Exploring the Differing Usages of Programming Language Features in Systems Developed in C++ and Java

Michael English[1] and Patrick McCreanor[2]

[1] Lero, Computer Science and Information Systems Department, University of Limerick
Michael.English@ul.ie
[2] Computer Science and Information Systems Department, University of Limerick

Abstract. The Object-Oriented (OO) programming paradigm has claimed numerous advantages, including enhanced understandability, maintainability and reusability of OO software. However, these advantages do not automatically apply when an OO approach to software development is adopted. The features provided in OO languages such as C++ and Java can help to facilitate the development of understandable, maintainable and reusable software. However, systems built from different languages may have structural differences which may in turn impact the understandability, maintainability or reusability of the systems. In this paper an empirical study is presented which utilises software metrics to examine the structure of software systems written in C++ and Java. The findings suggest that C++ systems may be more difficult to understand or maintain than Java systems and that information hiding principles are applied only to a limited extent in the development of OO software.

1 Introduction

Java and C++ are two of the most popular (Object-Oriented) programming languages. These languages have been adopted as the development languages of many software applications as demonstrated on the open-source software repository sourceforge.net. Many comparisons of the languages and the features they offer exist, e.g. [14]. However, there has been limited empirical analysis of how these features are utilised in the development of software applications. In this paper, this gap in the literature is addressed by utilising software metrics to extract information about the structure of twenty software applications written in C++ and Java and using this information to determine if the applications analysed satisfy recommendations regarding good OO design and development.

The analytic approach adopted in this paper offers a number of potential benefits:

- It can provide insights on how the language features of C++ and Java support the design and development of applications using the Object-Oriented (OO) paradigm
- It can be useful to help identify possible characteristics of the source code of software systems that may provide a bottleneck in the understanding and maintenance of systems.
- It can provide evidence in support of the application of best practices in OO design and development.
- It can provide guidelines to professional software engineers, software engineering teachers and students of software engineering on the appropriate use of language features in software development.

The paper is structured as follows. The next section discusses the related work. Section 3 discusses the empirical study, including the specific research questions addressed, the data analysis and interpretation of results. Finally, some conclusions and future work are described in Section 4.

2 Related Work

Comparisons of the C++ and Java programming languages are common, e.g. [14, 11, 20, 12]. These papers tend to focus on the features provided by the languages. For example, abstraction, inheritance, multiple inheritance, memory management, threads, operator overloading,

templates, robustness and portability are topics that have received much attention in these papers and on numerous web-based discussion forums. However, there has been less focus on an empirical comparison of systems developed using the languages, that is, a comparison of how the features of the languages are used in the practice of software development. This section reviews some existing work in this area and describes the software metrics that are used as part of this study.

2.1 Comparison of C++ and Java Software Systems

Mayrand *et al.* [15] presented an assessment framework to evaluate the quality of object-oriented systems at three levels of granularity; system, class and method. Nine software systems (Eight Java systems and one C++ system) were evaluated in the study. The differences in the organisation of the software from a module perspective between the two languages were highlighted. For example, C++ depends on the file structure to provide modular support whereas Java offers modular support through the package facility. The system level metrics extracted are based around the use of inheritance. The class metrics focus on the use of the visibility qualifiers: public, protected and private, and the number of methods and attributes defined for a specific visibility qualifier. Finally, the method level metrics reflect the number of calls to and from methods and the number of statements in methods. Their findings suggest that the C++ system had the largest depth of inheritance at a level of 9 indicating 'over engineering' of the inheritance hierarchies. The C++ system had a very low proportion of public attributes, suggesting that the classes in the system are well encapsulated. In this paper we extend the work of Mayrand *et al.* by considering a larger cohort of systems utilising size, inheritance, coupling and cohesion metrics.

Phipps [19] compared Java and C++ from the perspective of bug, defect and productivity rates. The author found that C++ had more bugs and defects per line of code and that Java was more productive in terms of lines of code per minute. However, the empirical study involved one programmer developing a C++ and Java system and therefore requires further investigation to support or refute these findings. Prechelt [21] compared C++ and Java from an efficiency perspective and also compared the performance differences between programmers. The study used 40 implementations of the same program by 38 different programmers. The findings suggest that performance differences between programmers are much larger than the efficiency differences between languages. The author highlights the importance of good program design.

