Comparing Observed Bug and Productivity Rates for Java and C++

Geoffrey Phipps
Spirus
gphipps@spirus.com.au

Summary

An experiment was conducted to compare programmer productivity and defect rates for Java and C++. A modified version of the Personal Software Process (PSP) was used to gather defect rate, bug rate, and productivity data on C++ and Java during two real world development projects. A bug is defined to be a problem detected during testing or deployment. A defect is either a bug, or an error detected during compile time. A typical C++ program had two to three times as many bugs per line of code as a typical Java program. C++ also generated between 15% and 50% more defects per line, and perhaps took six times as long to debug. Java was between 30% and 200% more productive, in terms of lines of code per minute. When defects were measured against development time, Java and C++ show no difference, but C++ had 2 to 3 times as many bugs per hour. Statistics were generated using Student's *t*-test at a 95% confidence level. Some discussion of why the differences occurred is included, but the offered reasons have not been tested experimentally. The study is limited to one programmer over two projects, so it is not a definitive experimental result. The programmer was experienced in C++, but only learning Java, so the results would probably favour Java more strongly for equally-experienced programmers. The experiment shows that it is possible to experimentally measure the fitness of a programming language.

Key Words
C++, Java, Programming Languages, Metrics

Background and Motivation

Much has been said and written about Java's claimed superiority to C++[1], but there is no hard data to back up such claims. The main reason that such data does not exist is that it is difficult and time consuming to perform the necessary experiments.

The Personal Software Process[2] (PSP) is a methodology designed by Watt Humphrey to be used by individual software engineer. Unlike most methodologies, PSP is an experimentally based process and so the claims can be tested experimentally. I used PSP for a four month C++ project in late 1996 and found that it did improve my project estimation and code quality. Accordingly I used PSP for my next project, which was written in Java. At that point I realised that I had accurate productivity and defect numbers for two projects that differed mainly in the implementation language. Hence it was possible to experimentally compare C++ and Java.

The aim of both projects was to produce commercial quality code to ship to customers. PSP was used to achieve that goal, comparing C++ and Java was only a side effect. The idea of comparing the languages only emerged after the C++ project was concluded, so there are gaps in the C++ data. This experiment is therefore *not* definitive, rather it points the way towards a definitive experiment. However, this work is more robust than unsupported opinion.

Personal Software Process (PSP)

The aim of PSP is to allow individuals to "control, manage, and improve the way [they] work."[2] The specific aims are to improve:

1) estimation accuracy
2) code quality
3) productivity

PSP is a methodology for individuals, not for whole corporations. Although the published version has many steps to perform and many forms to fill out, the core design principle is simple: make careful quantitative observations of your progress and use feedback to improve subsequent projects. The minutes spent on every phase of a project are tracked, as are the number of lines of code written or modified. These measures are used to predict how long it will take to complete future projects. All that is required is some self discipline and management support.

Method

General

Two projects of approximately equal complexity were specified, designed and implemented by the author using the same software methodology. Both projects used PSP, approximately at level 1.1. The exact projects and other aspects of the development environment are discussed in Section .

Specific

Both projects were quite small, so they used a waterfall model for the overall plan, with incremental delivery for the actual coding. Each release cycle was approximately six weeks long. The project steps in detail were:

1) Formal requirements gathering for all the functionality, resulting in a requirements document.
2) Production of a project plan identifying the pieces of functionality to be included in each release.
3) High-level design to identify all the components, i.e.the task breakdown. Each component was estimated to take between three and five days to complete.
4) A series of release cycles, each cycle having the following phases:
   - 4.1) Detailed design of each component to be included in the current release. Map each important use-case scenario to a message trace diagram. Identify all classes and their major methods.
   - 4.2) Estimate the number of lines of code using historical metrics. This number is known as the “designed lines of code” estimate.
   - 4.3) Multiply the designed lines of code in Step 4.2 by the developer’s personal expansion factor. The result is the predicted number of lines. The personal expansion factor is described below.
   - 4.4) Write the code, recording the elapsed time and defect rates.
   - 4.5) Update the expansion factor based on the predicted time and feature metrics as compared to the actual results.
   - 4.6) Use the new expansion factor to refine the estimates for the next cycle.

It is important to understand the expansion factor used in Step 4.3. It is the observed difference between the paper design and the final implemented system. It represents the degree of detail to which you take your design before beginning coding. Obviously the expansion factor varies from individual to individual. For someone who is very careful and manages to identify all classes and methods before they begin coding, the factor will be around 1. A person who likes to code without much prior thought will have a very high factor. My personal factor is 1.8, meaning that for every line of code I identify during design, I consistently write 1.8 lines during development. If you use PSP in a consistent fashion then your expansion factor will stabilise after several iterations. What matters is that your expansion factor stabilises, *not* that it be 1.

