies of code.
- We identify the issues regarding performing replication of studies that analyse Java code.
- We describe the Qualitas Corpus, a curated collection of Java code that reduces the cost and increases the replicability of empirical studies.

The rest of the paper is organised as follows. In the next section we present the motivation for our work, which includes inspiration from the use of corpora in applied linguistics and the limited empirical studies of code that have been performed. We also discuss the use of reference collections in other areas of software engineering and in computer science, and discuss the need for a curated collection of

1530-1362/10 $26.00 © 2010 IEEE
DOI 10.1109/APSEC.2010.46

code. In section III we discuss the challenges faced when doing empirical studies of code, and from that, determine the requirements of a curated corpus. Section IV presents the details of the Qualitas Corpus, its current organisation, immediate future plans, and rationale of the decisions we have taken. Section V evaluates the Qualitas Corpus. Finally we present our conclusions in section VI.

II. Motivation and Related Work

The use of a standard collection of artifacts to support study in an area is not new, neither in general nor in software engineering. One area is that of applied linguistics, where standard corpora are the basis for much of the research being done. Hunston [2] opens her book with "*It is no exaggeration to say that corpora, and the study of corpora, have revolutionised the study of language, and of the applications of language, over the last few decades*." Ironically, it is the availability of software systems support for language corpora that has enabled this form of research, whereas researchers examining code artifacts have been slow to adopt this idea. While the goals of applied linguistics research is not exactly the same as ours, the similarities are close enough to warrant examining how corpora are used in that field. Their use of corpora is a major motivation for the Qualitas Corpus. We will discuss language corpora in more detail in section III.

A. Empirical studies of Code

To answer the question of whether a code corpus is necessary, we sample past empirical studies of code. By "empirical study of code" we mean a study in which the artifacts under investigation consist of source code, there are multiple, unrelated, artifacts, and the artifacts were developed independently of the study. This rules out, for example, studies that included the creation of the code artifacts, such as those by Briand et al. [3] or Lewis et al. [4], and studies of one system, such as that by Barry [5].

Empirical studies of code have been performed for at least four decades. As with many other things, Knuth was one of the first to carry out empirical studies to understand what code that is actually written looks like [6]. He presented a static analysis of over 400 FORTRAN programs, totalling about 250,000 cards, and dynamic analysis of about 25 programs. He chose programs that could "run to completion" from job submissions to Stanford's Computation Center, various subroutine libraries and scientific packages, contributions from IBM, and personal programs. His main motivation was compiler design, with the concern that compilers may not optimise for the typical case as no-one knew what the typical case was. The programs used were not identified.

In another early example, Chevance and Heidet studied 50 COBOL programs also looking at how language features are used [7]. The programs were also not identified and no details were given of size.

Open source software has existed for several decades, with systems such as Unix, emacs, and TEX. Their use in empirical studies is relatively recent. For example, Miller et al. [8] studied about 90 Unix applications (including emacs, TEX, LATEX, yacc) to determine how they responded to input. Frakes and Pole [9] used Unix tools as the basis for a study on methods for searching for reusable components.

During the 1990s the number of accessible systems increased, particularly those written in C++, and consequently the number of studies increased. Chidamber and Kemerer applied their metrics to two systems, one had 634 C++ classes, the other had 1459 Smalltalk classes [1]. No further information on the systems was given.

Bieman and Zhao studied inheritance in 19 C++ systems, ranging from 7 classes to 922 classes in size, with 2744 classes in total [10]. They identified the systems studied, but did not identify the versions for all systems.

Harrison et al. applied two coupling metrics to five collections of C++ code, consisting of 96, 197, 113, 61, and 12 classes respectively [11]. They identified the systems involved but not the versions studied.

Chidamber et al. studied three systems, one with 45 C++ classes, one with 27 Objective C classes, and one identifying 25 classes in design documents [12]. They were required to restrict information about the systems studied for commercial reasons.

By the end of the millennium, repositories supporting open source development such as `sourceforge`, as well as the increase in effectiveness of Internet search systems, meant a large number of systems were accessible. This affected both the number of studies done, and often their size. A representative set of examples include one with 3 fairly large Java systems [13], a study of 14 Java systems [14], and a study of 35 systems, from several languages including Java, C++, Self, and Smalltalk [15].

Two particularly large studies were by Succi et al. [16] and Collberg et al [17]. Succi et al. studied 100 Java and 100 C++ applications. The Java applications ranged from 28 to 936 classes in size (median 83.5) and the C++ applications ranged from 30 to 2520 classes (median 59). The actual applications were not identified. Collberg et al. analysed 1132 Java `jar` files collected from the Internet. According to their statistics they analyse a total of 102,688 classes and 12,188 interfaces. No information was given as to what applications were analysed.

