Evaluating Usability Aspects of PJama based on Source Code Measurements

Stein Grimstad Dag I.K. Sjøberg
Department of Informatics, University of Oslo
PO box 1080, Blindern, N-0316 OSLO, Norway
{steingr,dagsj}@ifi.uio.no

Malcolm Atkinson Ray Welland
Computing Science Department, University of Glasgow
Glasgow G12 8 QQ, Scotland
{mpa,ray}@dcs.gla.ac.uk

Abstract

PJama is a system that provides orthogonal persistence defined by reachability with no changes to the Java [13] language. Introduction of persistence into the Java language is expected to give software productivity gains and reduce maintenance costs. A set of hypotheses that investigate these expectations have been defined and tested. The contribution of this paper is two-fold. First, it describes a tool that automatically measures the use of persistence in Java source code given a keywordfile specific to the technology being used. Second, we have tested a set of hypotheses applying this measurement technology. The results indicate that it is possible to have persistence with minimal changes to the source code in PJama. The results also indicate that the number of lines that explicitly use persistence does not grow with the size of the application.

1 Introduction

Developing code to create, update and read persistent data is a major task in application building. It is frequently mentioned in the persistent programming literature that typically 30% of all code in conventional languages is concerned with transferring data to and from secondary storage [8]. Persistent programming languages aim to simplify this task. PJama [2] is an experimental persistent programming system for the Java programming language. It has much in common with OO databases systems used together with Java. PJama is being developed in a collaborative project between Glasgow University and Sun Microsystems. The purpose of this project is to make PJama an efficient, orthogonally, persistent system [2].

The work addressed in this paper evaluates usability aspects of PJama. In this context, we define usability to mean how easy it is for programmers to maintain, understand and reuse the software. There are indications that usability of PJama will be good if the following two design goals, as stated in [2], for PJama are achieved. There should be no syntactic changes, negligable semantic changes and minimal requirements to use additional 'core' classes for the Java language to include persistence, and there should be a simple, almost subliminal, use of persistence for many programmers who can be satisfied by a default transaction model.

In order to evaluate usability, we have identified relationships between the respective usability attributes maintainability, understandability and reusability, and some directly measurable attributes regarding length and clustering of code. We defined a set of hypotheses on these measurable attributes. The hypotheses were tested on a collection of PJama applications.

The results indicate that it is possible to introduce persistence with almost no changes to the source code. The results also indicate that the number of lines that explicitly deal with persistence, does not grow with the size of the application. A consequence of this is the claim that the extra burden of maintenance will be insignificant and that the source code will not be harder to understand nor more difficult to reuse.

There are two motivations for our work. Hopefully the results will be useful for the further development of PJama. Another goal is to provide a means to compare PJama with competing persistent technologies for Java.

Other viewpoints than usability for evaluation of persistent systems, such as performance, tool-support and the price of the system, are beyond the scope of this paper.

The reminder of this paper is organized as follows. Section 2 describes the hypotheses. Section 3 introduces a source code measurement tool. Section 4 presents the applications investigated in our study. Section 5 discusses the results of the study. Section 6 evaluates. Section 7 suggests future work.

2 Hypotheses

People often choose software technology on the basis of their subjective opinions with no quantifiable evidence that one technology is better than another. An aim in software engineering should therefore be to provide a set of measurable criteria to evaluate different technologies.

Our work focuses on the usability aspect of PJama, that is, to what extent it is easy to develop software with higher quality and lower costs when using PJama as development environment. We divide the concept of usability into maintainability, understandability and reusability (Figure 1):

- Software maintenance is the process of changing a system after it has been delivered and is in use. Maintenance includes fixing errors, accommodation of changes to the environment and adding new functionality [12]. Maintainability indicates how much effort such changes require.