The method used is a modification of PSP. The forms were changed to reflect object-oriented languages, and to streamline the data capture. Although PSP has been ported to C++, it was originally designed for procedural languages and that heritage is still visible. For example, defects are categorised as either “function” or “data” in the original PSP forms. The difference between function and data (largely) disappears in the OO paradigm, whereas new types of defects occur because of the inheritance system. Hence the defect categories had to be changed. The categories used were:

1) Syntax errors not covered in other categories
2) Incorrect package use
3) Declaration errors
4) Method call errors
5) Control flow not covered in other categories
6) Class Design
7) Method Design
8) Missing Defensive Programming checks
9) Documentation

10)Incorrect tool usage

As mentioned earlier, both projects used PSP, approximately at level 1.1.The elements from PSP level 2 that were not used were:

1) Prediction intervals
2) Formal design and code reviews
3) Analysis of appraisal time and failure time

The elements from PSP Levels 2 and 3 that were used are:

1) Functional specifications
2) Operational scenario templates
3) Cyclic development strategy
4) Test plan
5) Design templates
6) Issue tracking
7) Documentation standards for all documents

The actual document formats used for items 1 through 5 differed from the PSP formats, but they contained the same information.

The Numbers

Definitions

Lines of Code This work uses the simple definition of a non-blank line. The reasons are discussed in "Choice of Units".

Defect A defect is any error that causes an unplanned change in the code. Defects include syntax errors, logical errors, and changes in variable names. The category does not include replacing code that was known to be prototype code when it was written, or changes caused by a change in the project requirements. This is the standard PSP definition.

Bug In this work a bug is defined to be a problem detected during test or deployment. A bug is therefore a control flow defect or a missing feature. The definition excludes syntax errors, type errors, and other errors caught by the compiler.

Minute A minute of uninterrupted work. Telephone calls, staring out the window and the like are excluded.

Defects per Line of Code

I was learning PSP during the C++ project, and the effect is obvious in the graph of defect rates (see Figure 1). After the second task I reviewed my defect logs and detected patterns in defects. By paying attention to those problematic coding habits I was able to reduce the defect rate from approximately 260 per thousand lines of code (**kLoc**) to approximately 50 per kLoc. Therefore the statistical study ignores the first two data points because I was using a different method. In addition, the sixth data point has been excluded because there were problems recording the number of lines changed. The coding rate (lines of code per minute) was twice as high as usual, and the defect rate half the average. The task also spanned a three week period of leave. Although there is no solid proof, it appears that I accidentally double-counted the number of lines written during that subtask. Of course, the exclusion of these data points underlines the preliminary nature of this investigation. The final defect counts are shown in Figure 2. All subsequent discussion will be restricted to this set of data points, called “the relevant set.” Although an informal examination of the graph implies a final defect rate of around 90, it is clear that the defect rate has not stabilised for C++. The mean for Figure 2 is 82 defects per kLoc and the observed standard deviation is 25.

If we restrict our attention to bugs (errors detected during testing or deployed usage), then the data for the relevant C++ set is shown in Figure 3. The mean is 18 bugs per kLoc and the observed standard deviation is 8.

The Java project was implemented after the C++ project, and as a result the PSP methodology was already stable. The only data point excluded is the first task, because it was affected by the author learning Java. The graph of defects is shown in Figure 4, the bug rates are shown in Figure 5.

Informal examination of Figure 2 and Figure 4 seems to reveal a final C++ defect rate of around 90 per kLoc, and a Java defect rate of around 60 per kLoc, for a ratio to about 3 to 2. However, statistical testing provides clearer insight.

The standard method for comparing two means is to compute an estimate of their difference. For small sample sizes (fewer than 30 samples) we use Student's *t*-test[3]. For a given confidence level (1- α), the *t*-test defines a range within which the difference between the means must lie. The test assumes that both populations are normally distributed, and that they have the same standard deviation. The two populations are defect rates for the author writing C++, and defect rates the author writing Java. The two activities are similar enough to assume that they have the same standard deviation.The difference in the observed standard deviations is caused by the smallness of the sample sizes.

The relevant means, standard deviations, and sample sizes for the relevant sets of both C++ and Java are shown in Table 1.