The studies described above suggest that there is interest in doing studies that involve analysing code and the ability to do such studies has significantly advanced our knowledge about the characteristics of code structure. There are several issues with these studies however. The first is that none of these studies use the same set of systems, making it difficult to compare or combine results. Another is that because full details of the systems analysed are not provided, we are limited in our ability to replicate them. A third issue is that

it is not clear that even the authors are fully aware of what they have studied, which we discuss further below. Finally, while the authors have gone to some effort to gather the artifacts needed for their study, few others are able to benefit from that effort, meaning each new study requires duplicated effort. The Qualitas Corpus addresses these issues.

B. Infrastructure for empirical studies

Of course the use of standard collections of artifacts to support research in computer science and software engineering is not new. The use of benchmarks for various forms of performance testing and comparison is very mature. One recent example is the DaCapo benchmark suite by Blackburn et al. [18], which consists of a set of open source, real world Java applications with non-trivial memory loads. Another example of research infrastructure is the New Zealand Digital Library project, which provides the technology for the creation of digital libraries and is publicly available so that others can use it [19].

There are also some examples in Software Engineering. One is the Software-artifact Infrastructure Repository (SIR) [20]. The explicit goal of SIR is to support controlled experimentation in software testing techniques. SIR provides a curated set of artifacts, including the code, test suites, and fault data. SIR represents the kind of support the Qualitas Corpus is intended to provide. We discuss SIR's motivation in the section III.

Bajracharya et al. describe Sourcerer, which provides infrastructure to support code search [21]. At the time of publication, the Sourcerer database held 1500 real-world open source projects, a total of 254,049 Java classes, gathered from Sourceforge. Their goals are different to ours, but it does give an indication as to what is available.

Finally, we must mention the Purdue Benchmark Suite. This was described by Grothoff et al. in support of their work on confined types [22]. It consisted of 33 Java systems, 5 with more than 200 classes, and a total of 46,165 classes. At the time it was probably the largest organised collection of Java code, and was the starting point for our work.

C. The need for curation

If two studies that analyse code give conflicting reports of some phenomena, one obvious possible explanation is that the studies were applied to different samples. If the two studies claimed to be analysing the same set of systems, we might suspect error somewhere, although it could just be that the specific versions analysed were different. In fact, even if we limit our sample to be from open source Java systems, there is still room for variation even within specific versions, as we will now discuss.

In an ideal world, it would be sufficient for a researcher to just analyse what was provided on the system's download website. However, it is not that simple. Open source Java systems come in both deployable ("binary") and source versions of the code. While we are interested in analysing the source code, in some cases it is easier to analyse the binary version. However, it is frequently the case that what is distributed in the source version is not the same as what is in the binary version. The source often includes "infrastructure" code, such as that used for testing, code demonstrating aspects of the system, and code that supports the installation, building, or other management tasks of the code. Such code may not be representative of the deployed code, and so could bias the results of the study.

In some cases, this extra code can be a significant proportion of what is available. For example, `jFin_DateMath` version `R1-0.0` has 109 top-level non-test classes and 38 JUnit test classes. If the goal of a study is to characterise how inheritance is used, then the `JUnit` classes (which extend `TestCase`) could bias the result. Another example is `fitjava` version `1.1`, which has 37 top level classes, and, in addition, 22 example classes. If there are many example classes, which are typically quite simple, then they would bias the results in a study to characterise some aspect of the complexity of the system design.

Another issue is identifying the infrastructure code. Different systems organise their source code in different ways. In many cases, the source code is organised as different source directories, one for the system source, one for the test infrastructure, one for examples, and so on. However there are many other organisations. For example, `gt2` version `2.2-rc3` has nearly 90 different source directories, of which only about 40 contain source code that is distributed in binary form.

The presence of infrastructure code means that a decision has to be made as to what exactly to analyse. Without careful investigation, researchers may not even be aware that the infrastructure code exists and that a decision needs to be made. If this decision is not reported, then it impacts other researchers' ability to replicate the study. It may be possible to avoid this problem by just analysing the binary form of the system, as this can be expected to represent how the system was built. Unfortunately, some systems do include infrastructure code in the deployed form.

Another complication is third-party libraries. Since such software is usually not under the control of the developers of the system, including it in the analysis would be misleading in terms of understanding what decisions have been made by developers. Some systems include these libraries in their distribution and some do not. Also, different systems can use the same libraries. This means that third-party library use must be identified, and where appropriate, excluded from the analysis, to avoid bias due to double counting.

Identifying third-party libraries is not easy. Some systems are deployed as many archive (`jar`) files, meaning it is quite time-consuming to determine which are third-party libraries and which are not. For example, `compiere` version `250d` has 114 archive files in its distribution. Complicating the

identification of third-party libraries is the fact that some systems have such libraries packaged along with the system code, that is, the library binary code has been unpacked and then repacked with the binary system code. This means excluding library code is not just a matter of leaving out the relevant archive file.

Some systems are careful to identify what third-party systems are included in the distribution (`eclipse` for example). However usually this is in simple text document that must be processed by a human, and so some judgement is needed.