Counsell and colleagues have performed a number of studies evaluating the structure and design of object-oriented systems and their use of object-oriented facilities such as inheritance, encapsulation and the friend mechanism of C++ [4, 5, 3, 6]. In these studies, metrics are employed to facilitate the analysis and interpretation of a number of hypotheses.

One such study considered the relationship between inheritance and encapsulation by comparing the use of the protected and private language facilities, in classes which are engaged in inheritance and which are not engaged in inheritance [3]. Results indicated that stand-alone classes make considerable use of the protected mechanism, even though this feature is only useful for classes engaged in inheritance. Two possible reasons for this are suggested. The first is that these classes were removed from the inheritance hierarchy without the protected members being redefined. This may occur if the classes in question were key classes in the system and were removed from the hierarchy to allow easier access to their members [3]. The second possible reason for protected methods to appear in stand-alone classes is that it may have been anticipated that classes were designed with the intention of being part of an inheritance hierarchy at some future point. Other studies by Counsell and colleagues compared the use of the coupling mechanisms (aggregation, association and generalisation) with the number of methods and attributes in a class [5, 6].

The study described in this paper focuses on the use of the language features by programmers in developing software systems in C++ and Java. A deeper insight on why and how programmers use specific language constructs can help inform studies (like those described above) that assess external quality characteristics and help to identify cause-effect relationships between internal and external characteristics of software.

2.2 Software Metrics

Numerous software metrics have been defined for the Object-oriented paradigm. In addition, a number of tools to extract metrics from software systems have been developed. One such tool, Understand, [23], was utilised in this work. The metrics extracted are utilised to compare the structure of systems developed in C++ and Java.

The metrics extracted from the software systems were:

- Weighted Methods per Class (WMC): This is the sum of the weights of all the methods in a class. If the weight assigned to each method is just one then WMC is the number of methods in the class (NMC) [4]. NMC is the metric adopted in this study.
- Number of Public Methods per Class (CMPub): This is the number of public methods in a class [13]. The public methods in a class define the interface to the class and thus the services that are available to other classes. This metric is a good measure of "the amount of responsibility in the class" [13].
- Number of Protected Methods per Class (CMPro): This is the number of protected methods in a class. Protected methods can be inherited by descendant classes in the inheritance hierarchy. In C++ the inheritance must be either public or protected for protected methods to be inherited to descendant classes. If private inheritance is used, methods are inherited to the subclass but are not inherited further down the hierarchy. For classes outside the inheritance hierarchy, protected methods cannot be accessed.
- Number of Private Methods per Class (CMPri): This is the number of private methods in a class. Private methods can only be called from other methods in the class (with the exception of the use of friends in C++). The private and protected methods in a class are "those the class uses to get its work done to honour the public services it has made available" [13].
- Depth of Inheritance Tree (DIT): The DIT of a class is the maximum distance in the inheritance tree of the class from the root node of the hierarchy [2].
- Number of Children (NOC): The number of children (directly derived classes) of a class is the number of classes inheriting directly from the class. If a class has a large number of children this suggests that this class provides a broad description of a set of classes [2].
- Count Class Base (CCB): CCB is the number of immediate base classes that a class inherits from [23]. In C++ multiple inheritance is allowed. However, in Java only single inheritance of classes is allowed but multiple inheritance of interfaces is allowed. Inheritance from both interfaces and classes is counted by this metric.
- Coupling between object classes (CBO): "CBO for a class is a count of the number of other classes to which it is coupled" [2]. Two objects are coupled if "one of them acts on the other, i.e. methods of one use methods or instance variables of another ". This measure of coupling includes coupling due to inheritance. Chidamber and Kemerer state that "excessive coupling between object classes is detrimental to modular design".
- Response for a class (RFC): RFC is a measure of the magnitude of the response set for a class. The response set is composed of all the methods in the class itself and all the methods which can be called from the methods in the class [2].
- Lack of Cohesion (LCOH): LCOH is 100% minus the "average cohesion for class data members" [23]. The cohesion of a class data member is the percentage of methods that use it.