	Defects		Bugs		No. of samples
	Mean	Std Dev	Mean	Std Dev	
C++	82	25	18	8	7
Java	61	11	6	2.5	5

Table 1: Observed Defect and Bugs per kLoc

We are interested in testing at a 95% confidence limit, i.e. when α is 0.05.

Using the numbers for defects, we find that the difference between the two means lies in the interval (9.7, 32). In other words, we can be 95% confident that C++ has at least 9.7 more defects per kLoc than Java, perhaps even 32 more defects per kLoc. If we take the observed mean for Java of 61 defects per kLoc, then C++ has between 15% and 52% more defects. Informal examination of the graph implied a difference of about 50%. It is interesting to note how seemingly convincing graphs have to be carefully examined using statistics.

The bug rates can be analysed in the same way. Here the difference between the two means lies in the interval (8.6, 15). So we can be 95% confident that C++ has between 8.6 and 15 more bugs per kLoc than does Java. Comparing it to the base rate for Java of 6 bugs per kLoc, the difference is between 240% and 360%.

The experiment suggests that a C++ program will have three times as many bugs as a comparable Java program.

Defects per Hour

The same set tests can be applied to the same data, but measuring defects per hour, rather then defects per kLoc. The defect per hour statistics are shown in Table 2.

	Defects		Bugs		No. of samples
	Mean	Std dev	Mean	Std dev	
C++	8.38	8.23	1.53	0.95	7
Java	5.35	1.82	0.56	0.25	5

Table 2: Observed Defect and Bugs per Hour

Clearly the C++ defect mean is very noisy, because the standard deviation is almost the same as the observed score. The noise is introduced by just one data point, without which the main is 5.42 defects per hour and the standard deviation 2.76. However, we continue to use the same relevant set as before.

The result for defects is that the C++ mean could be between 0.5 below, and 6.6 above, the Java mean. In other words, we cannot be sure which language produces fewer defects per hour. When restricted to bugs, the C++ mean is 0.56 to 1.4 defects per hour higher, which translates to a factor of between 2.0 and 3.5.

So Java and C++ produce the same number of total defects per hour, but C++ has two to three times as many bugs per hour. The first result is perhaps not surprising, given that Java and C++ are almost identical syntactically.

Time to Fix Defects

The fact that Java's hourly bug rate is approximately one-third that of C++'s is not the only difference. It is also true that problems in Java programs can be fixed faster. Table 3 shows the time taken to fix a defect according to the phase where it was removed. The relative sizes of the numbers in the C++ column are not surprising, it has been known experimentally for years that the earlier in the development cycle a bug is removed, the less time it takes.

Phase Fixed	C++	Java
Compile	6	1.5
Unit Test (Bugs)	30	15
Post-deployment	240	?

Table 3: Minutes to Fix Defects by Phase

Explaining the difference between Java and C++ is more interesting and complicated. It is likely that the dominant reason for the faster compile-time fixes in Java is the change from command-line compilation to an integrated IDE. The experiment cannot show whether the improvement is speed is due to Java versus C++, or Symantec Cafe versus emacs/make, but I suspect the environmental change dominates.

Unfortunately the experiment did not categorise defects by the language feature (or missing language feature) which caused them. Hence this paper cannot make a claim such as "Java's lack of templates causes 17% of all errors." Not only would finer-grained data need to be collected, the sample size would have to be much larger because there are more categories.

The difference in debug times is probably linguistically based. For various reasons the same basic technique (simple trace messages) was used to debug both C++ and Java code. Hence the change in programming environment is less important, and the dominant cause is the change in programming language. The main cause is most likely Java's safe memory model. The hardest and most time consuming bugs to remove in C++ are those caused by bad pointers and other mistakes made in memory management. Java has eliminated these errors, so it is not surprising that the mean time to fix bugs has dropped. The bug and defect forms did not include a separate category for memory problems, such problems were included in the heading "control flow". In fact, 95% of bugs were "control flow", with "class design" and "method design" each accounting for another 5%. Therefore the study cannot definitively say that C++'s memory is the cause.

Debug times include the time taken to rebuild each application. Data for the time spent compiling was not recorded, so no exact comparison can be made. However, a 30 minute debug session in C++ probably included two compiles of around three minutes each. So, if it were possible to construct a C++ compiler that could compile as fast as a Java compiler, then the true times for C++ and Java unit-test bug fixes would be more like 25 minutes and 15 minutes respectively. However, due to C++'s macro preprocessor, it is not possible in general to compile C++ as fast as Java. The reason is the inability to store precompiled header files, because the meaning of each header file can change completely based on macro definition passed in on the command line.