Another means to determine what to analyse might be to look at the code that appears in both source and binary form. Since there is no need for third-party source to be distributed, we might reasonably expect it would only appear in binary form. However, this is not the case. Some systems do in fact distribute what appears to be original source of third-party libraries (for example `compiere` version `250d` has a copy of the Apache Element Construction Set[1] that differs only in one class and that only by a few lines). Also, some systems provide their own implementations of some third-party libraries, further complicating what is system code and what is not.

In conclusion, to study the code from a collection of systems it is not sufficient to just analysis the downloaded code, whether it is binary or the original source. Decisions need to be made regarding exactly what is going to be analysed. If these decisions are not reported, then the results may be difficult to analyse (or even fully evaluate). If the decisions are reported, then anyone wanting to replicate the study has, as well as having to recreate the collection, the additional burden of accurately recreating the decisions.

If the collection is *curated*, that is, the contents are organised and clearly identified, then the issues described above can be more easily managed. This is the purpose of the Qualitas Corpus.

III. Designing a corpus

In discussing the need for the Software-artifact Infrastructure Repository (SIR), Do et al. identified five challenges that need to be addressed to support controlled experimentation: supporting replicability across experiments; supporting aggregation of findings; reducing the cost of controlled experiments; obtaining sample representativeness; and isolating the effects of individual factors [20]. Their conclusion was that these challenges could be addressed to one degree or other by creating a collection of relevant artifacts.

When collecting artifacts, the target of those artifacts must be kept in mind. Researchers use the artifacts in SIR to determine the effectiveness of techniques and tools for testing software, that is, the artifacts themselves are not the objects of study. Similarly, benchmarks are also a collection of artifacts where they are not the object of study, but provide input to systems whose performance is the object of study. While any collection of code may be used for a variety of purposes, our interest is in the code itself, and so we refer to our collection as a corpus.

Corpora are now commonly used in linguistics and there are many used in that area, such as the International Corpus of English [23]. The development of standard corpora for various kinds of linguistics work is an area of research in itself. Hunston says the main argument for using a corpus is that it provides a reliable guide to what language is like, more reliable than the intuition of native speakers [2, p20]. This applies to programming languages as well. While both research and trade literature contain many claims about use of programming language features, code corpora could be used to provide evidence for such claims.

Hunston lists four aspects that should be considered when designing a corpus: *size*, *content*, *representativeness*, and *permanence*. Regarding size, she makes the point that it is possible to have too much information, making it difficult to process it in any useful way, but that generally linguistics researchers will take as much data as is available. For the Qualitas Corpus, our intent is to make it as big as is practical, given our goal of supporting replication.

According to Hunston, the content of a corpus primarily depends on the purpose it used for, and there are usually questions specific to a purpose that must be addressed in the design of the corpus. However, the design of a corpus is also impacted by what is available, and pragmatic issues such as whether the corpus creators have permission from the authors and publishers to make the contents available. The primary purpose that has guided the design of the Qualitas Corpus has been to support studies involving static analysis of code. The choice of contents is due to the large number of open source Java systems that are available.

The representativeness of a corpus is important for making statements about the population it is a sample of, that is, the generalisability of any conclusions based on its study. Hunston describes a number of issues that impact the design of the corpus, but notes that the real question is how the representativeness of the corpus should be taken into account when interpreting results. The Qualitas Corpus supports this assessment by providing full details of where its entries came from, as well as metadata on such things as the domain of an entry.

Finally, Hunston notes that a corpus needs to be regularly updated in order to remain representative of the current usage, and so its design must support that.

IV. The Qualitas Corpus

The current release is 20100719. It has 100 systems, 23 systems with multiple versions, with 495 versions total. The full distribution is 9.42GiB in size, which is 32.8GiB once installed. It contains the source and binary forms of each

[1] `http://jakarta.apache.org/ecs`

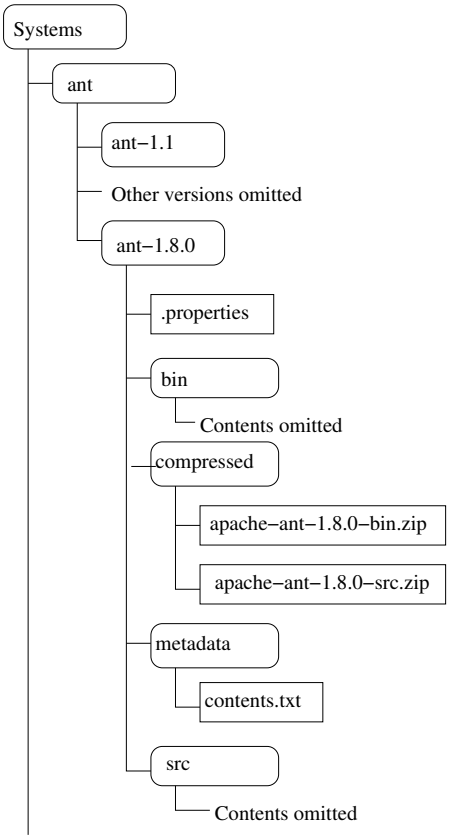

Figure 1. Organisation of Qualitas Corpus.

system version as distributed by the developers (section IV-B). The 100 systems had to meet certain criteria (section IV-C). These criteria were developed for the first external release, one consequence of which is that some systems that were considered part of the corpus previously now are not as they do not meet the criteria (section IV-I). There are questions regarding what things are in the corpus (section IV-E). The next release is scheduled for the end of October 2010 (section IV-J).