- Fan-in (CountInput): This metric reflects the number of inputs that a function uses. It includes parameters and global variables [23].
- Fan-out (CountOutput): This is the number of outputs that are set. This could be parameters or global variables [23].
  CountInput and CountOutput reflect the information flow relationships between procedures and between procedures and data items [10, 22].
- Number of Declarative Statements (CountStmtDecl): This is the number of statements involving declarations in a method [23].
- Number of Executable Statements (CountStmtExe): This is the number of executable statements in a method [23].

3 Empirical Study

Some of the structural features of software systems developed in C++ and Java are compared in this study. In this section, the systems analysed are discussed, the research questions under investigation are outlined and the data analysis and interpretation is presented.

3.1 Study Design

Twenty open-source software systems were analysed in this study. Ten of the systems were developed solely in C++ and 10 solely in Java. These systems are taken from the top 100 most downloaded systems from Sourceforge.net. Systems written in multiple language were not considered as part of this analysis to eliminate the influence of other languages on the software projects analysed. For example, in systems written in multiple languages, C++ or Java may only be used for specific purposes within the projects, e.g. C++ might be used to implement some computationally intensive algorithms.

Systems from a number of different domains have been included in the analysis. This helps to broaden the applicability of any conclusions we may draw from the analysis. However, given that all the systems being analysed are from the open-source domain, conclusions drawn may not immediately extend to commercial software systems. However, "perceived differences between open source and conventional software development projects are only partially true" [17]. Many successful open-source project have paid developers who have face-to-face meetings similar to 'conventional' development.

Some summary statistics are presented in Table 1 and Table 2 for the Java and C++ systems analysed. These statistics show some interesting findings. The average values suggest that the C++ systems are bigger in terms of lines of code and number of files but have less than half the classes. The differences due to the number of files is probably because it is common in C++ to include both a header file with a class declaration and an implementation file with the definition of the class. In Java, it is common to use one file per class. The differences in the number of lines of code may also be partly due to the use of both header and implementation files in C++. It may also be the case that Java software systems utilise more built in functionality, that is, functionality provided in libraries external to the system itself.

The number of classes in the C++ and Java systems bucks the trend of the number of files and number of lines of code to suggest that based on this measure C++ systems are smaller than the Java systems. However, the number of lines of code is a more tangible (i.e. possibly more intuitive) measure of size. Lorenz and Kidd suggest thresholds for some object-oriented metrics. Based on their guidelines for the number of lines of code in methods and the number of methods per class in C++ systems, the suggested upper threshold on the number of lines of code per class is in the range 300-400 LOC. There is just one system which seems to be an outlier on the upper side of these guidelines and that is Audacity. Guidelines for Java projects don't exist but if such existed they may be more in line with Smalltalk projects. Lorenz and Kidd

suggest that the "average method size for Smalltalk should be under six lines of code". However, it should be noted that these figures should only be used as guidelines. Mayrand *et al.* state that metrics "should always be used as inspection guidelines instead of being used as absolute rules". Further work is needed to provide guidelines for different programming languages and for different application domains. Such guidelines should have a strong empirical basis.

Table 1. Summary Stats for 10 **Java** Systems

System	No. of Classes	Files	LOC	Domain
Openstego	21	14	2212	Security
VLM	84	36	4425	Chat
Opencard	205	109	8826	Learning
Sweet Home 3D	490	96	21526	3D Rendering
Areca	449	323	35384	Backup
Freemind	864	509	70642	File Management
iText	499	427	79154	PDF Generation
JEdit	890	394	88443	Text Editor
TV Browser	1867	834	98791	TV Guide
Freechart	991	920	128209	Charts
Average	636	366	53761	

Table 2. Summary Stats for 10 **C++** Systems

System	No. of Classes	Files	LOC	Domain
Meshlab	37	34	3852	Rendering
Eraser	97	222	32641	Data Removal
Mumble	145	236	34023	Chat
AC3	110	197	47059	AVI Playback
DC++	366	349	62936	File Conversion
Filezilla	238	292	77287	Filezilla
7Zip	485	978	89407	File Archival
Winscp	398	372	112255	File Transfer
Emule	416	682	162595	File Transfer
Audacity	200	983	178027	Audio Editor
Average	249	435	80008	

3.2 Research Questions

A number of research questions are addressed in this study:

RQ1: How does the size of classes compare across C++ and Java systems?