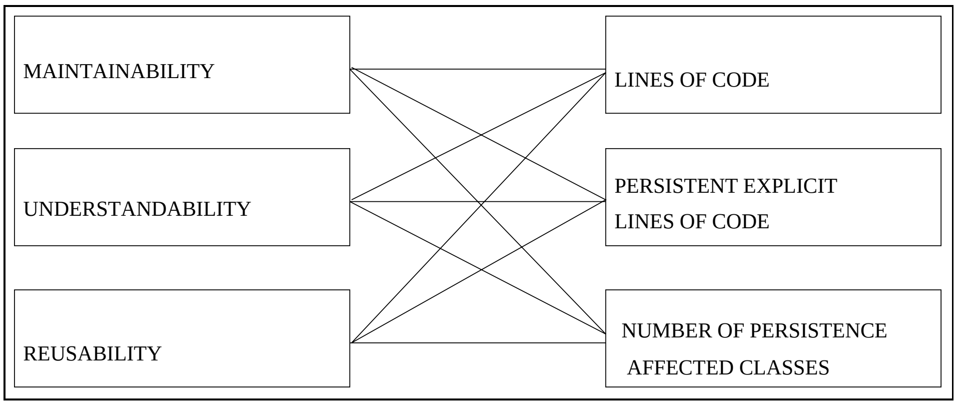
MAINTAINABILITY
UNDERSTANDABILITY
REUSABILITY
LINES OF CODE
PERSISTENT EXPLICIT LINES OF CODE
NUMBER OF PERSISTENCE AFFECTED CLASSES

Figure 1: Relationship between external and internal attributes

- Understandability indicates how easy it is to understand source code. Understandability is a difficult concept which has been subject of much investigation in the research community [14].
- Demands for lower software development and maintenance costs together with higher quality can only be met by widespread software reuse. Therefore, reusability aspects of persistent technology are important. Reuse is not limited to code, but applies to specification and design documents.

Of course, these three concepts are interrelated. For example, we would expect improved understandability to reduce maintenance costs and also aid reusability.

A major problem with maintainability, understandability and reusability are that they are *external attributes*. That is, they can only be discovered after the software has been put into use [1]. Because of their subjective nature, external attributes are not directly measurable and thus difficult to compare. The best thing we can do, is to assume that there is a relationship between external and *internal attributes*. "Internal attributes are attributes of the software products which are dependent on only the product itself." [3] Hence, the values of internal attributes are directly measurable.

Although little research with actual experiments has been carried out, there are indications that there exist relationships between the external and internal attributes as shown in Figure 1 [1, 3, 6, 9, 10, 17].

It is assumed that fewer lines of code (internal attribute) improves external attributes of an application such as maintainability. Lipow [7] reports studies that show that faults per line of code is an increasing function of the lines of code of the application. Providing the same functionality with fewer lines of code therefore implies an improvement of the software quality. Albrecht [6] found that work-effort increases with the size of an application although with great variations over different languages.

The number of lines that explicitly involve persistent code relates to understandability aspects of the technology. It is reasonable to assume that the fewer lines of persistent code there is, the easier it is to comprehend the persistent aspects of the code.

A high degree of cohesion and low degree of coupling is a well-known design principle [15] to improve understandability, reusability and maintainability. Basili [10], using the OO metrics described by Chidamber and Kemerer [9], found that highly coupled classes were more fault prone than weakly coupled classes.

On the assumptions that lower values for the three internal attributes described above leads to better values for the external attributes, we have defined three hypotheses:

1. PJama programmers provide persistence in an application with overhead that does not exceed 100 lines of code compared with a transient version of the application.[1]

2. PJama programmers use less than 20 lines of code directly using PJama statements to provide persistence in an application.

3. PJama programmers collect their persistent code in one or two classes.

The difference between hypotheses (1) and (2) is that (1) requires a version not using persistence in addition to the persistent version, since the actual overhead in a persistent version compared with a transient version of the same program is measured. Hypothesis (2) applies also to the cases where only a persistent version of the program being measured is available.