As discussed previously, the main goals for the corpus are that it reduces the costs of studies and supports replication of studies. These goals have impacted the criteria for inclusion and the corpus organisation.

A. Organisation

The corpus contains of a collection of **systems**, each of which consists of a set of **versions**. Each version consists of the original distribution (**compressed**) and two "unpacked" forms, **bin** and **src**. The unpacked forms are provided in order to reduce the costs of performing studies. The **bin** form contains the binary system as it was intended to be used, that is, Java bytecode. The **src** form contains everything in the source distribution. If the binary and source forms are distributed as a single archive file, then it is unpacked

ant antlr aoi argouml aspectJ axion azureus c_jdbc checkstyle cobertura colt columba compiere derby displaytag drawswf drjava eclipse_SDK emma exoportal findbugs fitjava fitlibraryforfitnesse freecol freecs galleon ganttproject gt2 heritrix hibernate hsqldb htmlunit informa ireport itext ivatagroupware jFin_DateMath jag james jasml jasperreports javacc jchempaint jedit jena jext jfreechart jgraph jgraphpad jgrapht jgroupsn jhotdraw jmeter jmoney joggplayer jparse jpf jrat jre jrefactory jruby jsXe jspwiki jtopen jung junit log4j lucene marauroa megamek mvnforum myfaces_core nakedobjects nekohtml openjms oscache picocontainer pmd poi pooka proguard quartz quickserver quilt roller rssowl sablecc sandmark springframework squirrel_sql struts sunflow tomcat trove velocity webmail weka xalan xerces xmojo

Figure 2. Systems in the Qualitas Corpus.

in **src** and the relevant files are copied into **bin**. There is also a **metadata** directory that contains detailed information about the contents of the version and a file `.properties` that contains information on specific attributes of the version (section IV-D).

The original distribution is provided exactly as downloaded from the system's download site. This serves several purposes. First, it means we can distribute the corpus without creating the **bin** and **src** forms, as they can be automatically created from the distributed forms, thus reducing the size of the corpus distribution. Second, it allows any user of the corpus to verify that the **bin** and **src** forms match what was distributed, or even create their own form of the corpus. Third, many distributions contain artifacts other than the code in the system, such as test and build infrastructure and so we want to keep these in case someone wishes to analyse them as well.

We use a standard naming convention to identify systems and versions. A system is identified by a string that cannot contain any occurrence of "−". A version is identified by `<system>-<versionid>`, where `<system>` is the system name, and `<versionid>` is some system-specific version identifier. Where possible, we use the names used by the original distribution. So far, the only case where we have not been able to do this is when the system name contains "−", which we typically replace with "_".

Figure 1 shows an example of the distribution for `ant`. There are 19 versions of `ant`, from `ant-1.1` to `ant-1.8.0`. The original distribution of `ant-1.8.0` consists of `apache-ant-1.8.0-bin.zip`, containing the deployable form of `ant`, which is unpacked in **bin**, and `apache-ant-1.8.0-src.zip` containing the source code, unpacked in **src**.

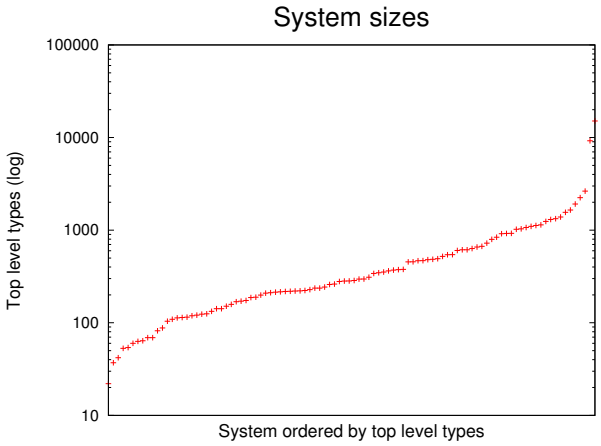

Figure 3. Distribution of sizes of systems (y is log scale).

Table I
DOMAINS REPRESENTED IN THE CORPUS.