Informally, it appears that it takes twice as long to fix a bug in C++ than it does in Java. Combined with the triple bug rate, it is likely that it takes 6 times as long to debug a C++ application than a Java application. Note that this six-fold increase has not been proven statistically, and it only applies to the debug phase of development, not the entire development cycle.

Productivity

Productivity is measured as lines of code per minute. The justification for choosing this measure is given in "Choice of Units". The productivity results for the relevant C++ set is shown in Figure 6, the results for Java are in Figure 7. The statistical values are shown in Table 4. Using these numbers and the *t*-test, we find that Java's mean productivity is between 0.28 and 0.95 lines of code per minute higher than the productivity rate for C++. This is between a 30% and 200% increase. The data is too noisy to be cer-

Data Set	Mean	StdDev	No. samples
C++	0.93	0.46	7
Java	1.55	0.78	5

Table 4: Observed Productivity Statistics

tain that Java is twice as productive as C++, but it is certainly better.

Choice of Units

Project "size" can be measured in many ways. The first metric of interest to software professionals is "size," which we measure in thousand lines of code (kLoc). The second metric is time. The third metric is delivered functionality (scope of the external requirements). The fourth measure is "complexity," meaning the "difficulty of the program to be understood by human programmers." Complexity is thought to indicate effort required for program maintenance.

Java and C++ could be compared using all four metrics (size, time, functionality, and complexity). This restriction to the first two metrics, and the units used, are discussed in the following sections.

This paper uses line counts because it is the most commonly used measure of software system complexity in industry. Practising project managers and programmers know that line counts are flawed, but line counts are widely used and understood. If this study is to be understood by those who work in software, then it must use the de facto standard for "project size."

Size

It is generally recognised that simple line counts are not a reliable metric of complexity of a piece of software[4]. Line counts vary according to code formatting conventions of the developers, and also by the expressive power of the language (consider APL versus COBOL). An arguably more accurate measure would be to measure linguistic elements, such as statements and expressions. However, this study counts non-blank lines because the same formatting style was used for both projects. Therefore *n* statements of Java take the same number of lines as *n* statements of C++. There are no errors introduced by white space. Grammatically based line counters for C++ are difficult to write correctly.

Time

Time is measured using the PSP metric, which is minutes spent on the actual task. Time spent doing other things (attending meetings etc.) is ignored. This unit is used in preference to simple elapsed days, because it avoids the largest source of error - time spent on other things. It does not account for the loss of efficiency caused by very short interruptions.

Functionality

Measuring defects per delivered functionality is arguably the best way to compare Java and C++. Delivered code, is, after all, the goal of most projects. Any future work on this study should consider functionality, but it is likely that the fundamental results will not change, for the following reasons.

If the same functionality (program requirements) are implemented using Java and C++, then it is reasonable (but unproven) to assume that the two programs will be similar in size because the two languages have very similar expressive power. Hence it is reasonable to assume that if Java produces fewer defects per kLoc than C++, then Java will produce fewer defects per function point than C++. The assumption that the two languages are sufficiently similar enough for simple "lines of code" comparison to be valid has not been proven. Therefore any future work on this study should include function point analysis to check this reasoning.

Complexity

Line-counts (and other size metrics) do not necessarily measure complexity. Other metrics exist[4], such as class-coupling (which measures the degree of interconnections between classes). Most of the proposed complexity metrics have not been proven experimentally to predict effort (programmer time), so they are not useful. In addition, very few of the complexity metrics are used or understood by software professionals.

Other Possible Sources of Error

This experiment was not conducted in a closed environment where all the variables were identified and strictly controlled. This section discusses the non-linguistic variables which probably affected the experimental results.

Generality of the Programmer and the Two Projects

The experiment used one programmer, one C++ project and one Java project.The results can only be generalised to "all" programmers and "all" C++ and Java projects if the programmer is a "typical" programmer, and the applications are "typical" applications. There are no such standards, so we must resort to informal argument.

The particular programmer has nine years commercial experience in seven companies, and two years programming during a PhD thesis. Seven of these years were spent using object-oriented languages. The programmer has been exposed to many different programming cultures. From personal observation the programmer claims that his coding style is not unusual.

This paper claims that the programmer is "typical enough for generalisation." However, to be precise it should be limited to programmers who are experienced with C++ but new to Java.