The summary statistics presented in Tables 1 and 2 suggest that on average classes in Java systems are much smaller than classes in C++ systems. This hypothesis investigates this claim further.

RQ2: How does the use of inheritance differ across C++ and Java systems?

Inheritance is one of the distinguishing features of the OO domain. It has received much attention in the literature in terms of what constitutes appropriate uses of inheritance. In this paper, we consider the structure of inheritance hierarchies in systems developed in C++ and Java. We consider how the use of inheritance might effect external quality characteristics of the system such as understandability and maintainability.

RQ3: How much does the coupling and cohesion of classes differ between C++ and Java systems?

Low coupling and high cohesion are two of the guiding principles of OO design. High coupling between system parts makes it more difficult to understand one part of a system on its own. High coupling has been associated with fault-proneness [1, 7, 9, 24, 25]. Each class should form a cohesive unit by itself and should not be composed of two or more cohesive units.

RQ4: How does the structure of methods in classes differ between systems developed in C++ and Java?

This research question considers the size of methods in classes and coupling between methods. Guidelines for good object-oriented design would suggest that these characteristics should be small.

Two of the main issues that may effect the generalisation of any conclusions drawn from the above hypotheses are that the systems analysed vary greatly in size and that the systems are from a range of application domains and the conclusions may not hold across all domains. The analysis of the research questions outlined above is undertaken in Section 3.3, and following this two systems (one Java and one C++) of comparable size and two systems (one Java and one C++) from the same domain are compared using the metrics outlined in Section 2.2.

3.3 Data Analysis

RQ1: How does the size of classes compare across C++ and Java systems?

Table 3 presents four metrics illustrating the size of classes in Java and C++ systems. In the systems analysed, classes in C++ systems consistently seem to have more methods than classes in Java systems. Lorenz and Kidd suggest an upper threshold of 20 (40 for User-Interface classes) for the number of instance methods in a class. While the mean NMC (Number of Methods per Class) falls within this range for both C++ and Java classes, the maximum value is many multiples of this threshold. The larger values for the number of methods in classes in C++ systems suggest that these systems may be more difficult to understand and maintain. In addition the average number of lines of code per class is 85 for the Java systems (53761/636 from Table 1) and 321 for the C++ systems (80008/249 from Table 2). If software developers/maintainers attempt to understand complete classes during the course of their maintenance work, then the job of the C++ maintainers is considerably more difficult than that of the Java maintainers. This analysis assumes that all C++ source code is part of a class when this may not be the case. In fact, in many C++ systems there may be functions that do not belong to any class. This is in contrast to Java which is a much more class-centric language. By excluding the C++ functions that do not belong to classes, the average number of methods per class and the average number of lines of code per class may be reduced although it is unlikely that they would be reduced in line with the corresponding values for the Java systems.

The use of private and protected methods is limited in the systems analysed. At least 50% of all the classes in both C++ and Java systems do not have any protected or private methods. In fact the Java systems use protected methods to a very limited extent (given a max value of 20). The C++ systems seem to use protected and private methods equally, with the mean, median and maximum values for CMPro (Number of Protected Methods in class) and CMPri (Number of Private Methods in class) showing similar values.

As well as being measures of size, CMPub (Number of Public Methods in class), CMPro and CMPri also reflect the level of information hiding in classes. Given such small values of CMPro and CMPri suggests that principles like information hiding [18] are not being (consistently) adopted. Meyers also suggested that an interface should be complete and yet minimal, [16]. Application of these principles in practice would require considerable use of the protected and private visibility specifiers in Java and C++ systems, something which seems to be lacking in the systems analysed here.