Domain	No.
3D/graphics/media	6
IDE	4
SDK	7
database	7
diagram/visualisation	9
games	3
middleware	15
parsers/generators/make	8
programming language	2
testing	12
tool	27

B. Contents

Figure 2 lists the systems that are current represented in the corpus. Figure 3 gives an idea of how big the systems are, when listing the latest version of each system in the current release in order of number of top-level types (that is, classes, interfaces, enums, and annotations). Note that the y-axis is on a log scale. Table I shows the representativeness of the corpus in terms of domains represented and number of systems in each domain.

For the most part, the systems in the corpus are open source and so the corpus can contain their distributions, especially as what is in the corpus is exactly what was downloaded from the system download site. One exception to this is `jre`. The license agreements for the binary and source distributions appear to not allow their inclusion in the corpus. Since `jre` is an interesting system to analyse, we consider it part of the corpus however corpus users must download what they need from the Java distribution site. What is provided by the corpus for `jre` is the metadata similar to that for other systems.

C. Criteria for inclusion

Currently, the criteria for a system to be included in a release of the corpus are as follows:

1) **In the previous release** We do not want to remove things from a release that was in a previous release. This allows people to have the latest release and yet still be able to reproduce studies based on previous releases. While we intend to continue to distributed previous releases, we assume most people would prefer not to have to juggle multiple versions of the corpus.
2) **Written in Java** The choice of Java is due to both the amount of open source code available (far more than C# at the moment, although perhaps not as much as C++) and the relative ease with which it can be analysed (unlike, for example, C++). Should the opportunity arise, other languages will be added, but doing so is not a priority at the moment.
3) **Distributes both source and binary forms** One advantage with Java is that its "compiled" form is also fairly easy to analyse, easier than for the source code in fact (section IV-E), however there are slight differences between the source and binary forms. Having both forms means that analysis results from the binary form can be manually checked against the source.
   In order for it to make sense to have both source and binary forms, the binary form must really be the binary form of the source. It is expensive (in time) to download source and then compile it as every project has a different build technology (e.g. `ant`, `bat` files, uses `eclipse` infrastructure) that takes significant effort to understand. We have made the decision to simply take what is distributed by the developers, and assume that the binary form is from the source that is distributed. For this reason, we only include systems that do actually distribute both forms in a clearly identifiable way.
   This rules out, for example, systems whose source are only available through a source control system. While in theory it should be possible to extract the source relevant to a given binary release, being confident that we can extract exactly the right versions of each file is sufficiently hard that we just avoid the problem at the moment. In the future we hope to relax this, at least for systems where the relevant source version is clearly labelled.
4) **Distribute binary forms as a set of jar files** The binary form of systems included in the corpus must be bundled as `.jar` files, that is, not `.war`, `.ear`, etc, and not unbundled `.class` files. This is solely due to the expectations of our tools for managing the corpus and doing analysis using the corpus. This criterion will

probably be the first to completely go away.

5) **Available to anyone independent of the corpus** This criterion is intended to avoid ephemeral systems that crop up from time to time, or systems that are only known to us that cannot be acquired by other researchers. This allows the possibility of others to independently check the decisions we have made.
   This is the hardest one to meet, as we can not be sure when development will stop on some system. Some systems we used (and analysed) before the first external release of the corpus have suffered this fate, and so are not in the corpus. In fact we already have the situation where the version of a system we have in the corpus is now apparently no longer available, as the developers only appear to keep (or make available at least) the most recent versions. Due to criterion 1, we have chosen to keep these, even though they may not now be available to everyone.
6) **Identifiable contents** As discussed in section II-C, it is not always easy to determine what the contents of a system are. If there is uncertainty regarding the contents of a system, we do not include it.
   For example, the binary form of `netbeans` has 400+ jar files. Trying to determine what is relevant and what is not has proven to be a challenge that we are still struggling with, and so it is not in the corpus (yet).

These criteria were developed to simplify the management of the corpus. Eventually we hope some of them will be relaxed (e.g. 2 and 4) or will have less impact (e.g. 6).

D. Metadata

As part of the curation process we gather metadata about each system version, and we will continue to improve what metadata is provided (section IV-J). The corpus provides this metadata in part to resolve the issues discussed in section II-C. Ideally we would like have the exact specification as to what the developers consider to be "in" the system however it is a very time consuming process to get such information and it is not clear that even the developers would necessarily agree amongst themselves. Instead, we follow these two principles:

- Do not include something in a given system if it could also appear in some other system in the corpus. This will avoid (or at least reduce) double-counting of code measurements that are done over the entire corpus.
- Make some decision about what is in a system and *document it*. This means that even if the decision is not necessarily the best, others trying to reproduce a given analysis will know what actually was analysed.

One place where metadata is kept is in a `.properties` file (see Figure 1). This file is formatted so that it can be easily managed using `java.util.Properties`. For example, the decision we have made regarding what is identified as being in a given version of a system is recorded in the `sourcepackages` field of the `.properties` file. This is a space-separated list of prefixes of packages of Java types. Any type whose fully-qualified name has one of the listed package prefixes as a prefix of the name is considered a type that was developed for the system, and everything else is considered as being a library type. For example, for `azureus-3.0.3.4`, its `sourcepackages` value is "`org.gudy com.aelitis`", indicating that types such as `com.aelitis.azureus.core.AzureusCore` and `org.gudy.azureus2.core3.util.FileUtil` are considered part of that version of `azureus`, whereas `org.pf.file.FileUtil` (which is distributed in with `azureus`) would not.