A consequence of these hypotheses is that the work-load when using persistence will not increase with the size of the application. The overhead will be small, in many cases, insignificant. The extra burden of maintenance will be insignificant and the source code will not be harder to understand nor more difficult to reuse.

In formulating the hypotheses, we made three assumptions regarding PJama code:

- Two pieces of code of the same length have the same complexity.

- The default transaction model in PJama is used. More complex transaction models will imply more effort in the management of data and thus bias the measurements.

- Most applications use a large number of classes from libraries, particularly the core classes. Generally, these require no modification in the context of PJama, but there are a few exceptions. Threads and the AWT/SwingSet classes cannot become persistent at present. However, our experience so far is that these classes are very rarely made persistent by the programmers. We therefore assume that the necessary transformations in these rare cases can be ignored in our measurements.

[1]The actual figures in these hypotheses were chosen on the basis of a pilot study of PJama programs.

loc	Lines of code
ploc	Persistence affected lines of code
pirol	Persistence impact ratio on lines of code ($pirol = ploc/loc$)
noc	Number of classes
nopc	Number of persistence affected classes
piroc	Persistence impact ratio on classes ($piroc = nopc/noc$)
adploc	Average distance between persistence affected lines of code
pibploc	Length of the interval between the first and last keyword

Table 1: PCMT metrics

3 Measurement Technology

Manual data collection is expensive, unreliable and tedious. We thus built a measurement tool, the Persistent Code Measurement Tool (PCMT). The metrics produced by PCMT are shown in Table 1. PCMT measures the proportion of lines and classes that explicitly include persistent code in Java programs. The persistent keywords are loaded from a keyword file. Hence, PCMT can easily be adapted to new tools/versions for providing persistence, and also other program characteristics, by generating new keyword files.

PCMT parses Java source code and tracks the use of identifiers from the package(s) under investigation and of identifiers refering to instances of classes from those packages.

The total size of the application is measured in lines of code (*loc*). Lines are defined as productions rather than line-shifts to avoid some of the impact caused by personal programming styles. Comments are ignored.

Persistent affected lines of code (*ploc*) are also counted. A line is counted as persistent affected if at least one class/interface from the keyword file is used. For example, if *PJStore* is a word listed in the keyword file, the following line will be counted as a persistent line of code:

```
PJStore ps = PJStoreImpl.getStore();
```

Lines where a variable introduced by such a declaration is used, e.g.,

```
manyana = (ToDoList)(ps.getPRoot("ToDoList"));
```

will also be counted as a persistent line of code since *ps* is a variable of type *PJStore*, which (in this case) is a persistent keyword.

PCMT also reports the persistent impact ratio on lines of code (*pirol*), which is calculated by dividing the *ploc* metrics by the corresponding *loc* metrics.

Some statistics on class level are collected. The number of classes (*noc*) is counted. Internal classes are considered to belong to the outer class and are thus not counted as classes.

A class is considered to be persistence affected if it includes persistent affected code (*ploc*). The number of persistence affected classes (*nopc*) is the number of classes (*noc*) that have such code.

```
Application Name:       FamilyTreeDB
Parsed:                 10-Jan-98
Author:                 lisefr
Persistent technology:  sql

-----------------------------------------------------------------------
Class Name                 Loc      Ploc     Pirol    Adploc   Pibploc
-----------------------------------------------------------------------
ChildrenCanvas             13       0         0.0%    0            0%
ConnectDatabaseWindow      44       0         0.0%    0            0%
Woman                      41       0         0.0%    0            0%
IntElem                    4        0         0.0%    0            0%
FamilyTreeAppl             261      19        7.2%    11          75%
Man                        41       0         0.0%    0            0%
NewPersonWindow            118      0         0.0%    0            0%
Person                     149      3         2.0%    5            7%
SListEl                    11       0         0.0%    0            0%
FamilyTreeReg              254      2         0.7%    31          12%
FamilyTreeCanvas           49       0         0.0%    0            0%
-----------------------------------------------------------------------
Total                      985      24        0.0%
=======================================================================
Number of Classes                      : 11
Number of Persistence Affected Classes : 3
Persistence Impact Ratio on Classes    : 27.2%
=======================================================================
```

Figure 2: Example of output from PCMT

The persistent impact ratio on classes (*piroc*) resembles the *pirol* metric. It is calculated by dividing the *noc* metrics by the *nopc* metrics.