The first project was a C++ metrics tool written in C++. The code base has two major elements, the C++ parser, and the classes that modelled the parsed C++ code. The code was developed in traditional UNIX fashion using emacs and make on HPUX. The parser was written using ANLTR from the Purdue Compiler Construction Toolkit (PCCTS[7]). Before starting the project I expected that productivity and defect rates would differ markedly between the parser code and the model code. The parser was expected to be more difficult and therefore have higher defect rates and lower productivity. Interestingly the observed data showed no difference between the two sections.

The second project was a Java tool to numerically analyse large numbers of telephone call records. It was a standalone application written in Java 1.1 using Symantec Cafe.

Both applications included some parsing, although the parser in the telephone call record application was much simpler than the C++ parser. Both projects used files for I/O. Both had complex internal processing. Only the Java project had a GUI, but it was excluded from this study. Both projects took one person three months to complete.

External Environment

The C++ project was completed in a typical software office, shared with two other people. The Java project was completed in a home office. The two environments are quite different, but the experiment's method reduces the effect of the office environment because only time actually spent coding is counted. All interruptions are excluded. On the one hand, constant interruptions cause you to forget what you are doing when you do get back to work. On the other hand, there is no one to ask questions of when you get stuck. Analysis of the number of minutes worked each day certainly showed that there were fewer interruptions at home.

Development Environment

The C++ project was written using emacs and make on Unix. The Java project was written using Symantec Cafe, and with Java the operating system is irrelevant. The differences in development environments are discussed in Section .

Future Work

This experiment has many limitations. The experimental method was being perfected at the same time as the data was being gathered, so some of the data collected is suspect and excessively noisy. The noise, combined with the small sample size, makes it difficult to draw statistically reliable conclusions. The method does appear sound, so a future experiment involving two projects, each of four months duration is being planned. The major problem is finding two commercial projects of the correct size, of similar difficulty, and belonging to companies who are willing to participate in such an experiment. It is hoped to include function point analysis as well.

Experimental Conclusion

The method appears sound, if we ascribe the current noise to start-up effects. A longer and more careful experiment would probably answer the three questions (comparative defect rates, bug rates, and productivity rates).

The ratio of C++ bugs-per-kLoc to Java bugs-per-kLoc is statistically proven with a confidence level of 95% to be in the range 2.5 to 3.5. C++ also generates between 15% and 50% more defects per kLoc. Neither language appears better for defects per hour, but C++ produces between 200% and 300% more bugs per hour. Java is also between 30% and 200% more productive, in terms of lines of code per minute. It probably takes twice as long to fix each C++ bug, but this is not statistically proven due to the effect of differing compiler technology.

The experiment used one programmer, with one C++ project and one Java project, each of 3 months duration. The study assumes that the applications are representative of all applications. The programmer had seven years of C++ experience, but was only learning Java. Therefore the result can only safely be extrapolated to experienced C++ programmers who are learning Java. However, given that the results favour Java, and we assume that experienced programmers are better than inexperienced programmers, then the results would favour Java even more markedly if the programmer had equal experience in both Java and C++.

It is assumed that the samples follow a normal distribution.

New programming languages appear from time to time, but are rarely adopted. Part of the reason is due to the way new languages are marketed, but a major reason is that we have no way to evaluate languages. How can we know if a new language is better? Turing Equivalence is an interesting mathematical theory, but it has little predictive power as a scientific theory of humans writing programs. According to Turing machine theory, well-structured C++ has the same power as machine code written in binary, yet we know that humans in practise can build much larger systems with the former. Hence the two languages do not have the same power, so Turing theory is a poor predictor. This paper is an attempt to apply conduct a scientific experiment, but the results clearly show how difficult it is to design such experiments in practise. However, progress in Computer Science will only occur fortuitously unless we make greater use of the scientific method.

Acknowledgements

This work was carried out as a side effect of my normal role as software architect and developer. Accordingly thanks are owed to all the organisations and people who allowed me to collect the necessary numbers, although the analysis and paper writing was done on my own time. The C++ project was written at Object Oriented Pty Ltd. Phil Haynes of OOPL allowed me to use PSP and showed me the importance of statistical analysis.

Richard Favero of Soprano Designs allowed me to continue this research on the Java project.

The reviewers provided very useful advice on statistical methods and metric analysis.

Michael Van De Vanter of Sun Microsystems Laboratories provided encouragement and arranged for me talk at SMLI and other places within Sun. James Gosling of Javasoft provided comments on the Sun talk. Lisa Stapleton organised the chat show on JavaLive[8]. The participants offered many useful suggestions.