Other metadata we keep in `.properties` includes the release date of the version, notes regarding the system and individual versions, domain information, and where the system distribution came from. The latter allows users of the corpus to check corpus contents for themselves.

The most significant development in the latest release has been the addition of significantly more metadata. We have improved the domain identification to use a more rigorous classification system (as shown in table I). We now also list, for every `.java` file in **src** and every `.class` file found in an archive in **bin**, the actual location of the file, plus information regarding how the Java type these files corresponds to is classified in the corpus.

Figure 4 shows an example of the data provided. It shows three entries for `ant-1.8.0` (out of 2786). The first and third entries show that there are both `.class` (column 2) and `.java` files (column 3) corresponding to the Java types `org.apache.tools.zip.ZipEntry` and `org.apache.tools.zip.ZipExtraField`. The middle entry, for `org.apache.tools.zip.ZipEntry`, does not have data in column 2 indicating that while there is source code for it, it is not part of the `ant` deployment. Column 4 indicates whether the entry corresponds to a type identified as being in the system (that is, matches the `sourcepackages` value), with 0 indicating it does. Column 5 provides a summary of what forms the type exists in the corpus (0 meaning it is in both **src** and **bin**, 1 for **bin** only, and 2 for **src** only). The next column indicates whether or not the entry is for a type that is considered "distributed". Such types should also occur in **bin**, so this information can be used to identify non-public types — types that are declared in files with different names. Such types would be recorded as being not distributed but in **bin**. The remaining columns show whether types are public or non-public, number of physical lines of code, and the number of non-commented non-blank lines.

The information shown in Figure 4 is provided in a tab-separated file, along with scripts that do basic analysis and which can be extended by users of the corpus.

```
...
org.[..].ZipEntry       apache-[..]/ant.jar apache-ant-1.8.0/[..]/ZipEntry.java       0 0 0 0 435 195
org.[..].ZipEntryTest                       apache-ant-1.8.0/[..]/ZipEntryTest.java   0 2 0 0 208 144
org.[..].ZipExtraField  apache-[..]/ant.jar apache-ant-1.8.0/[..]/ZipExtraField.java  0 0 0 0  85  11
...
```

Figure 4. Metadata for system version content details for `ant-1.7.1`. Some names have been elided for space.

E. Issues

Given the goal of replication of studies, the biggest challenge we have faced is clearly identifying the entities, as discussed in section II-C. There are, however, other issues we face. One is that systems change their name, such as the system that used to be called `azureus` now being called `vuze`. This creates the problem of whether the corpus entry should also change its name, meaning corpus users would have to be aware of this change when comparing studies done on different releases of the corpus, or maintaining the old name in the corpus. We have chosen the latter approach.

Another issue is what to do when systems stop being supported or otherwise become unavailable. One example of this issue is `jgraph`, which is no longer open source. Since we keep the original distribution as part of the corpus, there should be no problem with simply keeping such systems in the corpus. While we target systems we hope will be long-lived for inclusion in the corpus, we cannot guarantee that the systems will in fact continue to exist. Already there are a number of systems in the corpus that no longer appear to be actively developed (e.g., `fitjava`, `jasml`, `jparse` — see section IV-J). For now we will just note the status of such systems.

F. Content Management

Following criterion 1, a new release of the corpus contains all the versions of systems in the previous release. There are however some changes between releases. If there are errors in a previous release (e.g. missing or wrong metadata, mis-named systems or versions, problems with installation) then we will fix them, while providing enough information to allow people to determine how much the changes may affect attempts to reproduce previous studies.

We have developed processes over time to support the management of the corpus. The two main processes are for making a new entry of a version of a system into the corpus, and creating a distribution for release. In the early days, these were all manual, but now, with each new release, scripts are being developed to automate more parts of the process.

G. Distributing the Corpus

To install the copy one acquires a *distribution* for a particular *release*. The release indicates the decision point as to what is in the corpus and so is used for identification in studies (section IV-H). A given distribution of a release provides support for particular kinds of studies. For example, one distribution contains just the most recent version of each system in the corpus. For those interested in just "breadth" studies, this distribution is simpler to deal with (and much smaller to download). As the corpus grows in size we anticipate other distributions will be provided.

Releases are identified by their date of release (in ISO 8601 format). The full distribution uses the release date, whereas any other distribution will use the release date annotated to indicate which distribution it is. For example, the current release is `20100719` and the distribution containing only the most recent versions of systems is `20100719r`.