Some measurements on distribution within a class are also collected. PCMT captures the average distance between the persistent affected lines of code (*adploc*) and the interval between the first and last persistent affected line (*pibploc*). The *pibploc* metric is reported as the proportion of the total number of lines.

The default of PCMT is to investigate the transitive closure of classes (source files) reachable from a persistent root. An example output from PCMT is shown in Figure 2.

3.1 Implementation

PCMT is built using JavaCC [4] and uses the Java grammar given there almost unchanged. One production is added to the import-statements to capture all legal statements. The original parser reported an error on an empty statement within imports— a construct that is accepted by the Java compilers. Instead of altering the grammar in order to collect the statistics, we have added actions on the different productions. These actions include methods to investigate whether an identifier is a persistent identifier, methods to keep track of the scope of the variables and a few other methods.

3.2 Strategy for Line Counting

A line is defined as a compiler directive, a declaration or an executable ending with a ";". Class and method headers are counted as lines too. Nested classes *loc/ploc* are counted as loc/ploc of the outer class and not as individual classes.

The options -bt, -bw and -bf of PCMT enable all statements within *try*-, *while*- and *for*-blocks to be counted as persistent lines if at least one of the lines includes a persistent keyword. This might help capture some of the cases where code has to be changed due to lack of orthogonality. However, these options should be used with care, as there is no guarantee that the block does not include other functionality than the persistence transformation.

A problem with PCMT is that it only captures explicitly persistent code. Ideally, we would like to include the cases where a class has to be changed due to lack of orthogonality. For example, suppose we were to analyse programs using PSE [23] (ObjectStore's object database architecture). `Vectors` in PSE for Java cannot be persistent, which means that an alternative storage structure has to be used. Such re-writing decreases the quality of the source code, since the application will have more lines of code, and require structural changes in some cases (e.g., new classes added).

4 Investigated Applications

This section describes the applications on which we tested the hypotheses. The applications were developed at the universities of Glasgow, Oslo and St Andrews, and are described below.

PJPresent is an application written for presentations. Information such as technical reports are specified in a text file with certain tags and a slide-like presentation is built. Different styles, fonts, backgrounds, etc. can be chosen for the presentation. The application was designed with PJama in mind.

ZEST is a system for modeling and automatically generating code for distributed systems [16]. It consists of different components such as parser, data model editor, validator, code generator, etc.

GAP is an acronym for Geographic Application of Persistence.

OCB is an object and class browser written entirely in Java. The program is available at [22].

Hyper-Program System consists of object browser and editor. Both are implemented using PJama.

Variadic Generity is example code for testing variadic generity. A detailed presentation of the code can be found in [21].

MountainClimbing is an application for registering people and the mountains they have climbed.

FamilyTree keeps track of people's family trees. You can register people and their relatives and get information about family relationships.

ProjectWorkflow is a beta version of an application for distributed project management, based on a client/server architecture. A server keeps track of the different tasks and associated documents. The clients request tasks and documents from the server. Persistence is only applied on the server side of the program.

Persistent RMI Demo The first step in implementing support for distribution in PJama is the porting of RMI to a persistent context. A first implementation of persistent RMI (PJRMI) [18] has been produced. The `Persistent RMI Demo` is a program that exploits this technology. It has support for persistence added for both the server and the client. The `Persistent RMI Demo` consists of four programs that can be run independently: persistent client, "shut-down" persistent client, persistent server and "shut-down" persistent server.