RQ2: How does the use of inheritance differ across C++ and Java systems?

Table 4 presents some inheritance-based metrics for the 20 systems analysed. The minimum values of these metrics will always be zero, since there will always be classes at the root of

Table 3. Size Metrics for Java and C++ Systems

	Min	Max	Median	Mean	St. Dev.
Java					
NMC	0	351	3.5	6.67	5
CMPub	0	214	2.5	4.5	7.2
CMPro	0	20	0	0.3	2.2
CMPri	0	257	0	0.8	0.8
C++					
NMC	0	564	6.5	10.4	8.9
CMPub	0	557	4	7.1	8.2
CMPro	0	116	0	2	4
CMPri	0	110	0	1.9	3.4

a hierarchy (or stand-alone classes) and there will always be classes that do not have any children. The max value for DIT (Depth in Inheritance Tree) only differs by 1 for the C++ and Java systems suggesting that C++ systems may use inheritance a bit more than Java systems. However, the mean value of DIT is considerably larger for the Java systems than for the C++ systems. This suggests that inheritance is much more common in Java systems than in C++ systems. This claim is supported by the fact that the range of DIT values is the same for both sets of systems (in fact the maximum DIT for Java systems is one less than for C++ systems) and the fact that there are more than double the number of classes in Java systems than C++ systems. These facts suggest that inheritance is used more consistently in Java systems.

It is also worth noting the Lorenz and Kidd propose a threshold of six for the maximum DIT. The Java systems seem to comply exactly with this threshold with the C++ systems just outside the threshold. It is unlikely that the real-world being modelled by the OO systems have more than six levels of specialisation. Classes at deeper levels of inheritance are harder to understand and to test [13].

Table 4. Inheritance Metrics for Java and C++ Systems

	Min	Max	Median	Mean	St.Dev.
Java					
DIT	0	6	1	1.5	0.2
NOC	0	122	0	0.4	2.3
CCB	0	34	1	1.6	0.5
C++					
DIT	0	7	1	0.9	0.2
NOC	0	129	0	0.4	1.6
CCB	0	12	1	0.8	0.3

The distribution of the Number of Children (NOC) seems largely similar for both sets of systems. Given that the minimum, median and mean for both C++ and Java systems is zero at least 50% of classes are either part of trivial hierarchies (stand-alone classes) or are leaf nodes in the inheritance hierarchies. However, in both C++ and Java systems there is at least one class with in excess of 120 children. These classes are extensively reused through subclassing. It is likely that these classes are also key classes in the systems.

The Count Class Base (CCB) metric counts the number of immediate base classes for each class. For the Java systems this counts interfaces as well as classes. This is the reason why the max value is greater than one and most likely why the max value is considerably larger for Java systems than for C++ systems. The minimum and median values are the same for both Java systems and C++ systems. Otherwise, the average value of CCB supports the claim that Java systems have more inheritance than C++ systems.

RQ3: How much does the coupling and cohesion of classes differ between C++ and Java systems?

Low coupling and high cohesion are seen as a corner-stones of good object-oriented design. Here the coupling of the systems is studied through the CBO and RFC metrics and the cohesion of classes is studied through the LCOH metric.

The CBO (Coupling Between Object Classes) metric presented in Table 5 doesn't show any significant difference for C++ and Java systems. However, the RFC metric is significantly larger for classes in the C++ systems than the Java systems. The increased message passing highlighted by this metric (which is hidden by CBO) represents higher coupling in the C++ systems and therefore suggests that these systems may be more difficult to understand and maintain than the Java systems.

The LCOH (Lack of COHesion) metric returns percentage values and higher percentages indicate a greater lack of cohesion. The median and mean values in Table 5 suggest that the lack of cohesion is much greater in the classes in the C++ software systems compared to the Java software systems. Combined with the coupling results this data suggests that the classes in the Java systems follow the coupling and cohesion guidelines more closely than the C++ systems.