H. Using the corpus

The corpus is designed to be used in a specific way. A properly-installed distribution has the structure described in section IV-A. If every study is performed on the complete contents of a given release, using the metadata provided in the corpus to identify the contents of a system (in particular `sourcepackages`, section IV-D), then the results of those studies can be compared with good confidence that comparison is meaningful. Furthermore, what is actually studied can be described succinctly by just by indicating the release (and if necessary, particular distribution) used.

There is, however, no restriction on how the corpus can be used. It has been quite common, for example, to use a subset of its contents in studies. In such cases, in addition to identifying the release, we recommend that either what has been included be identified by listing the system versions used, or what has been left out be similarly identified. If systems not in the corpus are also used in a study, then not only do the system versions need to be identified, but some discussion regarding how the issues described in section II-C have been resolved, and, ideally, some indication as to how others can acquire the same system code distributions.

I. History

The Qualitas Corpus was initially conceived and developed by one of us (Melton) for Ph.D. research during 2005. Many of the systems were chosen because they have been used in other studies (e.g., [22], [14], [15]) although not all were still available. In its first published use (the work was done in 2005 but published later) there were 21 systems in the corpus [24].

The original corpus was used and added to by members of the University of Auckland group over the next three years, growing from 21 systems initially. It was made available for external release in January of 2008, containing 88 systems, 21 systems with multiple versions, a total of 214 entries. As noted earlier, some of the systems that were originally in the corpus and used in studies before its release did not meet the criteria used for the external distributions. By the end of 2008, there were 100 systems in the corpus. Since then, development of the corpus has focused on improving the quality of the corpus, in particular the metadata.

As the corpus has developed it has undergone some changes. The main changes have been in terms of the metadata that is maintained, however there has also been a change in terminology. Initially, the terminology used was that the corpus contained "versions" of "applications", however "application" implied something that functioned independently. This created confusion for such things as `jgraph` or `springframework`, which are not useful by themselves. We now use "versions" of "systems".

J. Future Plans

Our plans for the future of the corpus include growing it in size and representativeness (section V), making it easier to use for studies, and providing more "value add" in terms of metadata. As noted earlier, the next release is planned for late October 2010. The main goals for this release are to add new systems and to add the latest version of each of the existing systems.

One consequence of those outside the University of Auckland group using the corpus has been suggestions for systems to add. These will be the main candidates for new systems to be added. We will mainly consider large systems for this release. In the past such systems have typically been very expensive to process, however the scripts that produce the metadata described above will reduce that cost, making it easier to grow the corpus this way. This should allow us to, for example, include systems with complex structures such as `netbeans`.

Another consequence of people using the corpus is the need to perform studies different to what we originally envisaged. One example of this is that some studies need to have a complete deployable version of a system (e.g. for dynamic analysis). As we originally were only thinking of doing static analysis, we did not by default include third-party libraries in the corpus. We have now begun developing the infrastructure to provide versions that are deployable.

As there are more users of the corpus, more information (such as measurements from metrics) about the systems in the corpus is being gathered. We would like to include some of these measurements as part of the metadata in the future.

V. Discussion

The Qualitas Corpus has been in use now for 5 years, and has been made externally available for just over 2 years. There have been over 30 publications describing studies based on its use (see the website for details [25]). Increasingly, the publications are by researchers not connected to the original development group. It is in use by about 15 research groups spread across 9 countries. It is being used for Ph.D., Masters, and undergraduate research. Some of the users have started contributing to the development of the corpus, as evidenced by the author list of this paper.

Looking at how the corpus has been used, primarily it has been used to reduce the cost for developing experiments. It is difficult to determine the cost of the development of the corpus since early on it was done as an adjunct to research, rather than the main goal. However it is certainly more than 1000 hours and could easily be double that. Any user of the corpus directly benefits from this effort. Some users have in fact used the corpus merely as a starting point and added other systems of interest to them. In some cases, those other systems have been commercial systems, allowing relatively cheap comparison between commercial and open source code.

There has been less use of the ability to replicate experiments or compare results across experiments. Given that the corpus has only been available relatively recently, this is perhaps not surprising. Once other measurements and metadata become part of the corpus itself, we hope this will change.

As Do et al. note, use of infrastructure such as the Qualitas Corpus can be both of benefit and can introduce problems [20]. They note that misuse by users who have not followed directions carefully can be a problem, as we have also experienced. An example of where that can be a problem with the corpus is not using the `sourcepackages` metadata to identify system contents, meaning it is not clear which entities have being studied.

The main issue with the corpus is its representativeness. For now, it contains only open source Java systems. This issue is faced by any empirical study, but any users of the corpus must address it when discussing their results.

Hunston observes that there are limitations on the use of corpora [2]. While the points she raises (other than representativeness) do not directly relate to the Qualitas Corpus, they do raise an issue that does apply. The code in the corpus shows us what a software developer *wrote*, but what it cannot tell us is the *intent* of the developer.