The applications are divided into four groups. The Java and PJama experience of the programmers is described as one of the following:

- **experienced** — Java or PJama has been part of the their daily working environment for at least half a year.
- **moderately experienced** — prior experience with Java or PJama, but not part of the daily working environment.
- **inexperienced** — the developed application was their first application ever in Java or PJama.

`PJPresent`, `ZEST`, `GAP`, `OCB` and `Hyper-Program System` are large applications written by experienced programmers. `Variadic Generity` is a collection of programs also written by experienced programmers. We therefore assume that their code demonstrates smart use of the PJama technology.

`MountainClimbing` is an experiment where six teams of students implemented an application based on the same specification. Each team consisted of 3–4 fourth year students. The programmers were inexperienced.

`FamilyTree` was implemented in three versions using three different persistent technologies: The same, inexperienced programmer wrote all three implementations.

name	*technology*	*loc*	*ploc*	*noc*	*nopc*
PJPresent	PJama	4804	6	58	1
PJPresent	transient Java	4779	0	57	0
ZEST	PJama	5728	13	150	2
ZEST	transient Java	5654	0	148	0
GAP	PJama	3938	39	60	5
OCB	Pjama	2511	6	65	1
Hyper-Program System Browser	PJama	3395	8	62	2
Hyper-Program System Editor	PJama	4585	9	19	3
Variadic Generity (reflective)	PJama	625	13	12	3
Variadic Generity (generic)	PJama	379	13	16	3
Variadic Generity (non-generic)	PJama	478	13	21	3
Mountains-group1	PJama	1249	100	32	13
Mountains-group2	PJama	801	87	18	15
Mountains-group3	PJama	913	13	20	3
Mountains-group4	PJama	398	25	11	4
Mountains-group5	PJama	1247	87	21	14
Mountains-group6	PJama	997	90	20	16
FamilyTree	Java.io	822	48	10	2
FamilyTree	PJama	871	28	11	2
FamilyTree	JDBC	985	24	11	3
ProjectWorkflow	Java.io	510	96 (48)	17	11(3)
ProjectWorkflow	PJama	419	16	17	1
Persistent RMI demo	PJama	124	17	9	4

Table 2: Measurements of the applications

`ProjectWorkFlow` and the `Persistent RMI Demo` differ from the other programs measured as they are not "real-world" applications, but demonstration programs to test distributed technology. For example, user interfaces have not been properly implemented in these programs. `Project Workflow` was developed by a moderately experienced programmer, while `the Persistent RMI Demo` was developed by an experienced programmer.

5 Results

This section describes the testing of the hypotheses (Section 2), which is based on measurements resulting from applying PCMT on the applications described in the previous section (Table 2).

5.1 Hypothesis 1: extra overhead in lines of code

PJama programmers provide persistence in an application with overhead that does not exceed 100 lines of code compared with a transient version of the application.

The metric relevant to this hypothesis is *loc*. The hypothesis applies to only those applications that have both a transient and a persistent implementation. `PJPresent` and `ZEST` are the only two applications that meet this requirement. `FamilyTree` does not have a transient version, but the application had file management as part of its system requirements. The consequence is that persistence gained with PJama and JDBC is added on top of the Java.io version with almost no code from the Java.io version removed. Therefore, the Java.io version of `FamilyTree` can be viewed as a transient version in this context.

`PJPresent` has an overhead of 25 *loc*, `ZEST` 74 *loc* and `FamilyTree` 30 *loc*. These results support our hypothesis that an overhead of less than 100 *loc* is needed for providing persistence using PJama. These results also indicate that the absolute number of overhead-loc are independent of the total application size.

5.2 Hypothesis 2: persistent explicit lines of code

PJama programmers use less than 20 lines of code directly using PJama statements to provide persistence in an application.