Malcolm Atkinson of Glasgow University and Dag Sjoberg of the University of Oslo provided useful leads to related work. Marcia Derr and Glen Diener organised my early talk at USWest in Boulder Colorado.

Bibliography

1. George Gilder, *Forbes ASAP*, pp 123-130, August 25, 1997.

2. Watts S. Humphrey, *A Discipline of Software Engineering*, ISBN 0-201-54610-8, Addison Wesley, 1995.

3. John E. Freund and Ronald E. Walpole, *An Introduction to Mathematical statistics*, Prentice-Hall International, 1980.

4. S. D. Conte, H. E. Dunsmore, V. Y. Shen, Software Engineering Metrics and models, Benjamin/Cummings Publishing Company, 1986, ISBN 0-8053-2162-4

5. *Function Point Counting Practises Manual (Release 4.0)*, International Function Point Users Group Standards, January 1994

6. Geoffrey Phipps, “The Structure of Large C++ systems,” in Technology of Object-Oriented Languages and Systems 18, Melbourne 1995, ISBN 0-13-477200-8

7. Terence Parr, *Language Translation Using PCCTS & C++*, ISBN: 0-9627488-5-4

8. Geoffrey Phipps and Lisa Stapleton, and the on-line audience, *JavaLive Chat Show*, URL: http://developer.javasoft.com/developer/discussionForum/transcripts-8-97/CvsJava.html