VI. Conclusions

In order to increase our ability to use measurement of code to support software development practise we need to do more measurement of code in research. We have argued that this requires large, curated, corpora with which to conduct code analysis empirical studies. We have discussed the issues associated with developing such corpora and how these might impact their design.

In this paper we have presented the Qualitas Corpus, a curated collection of open-source Java systems. This corpus significantly reduces the cost of empirical studies of code by reducing the time needed to find, collect, and organise the necessary code sets to the time needed to download the corpus. The metadata provided with the corpus provides an explicit record of decisions regarding what is being studied. This means that studies conducted with the corpus are easily replicated, and the results from different kinds of studies are more likely to be able to be sensibly compared.

The Qualitas Corpus is the largest curated corpus for code analysis studies, with the current version having 495 code sets, representing 100 unique systems. The next release will significantly increase that. The corpus has been successful, in that it is now being used by groups outside its original creators, and the number and size of code analysis studies has significantly increased since it has become available. We hope that it will further encourage replication and sharing of experimental results. The corpus will continue to be expanded in content and in provision of metadata, in particular its representativeness.

References

[1] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

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

[3] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, no. 3, pp. 245 – 273, 2000.

[4] J. A. Lewis, S. M. Henry, D. G. Kafura, and R. S. Schulman, "An empirical study of the object-oriented paradigm and software reuse," in *Conference proceedings on Object-oriented programming systems, languages, and applications*, 1991, pp. 184–196.

[5] B. M. Barry, "Prototyping a real-time embedded system in smalltalk," in *Object-Oriented Programmes Languages and Systems*, oct 1989, pp. 255–265.

[6] D. E. Knuth, "An empirical study of FORTRAN programs," *Software–Practice and Experience*, vol. 1, no. 2, pp. 105–133, 1971.

[7] R. J. Chevance and T. Heidet, "Static profile and dynamic behavior of COBOL programs." *SIGPLAN Notices*, vol. 13, no. 4, pp. 44–57, apr 1978.

[8] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[9] W. Frakes and T. Pole, "An empirical study of representation methods for reusable software components," *IEEE Transactions on Software Engineering*, vol. 20, pp. 617–630, 1994.

[10] J. M. Bieman and J. X. Zhao, "Reuse through inheritance: a quantitative study of C++ software," in *Proceedings of the 1995 Symposium on Software reusability*. New York, NY, USA: ACM, 1995, pp. 47–52.

[11] R. Harrison, S. Counsell, and R. Nithi, "Coupling metrics for object-oriented design," *Software Metrics, IEEE International Symposium on*, vol. 0, p. 150, 1998.

[12] S. Chidamber, D. Darcy, and C. Kemerer, "Managerial use of metrics for object-oriented software: an exploratory analysis," *IEEE Trans. Software Engineering*, vol. 24, no. 8, pp. 629–639, Aug. 1998.

[13] R. Wheeldon and S. Counsell, "Power law distributions in class relationships," in *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM03)*, 2003.

[14] J. Y. Gil and I. Maman, "Micro patterns in Java code," in *OOPSLA '05: Proceedings of 20th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. ACM Press, 2005, pp. 97–116.

[15] A. Potanin, J. Noble, M. Frean, and R. Biddle, "Scale-free geometry in OO programs," *Commun. ACM*, vol. 48, no. 5, pp. 99–103, 2005.

[16] G. Succi, W. Pedrycz, S. Djokic, P. Zuliani, and B. Russo, "An empirical exploration of the distributions of the Chidamber and Kemerer object-oriented metrics suite," *Empirical Softw. Engg.*, vol. 10, no. 1, pp. 81–104, 2005.

[17] C. Collberg, G. Myles, and M. Stepp, "An empirical study of Java bytecode programs," *Softw. Pract. Exper.*, vol. 37, no. 6, pp. 581–641, 2007.

[18] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. B. Moss, B. Moss, A. Phansalkar, D. Stefanovi, T. VanDrunen, D. von Dincklage, , and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregan, oct 2006, pp. 169–190.

[19] I. Witten, S. Cunningham, and M. Apperley, "The New Zealand Digital Library project," *New Zealand Libraries*, vol. 48, no. 8, pp. 146–152, 1996.

[20] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Engg.*, vol. 10, no. 4, pp. 405–435, 2005.

[21] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," in *Companion To OOPSLA 2006*, 2006, pp. 681–682.

[22] 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*. New York, NY, USA: ACM Press, 2001, pp. 241–255.

[23] "International Corpus of English," http://ice-corpora.net/ice, accessed 28 May 2010, 2010.

[24] H. Melton and E. Tempero, "The CRSS metric for package design quality," in *Australasian Computer Science Conference*. Ballarat, Australia: Australian Computer Science Communications, Jan. 2007, pp. 201–210, published as CRPIT 62.

[25] Qualitas Research Group, "Qualitas Corpus Website," http://www.cs.auckland.ac.nz/~ewan/corpus, Aug. 2010.