The metric used to test this hypothesis is *ploc*. `PJPresent` has 6 *ploc*; `ZEST` 17 *ploc*; `OCB` 6 *ploc*; `Hyper-Program System Browser` and `Hyper-Program System Editor` have 8 and 9 *ploc*; all the `Variadic Generity` programs 13 *ploc*. Only one of the `MountainClimbing` applications has *ploc* as expected, even though another is close (25 *ploc*). The remaining four have significantly more (87-100 *ploc*). This differs greatly from our hypothesis. `FamilyTree` has 28 *ploc*, which also exceed the limit specified in the hypothesis. However, here code for creation of a store is included as part of the application. Both `ProjectWorkFlow demo` and `Persistent RMI Demo` have results as expected (respectively 16 and 17 *ploc*). `GAP` is the only large application that has more *ploc* than expected: 39 *ploc*. We received the GAP results just before the deadline for this paper, so we have at present no explanation why these numbers are higher than expected.

We have investigated the anomalous version of `MountainClimbing` by manually analyzing the code. It seems like persistence is included in places where transitive closure from a persistent root would have been sufficient. This may be a result of inadequate understanding of the technology. This is not totally unexpected. Similar results were found in a study of applications written in Napier88 [19]. There were significant differences between novices and experienced programmers in the way they designed their applications and their use of the languages constructs [11]. A common factor that also may explain why these applications conflict with the hypothesis is that quite a bit of test code is included (i.e., dumping all classes reachable from a root to a screen) and that code for creation of a store is included as part of the application.

The applications developed by *experienced* and *moderately experienced* programmers support our hypothesis with the exception of `GAP`, while most of the applications by *inexperienced* programmers conflict with the hypothesis. The results also suggest that the number might be a bit higher than 20 *ploc* when the creation of a persistent store is included as part of the application.

5.3 Hypothesis 3: classes including persistent code

PJama programmers collect their persistent code into one or two classes.

This hypothesis is tested using the *nopc* metric. `PJPresent` and `OCB` have one class containing persistent code, while `ZEST` and `Hyper-Program System Object Browser` have two. Hence, they all support the hypothesis. `ZEST` has new functionality added to the persistent version such as a save option. If new functionality had not been added, only one class would have been needed to provide persistence.

`Hyper-Program System Editor` and `Variadic Generity` both have three classes containing persistent code; `GAP` has five. These programs thus conflict slightly with the hypothesis.

All of the `MountainClimbing` applications conflict with this hypothesis. There are different degrees of conflicts, though. While two of them only just exceed the hypothesis (3-4 *nopc*), the four remaining have persistent affected code in almost all the classes (13–16 *nopc*). As discussed for hypothesis two, the probable reason for this is lack of understanding of how to design properly.

`FamilyTree` has two *nopc*. They include code to create a persistent store. `ProjectWorkflow demo` has one *nopc*; `Persistent RMI Demo` has four *nopc*. However, as discussed in the description of the applications, `Persistent RMI Demo` consists of four programs that can be run independently. Hence, all of them have to connect to a store, and therefore do not conflict with our hypothesis.

As for hypothesis 2, we observe that the investigated applications support the hypothesis with the exception of most of the applications developed by *inexperienced* programmers, and the `GAP` application.

6 Evaluation

During our study we encountered problems common to many aspects of software engineering research. Some of the aspects that might have biased the measurements are:

1. Individual programmer influence. Different programming styles and ineffective use of the technology probably affect the results.

2. Reuse of code. Use of local libraries might result in the same code being measured several times.

3. The application domain. The suitability of various persistence technologies may depend on the application domain.

4. Services supported by the application such as restore functions, versioning and advanced transaction models. In our study, we have assumed that the programmers use the default transaction model.

We are aware that other internal attributes than those related to size and clustering have an effect on the the external attributes we investigate. For example, methods per class directly influence maintainability [10]. However, size is often the key factor in models for evaluating cost, productivity and effort [1]. As discussed in Section 2, it is also reasonable to assume that clustering of the persistent code will improve the external attributes we investigate.