Table 5. Coupling and Cohesion for classes in Java and C++ Systems

	Min	Max	Median	Mean	St. Dev.
Java					
CBO	0	243	2	3.4	2.4
RFC	0	351	4	7.5	14.9
LCOH	0	100	0	30.7	2.4
C++					
CBO	0	111	2	3.4	2.4
RFC	0	564	12.5	20.6	13.3
LCOH	0	100	47.8	63.3	9.5

RQ4: How does the structure of methods in classes differ between systems developed in C++ and Java?

It is considered good object-oriented design to keep methods in classes small. As mentioned earlier, Lorenz and Kidd suggest on average 24 lines of code per method for C++ methods. In this section, the structure of methods in C++ and Java classes is analysed using some additional metrics at the method level.

Table 6 presents an overview of these metrics for the Java and C++ projects analysed. As has been seen with other metrics the main difference in CountInput (information flow between procedures and data items) for Java and C++ systems is in the maximum value. While the mean and median values suggest that a majority of classes have reasonably small values for this metric, there are some methods with extremely large values. This indicates the presence of key methods in addition to key classes in systems. For CountOutput (information flow between procedures and data items) the C++ systems have significantly larger mean, median and maximum values. This suggests that in this regard C++ methods seem to have consistently larger fan-out than methods in Java systems.

As regards the number of declarative statements per method (the number of statements involving declarations), there seems to be little significant difference between the C++ and Java methods. However, with respect to statements executed C++ again seems to have consistently more executed statements in methods, illustrated by the larger mean, median and maximum values for the CountStmtExe method.

The significantly larger values of CountOutput and CountStmtExe provide an indication that C++ methods may be more difficult to maintain than Java methods. However, this claim requires further empirical study and analysis before a definitive conclusion can be drawn.

Table 6. Method-Level Metrics for Java and C++ Systems

	Min	Max	Median	Mean	St. Dev.
Java					
CountInput	0	690	3	4	2.5
CountOutput	0	142	2	3.7	0.5
CountStmtDecl	0	130	1	2.1	1
CountStmtExe	0	411	2	5.4	3.1
C++					
CountInput	0	438	3	4.9	2.7
CountOutput	0	324	5.6	5.1	1.5
CountStmtDecl	0	202	0	1.7	0.9
CountStmtExe	0	1254	4	10.5	5.8

Comparison of two systems of the same size

The systems analysed here were JEdit (Java) with 88443 LOC and 7Zip (C++) with 89407 LOC. Table 7 presents the results for both systems. The results for the C++ system are inside the brackets in the table.

The size metrics have larger average and median values for the C++ than for the Java system, reflecting the fact that in general the C++ classes are larger than the Java classes. This supports the earlier findings. The only difference in this refined analysis is that the Java classes have a larger maximum value. This doesn't effect the overall trend.

The inheritance metrics are largely similar. The only significant difference is in the maximum value for NOC which in this case is much larger for the C++ classes. The mean value for DIT and CCB is slightly smaller and NOC is slightly larger for the C++ classes, suggesting that the hierarchies in the C++ system are more shallow and wider than in the Java system.

The mean CBO for the Java system is 2 compared to 1 for the C++ system whereas the mean CBO values were the same when all systems were compared together. While both the mean and median RFC for all ten C++ systems was larger than the corresponding value for all 10 Java systems, a similar trend is only visible here for the mean value. Therefore, the conclusions suggesting that C++ systems have larger coupling holds but is not as strong in this comparison. However, there continues to be less cohesion in the C++ classes than in the Java classes.

The main differences in the method level metrics here is in the CountStmtExe metric where the mean value is 10 times larger for the C++ system than for the Java system. While the CountOutput metric demonstrated much larger values for all ten C++ systems, in the two systems compared here this metric is very similar for both projects.

Comparison of two systems from the same domain

The systems analysed here were SweetHome (Java) and MeshLab (C++). These are both rendering applications. Table 8 presents the results for both systems. The results for the C++ system are inside the brackets in the table.