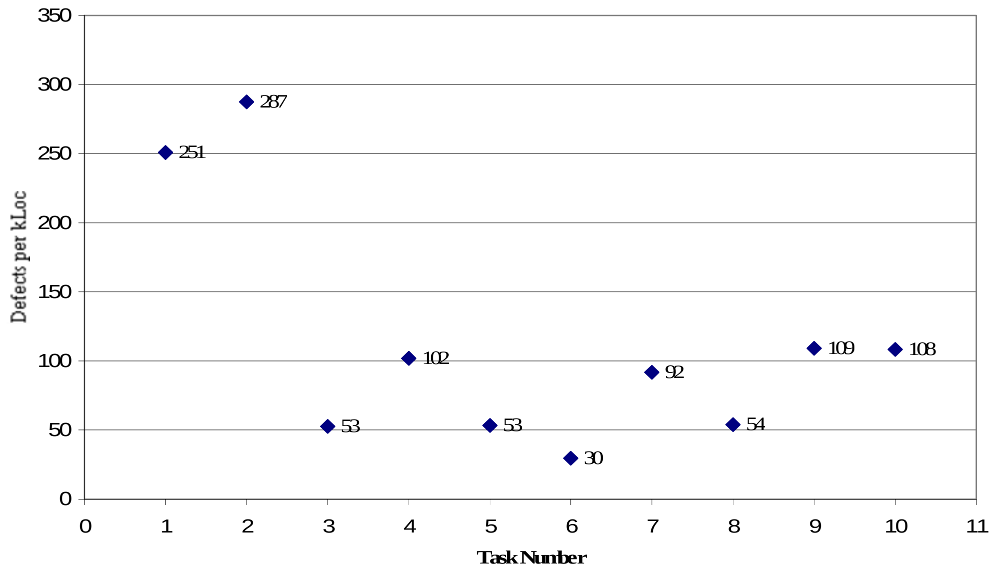

Figure 1: Defect Rates for All C++ Tasks

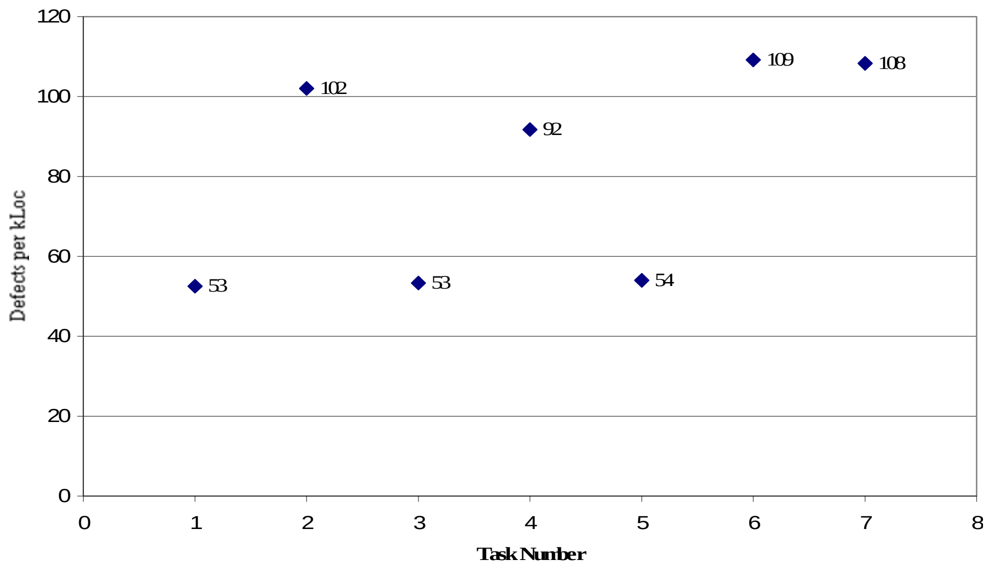

Figure 2: Defects for Relevant C++ Tasks

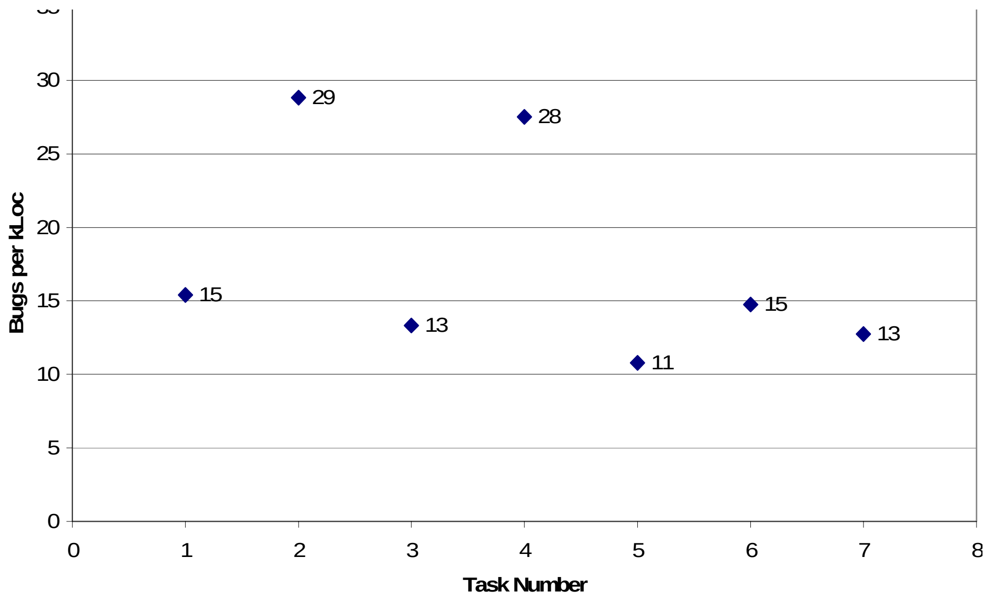