PCMT measures more attributes than we ended up using in our final set of hypotheses. We were unable to find use of the attributes concerning internal distribution of persistent code on class level (*pirol, piroc, adploc, pibploc*). It seems like most classes using persistence have the code scattered all over the class.

We have measured a small set of applications, and there are many uncertainties regarding the relationship between internal and external attributes. Even though we have confirmed our hypotheses, we still cannot confirm or reject whether this means that the usability aspects of PJama are good. We can only say that if the technology gives high scores on the investigated internal attributes, it is more likely that we have high scores on the related external attributes.

To provide more reliable results, we would like to measure the same application written with several different technologies by people who have experience with those technologies, have the same application domain experience and generally have the same programming skills. However, such an experiment is not realistic because of limitations regarding money, time, available people, etc.

7 Conclusions and Future Work

Our work is a first attempt to collect measurements of attributes to evaluate usability aspects of PJama. In order to evaluate usability, we identified relationships between the respective usability attributes maintainability, understandability and reusability, and some directly measurable attributes regarding length and clustering of code. This research showed that little extra effort is required from the programmer to provide persistence in applications.

The total overhead in lines of code of an application when introducing persistence with PJama is small (typically less than 100 *loc*). Of these, typically less than 20 were lines explicitly using the persistent technology. The numbers are relatively constant, therefore the proportions will be comparably smaller for larger applications. This indicates that the introduction of persistence by PJama does not adversely affect the usability aspects of the application code.

When using PJama, the code for providing persistence is in most cases kept within a few classes, typically 1 or 2. As for the lines of code figures, this number does not increase with the size of the applications. The result of this, is that maintainability, understandability and reusability aspects of an application do not deteriorate when persistence is introduced using PJama.

To provide data-collection automatically, we developed a general purpose measurement tool that also can exploit other programming constructs such as distribution and user-interfaces to help identify critical issues for software development. Optimizing one area might lead to worse results in other areas, and possibly less good overall results.

We investigated a number of different applications written by people with different backgrounds, and tried to extract trends from the results we collected. However, inexperienced programmers are still inexperienced programmers and our results suggest that there is a need for better introductory and tutorial material, and more extensive documentation about the PJama technology.

The results from the applications that have implementations using other persistent technologies than PJama, suggest that PJama has less overhead and changes to structure, and therefore has advantages in comparison. However, considerably more measurements are needed before we are able to collect sufficient reliable results to make strong claims.

A next step would be to investigate how other proposed OO metrics such as depth of inheritance trees and weighted methods per class [9] are affected by different persistent technologies. Such metrics have been shown useful in detecting error-prone classes [10].

As the cost of hardware resources fall and the cost of software staff rises, it is naturally prudent to invest computing resources in making software development and maintenance less labour intensive and less error prone.[2] The provision of orthogonal persistence has long been justified on these grounds [20]. A valid engineering approach requires a means of measuring whether the intended goal (less costly development and maintenance) is being met. This is particularly important for Java where many mechanisms for persistence are being developed. The only valid way to choose between these and to direct development is to use relevant and comparative measurements.

We are not particularly satisfied with the preliminary measurement technology we report here. Its primary merit is that it exists and it is cheap to use. We hope that its use will lead to better metrics and to comparisons between alternative technologies that have a relevant and testable basis.

When we consider the large investments being made by companies to develop persistence technologies for Java and the even larger costs of projects using these technologies, we believe it is very suprising that there is not a well developed strategy for comparing programming costs. Whilst performance remains important, the enthusiasm for benchmarks and the dearth of programmability measurements, may already be unballanced and becoming inappropriate.

Acknowledgements

We would like to thank the people in the RAPIDS group and the senior honour database students at Glasgow University, Lise Frengen, John Hagemeijster and Graham Kirby for providing us with measurements of their source code. We would particularly like to thank Susan Spence, Tony Printezis and Cathy Waite

[2]Bill Gates identified the unification of the plethora of stores (file systems, none registries, databases, etc.) on our computers and improved relationships between databases and programming languages as an urgent research priority [24].

for their helpful support as the work proceeded. The work is supported by the Pastel working group (EP 22552) funded by the European Community.