While the Java system has larger maximum values for the size metrics, the mean value for WMC and CMPri suggests that the classes in the C++ system are larger than the classes in the C++ system. Again the lack of use of protected and private methods is obvious. The C++ system analysed here seems to use very little inheritance with a maximum value of 1 for both DIT and NOC. The coupling metrics again suggest that while the Java system may have a larger maximum value, the mean and median values suggest that generally classes in the C++ system have more coupling than classes in Java systems. The C++ classes again seem to have

Table 7. Metrics for a Java and C++ System of Comparable Size

	Max	Median	Mean	St. Dev.
JEdit (**Java**)				
WMC	351	3	6.2	17.7
CMPub	214	2	4.3	11.2
CMPro	13	0	0.1	0.7
CMPri	257	0	1	9
DIT	4	1	1.5	0.7
NOC	38	0	0.3	5.6
CCB	34	1	1.5	1.3
CBO	67	2	4	2
RFC	351	3	8.9	18.7
LCOH	100	0	27.6	33.7
CountInput	263	3	5.1	7.7
CountOutput	78	2	4.1	5.5
CountStmtDecl	39	1	2.1	2.7
CountStmtExe	411	2	.7	12.2
7Zip (**C++**)				
WMC	206	9	18	25.6
CMPub	112	4	6.7	8.6
CMPro	11	0	0.1	0.6
CMPri	41	0	1.2	3.8
DIT	4	0	1	1.1
NOC	129	0	0.5	6
CCB	12	1	1	1.4
CBO	57	1	4.1	6.4
RFC	419	0	19	2.1
LCOH	100	25	33.1	34.7
CountInput	391	2	5.5	11.2
CountOutput	68	2	3.8	6
CountStmtDecl	98	0	1.9	5.4
CountStmtExe	345	3	7.9	16.7

less cohesion. The method level metrics CountOuput and CountStmtExe suggest that the C++ classes have a higher fan-out than the Java classes. This is reinforcing the evidence identified across all twenty systems studied.

Table 8. Metrics for a Java and C++ System from the same Application Domain

	Max	Median	Mean	St. Dev.
Sweet Home **Java**				
WMC	96	2	4.3	8.9
CMPub	85	1	5.7	3.1
CMPro	16	0	0.2	1.1
CMPri	48	0	0.9	3.8
DIT	4	1	0.5	0.6
NOC	15	0	0.2	1.2
CCB	3	2	1.7	0.5
CBO	30	2	2.5	3.1
RFC	96	2	5.7	10.5
LCOH	100	0	16.9	30.4
CountInput	82	3	4.2	5.1
CountOutput	6	2	3.8	5.1
CountStmtDecl	69	1	2.2	3.2
CountStmtExe	153	2	4.3	7.3
MeshLab **C++**				
WMC	65	5	7.4	11.1
CMPub	14	1	3.2	3.9
CMPro	2	0.1	0.3	0
CMPri	47	0	2.7	7.8
DIT	1	0	0.5	4.8
NOC	1	0	0	0.6
CCB	2	0	0.5	0.2
CBO	21	2	3.4	0.5
RFC	66	6	7.9	11.2
LCOH	100	3.5	31.7	5.3
CountInput	72	3	4.9	7.6
CountOutput	94	3	5.9	9.3
CountStmtDecl	17	0	1	2.3
CountStmtExe	136	2	7.1	14.3

4 Conclusion

The following general conclusions can be drawn from the analysis presented in this paper:

- C++ systems tend to have more lines of code and more files but less classes, suggesting that overall classes in C++ systems have more lines of code.
- Following on from the previous point, classes in C++ systems also tend to have more methods.
- There is limited use of the protected and private visibility qualifiers in all systems.
- Java seems more consistent in its use of inheritance, even though some C++ systems have deeper levels of inheritance.
- Coupling in C++ systems tends to be higher than in Java systems and cohesion in C++ systems tends to be lower than in Java systems.
- At a method level C++ systems tend to have a higher fan-out and a higher number of executable statements.
- The findings outlined here seem to hold when a C++ and Java system of the same size are compared and when a C++ and Java system from the same application domain are compared.