References

[1] Sommerville I. *Software Engineering.* Fifth edition, Addison-Wesley, 1996.

[2] Atkinson M, Jordan M, Daynes L, Spence S. Design Issues for Persistent Java: a Type-safe, Object-oriented, Orthogonally Persistent System. Seventh International Workshop on Persistent Object Systems (POS7) Cape May, New Jersey, May 1996.

[3] Fenton N. *Software Metrics.* First edition, Chapman & Hall, 1991.

[4] Sun Microsystems, Inc. The Java Compiler Compiler.

[5] Hamilton, Cattel. JDBC: A Java SQLAPI. http://splash.javasoft.com/jdbc, 1996.

[6] Albrecht AJ, Gaggney JE. Software Function, Source Lines of Code, and Development Effort Prediction: a Software Science Validation. *IEEE Trans Software Engineering* SE-9(6), pp. 639–648, 1983.

[7] Lipow M. Number of Faults Per Line of Code. *IEEE Trans Software Engineering*, SE-8(4), pp. 437–439, 1982.

[8] King F. IBM Report on the Contents of a Sample of Programs Surveyed. IBM Research Centre, San Jose, California, 1978.

[9] Chidamber R, Kemerer R. A Metric Suite for Object Oriented Design. *IEEE Trans Software Engineering* 20(6), pp. 476–493, 1994.

[10] Basili VR, Briand LC, Melo WL. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Trans Software Engineering* SE-22(10), pp. 751–761 1996.

[11] Sjøberg DIK, Cutts Q, Welland R, Atkinson M. Analysing Persistent Language Applications. Sixth International Workshop on Persistent Object Systems, pp. 235–255, Workshops in Computing, Springer-Verlag and British Computer Society, 1995.

[12] Swanson EB. The Dimension of Maintainance. Proceedings of the Second International Conference on Software Engineering, pp. 492–497, 1976.

[13] Gosling J, Joy B, Steel G. *The Java Language Specification*, Addison-Wesley, 1996.

[14] von Mayrhauser A, Vans AM. Comprehension Processes During Large Scale Maintenance. 16th International Conference on Software Engineering, pp. 39–49, 1994.

[15] Constantine LL, Yourdon E. *Structured Design.* Englewood Cliffs, N.J. Prentice-Hall, 1979.

[16] Waite C, Welland R, Atkinson M.P. Supporting Software Evolution in ZEST. Technical Report TR-1997-29, Department of Computing Science, University of Glasgow, December 1997.

[17] Banker RD, Datar SM, Kermerer CF, Zweig D. Software Complexity and Maintenance Costs. *Communications of the ACM* 36(11), pp. 81–94, 1993.

[18] Spence S. Persistent RMI. More information at http://www.sunlabs.com/research/forest/opj.tutorial.pjrmidoc.html

[19] Morrison R, Brown F, Connor R, Dearle A. The Napier88 Reference Manual. Technical Report PPRR-77-89, Universities of Glasgow and St Andrews, 1989.

[20] Atkinson MP, Bailey PJ, Chisholm KJ, Cockshott WP, Morrison R. An Approach to Persistent Programming. *The Computer Journal* 26(4), pp. 360–365, 1983.

[21] Kirby G, Morrison R. Variadic Genericity Through Lingustic Reflection: A Performance Assesment. To appear at POS8, 1998.

[22] OCB - Object Class Browser. Available for download at http://www-ppg.dcs.st-and.ac.uk/Java/ocb/

[23] PSE for Java from ODI. Available for download at http://www.odi.com

[24] Gates B. SIGMOD Opening and Plenary Keynote. ACM SIGMOD/PODS Seattle, june 1998.