The findings presented here suggest that C++ systems may be more difficult to understand and maintain than Java systems. However, more empirical studies need to be carried out to address this broader question. The lack of application of good object-oriented design principles such as information hiding is worrying. Future work will investigate this issue further. Software refactoring, [8] can help improve the design of systems like those analysed here. Future investigations should also investigate if such refactoring improve the understandability or maintainability of the software. Finally, a large cohort of software systems needs to be investigated to provide indicators or guidelines on acceptable ranges of object-oriented metrics for software projects.

5 Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant to Lero - the Irish Software Engineering Research Centre (www.lero.ie). I would also like to acknowledge the support of the CSIS Department, University of Limerick.

References

1. R.K. Bandi, V.K. Vaishnavi, and D.E. Turk. Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Transactions on Software Engineering*, 29(1):77–87, 2003.
2. S.R. Chidamber and C.F. Kemerer. A metrics suite for object-oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.
3. S.J. Counsell, G. Loizou, R. Najjar, and K. Mannock. On the relationship between encapsulation, inheritance and friends in C++ software. In *Proceedings of the International Conference on Software Systems Engineering and its Applications*, Paris, France, 2002.
4. S.J. Counsell and P. Newson. Use of friends in C++ software: An empirical investigation. *Journal of Systems and Software*, 53(1):15–21, 2000.
5. S.J. Counsell, P. Newson, and E. Mendes. Architectural level hypothesis testing through reverse engineering of object-oriented software. In *Proceedings of the Eighth International Workshop on Program Comprehension (IWPC 2000)*, pages 60–66, Limerick, Ireland, 2000.
6. S.J. Counsell, P. Newson, and E. Mendes. Design level hypothesis testing through reverse engineering of object-oriented software. *International Journal of Software Engineering*, 14(2):207–220, 2004.
7. K. El Emam, S. Benlarbi, N. Goel, and S. Rai. A validation of object-oriented metrics. Technical report, National Research Council of Canada, 1999.
8. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.
9. T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.
10. S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, 1981.
11. I. Joyner. *Objects Unencapsulated Java, Eiffel and C++??* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.
12. Y.C. Liu. Comparison between C++ and Java - a case study using a networked automated gas station simulation system. Master's thesis, Department of Computer Science, Concordia University, Montreal, Qubec, Canada, January 2002.
13. M. Lorenz and J. Kidd. *Object-Oriented Software Metrics.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1994.
14. R.C. Martin. Java and C++: A critical comparison, March 1997. www.sweforum.netmethodtooljavacpp.pdf.
15. J. Mayrand, J.F. Patenaude, E. Merlo, M. Dagenais, and B. Lague. Software assessment using metrics: A comparison across large C++ and Java systems. *Annals of Software Engineering*, 9:117–141, 2000.
16. S. Meyers. The most important design guideline. *IEEE Software*, 21(4):14–16, 2004.
17. J. Noll. *Open Source Development Communities and Quality*, volume 275/2008, chapter Requirements Acquisition in Open Source Development:Firefox 2.0, pages 69–79. Springer Boston, 2008.
18. D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
19. G. Phipps. Comparing observed bug and productivity rates for Javav and C++. *Software-Practice and Experience*, 29(4):345–358, 1999.
20. D.R. Pratt, A.J. Courtemanche, J. Moyers, and C. Campbell. An Evaluation of the Java and C++ Programming Languages. In *The Interservice/Industry Training, Simulation & Education Conference*, 2000.
21. L. Prechelt. Comparing Java vs. C/C++ efficiency differences to interpersonal differences. *Communications of the ACM*, 42(10):109–112, October 1999.

22. P. Rook. *Software Reliability Handbook*. Springer, 1990.
23. Scientific Toolworks Inc. Scitools maintenance, metrics and documentation tools for Ada, C, C++, Java and Fortran. web: www.scitools.com, 2006.
24. R. Subramanyam and M.S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.
25. P. Yu, T. Systa;, and H.A. Muller. Predicting fault-proneness using oo metrics: An industrial case study. In *CSMR '02: Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 99–107, Washington, DC, USA, 2002. IEEE Computer Society.