Figure 3: Bugs (Test Defects) for Relevant C++ Tasks

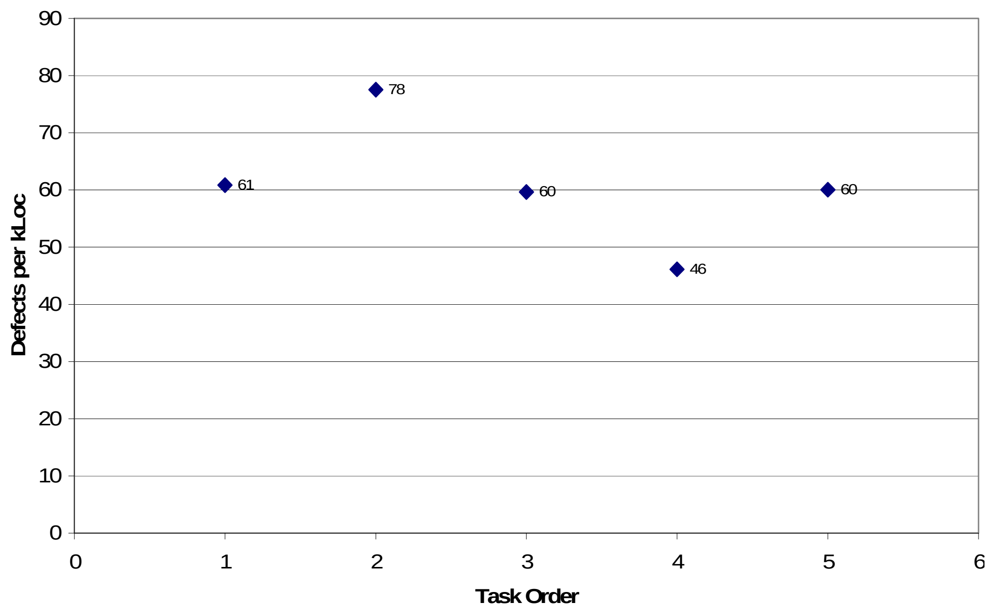

Figure 4: Java Defects

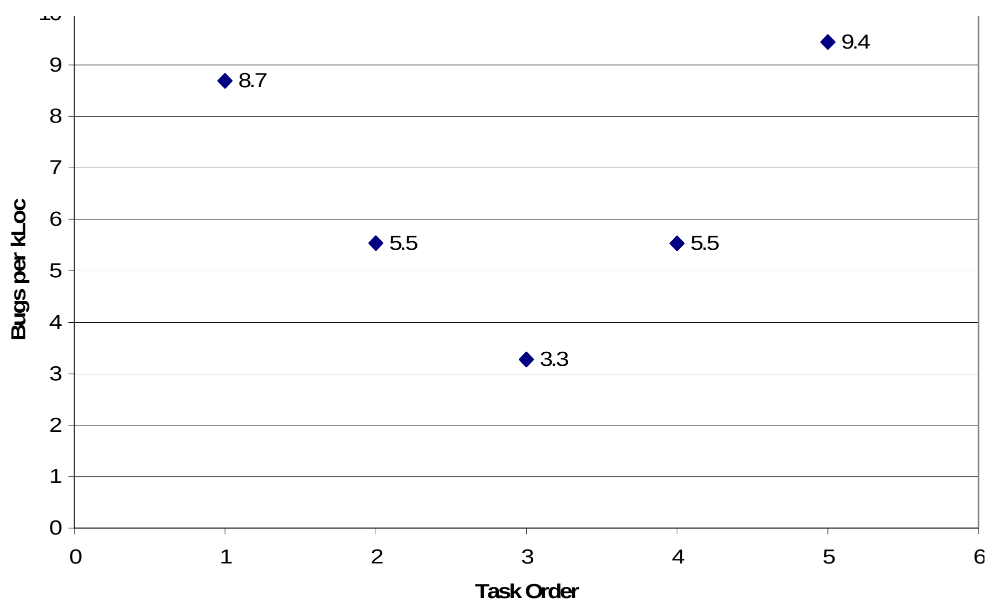

Figure 5: Java Bugs

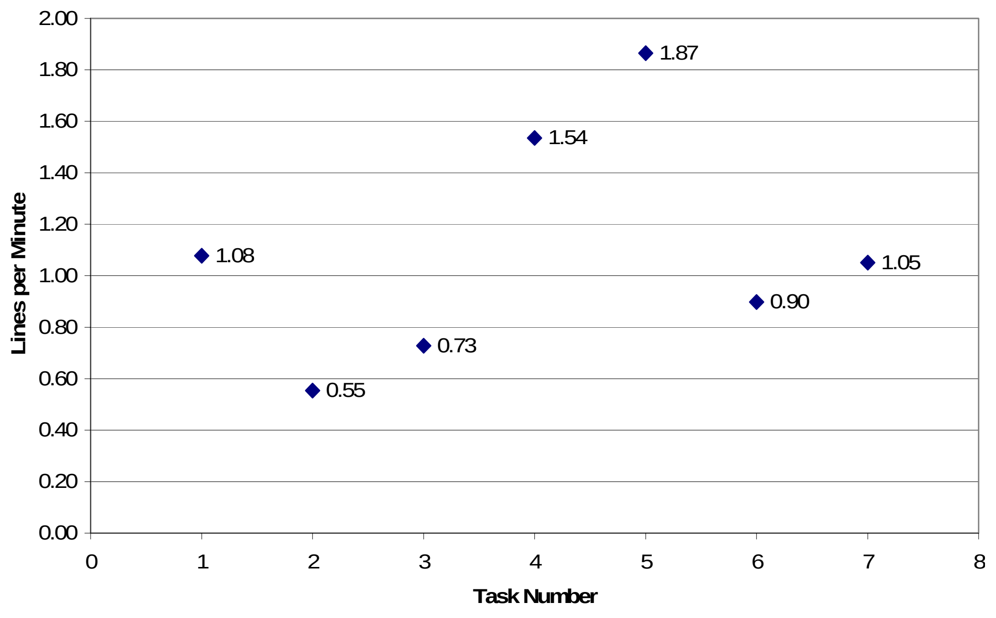

Figure 6: C++ Productivity

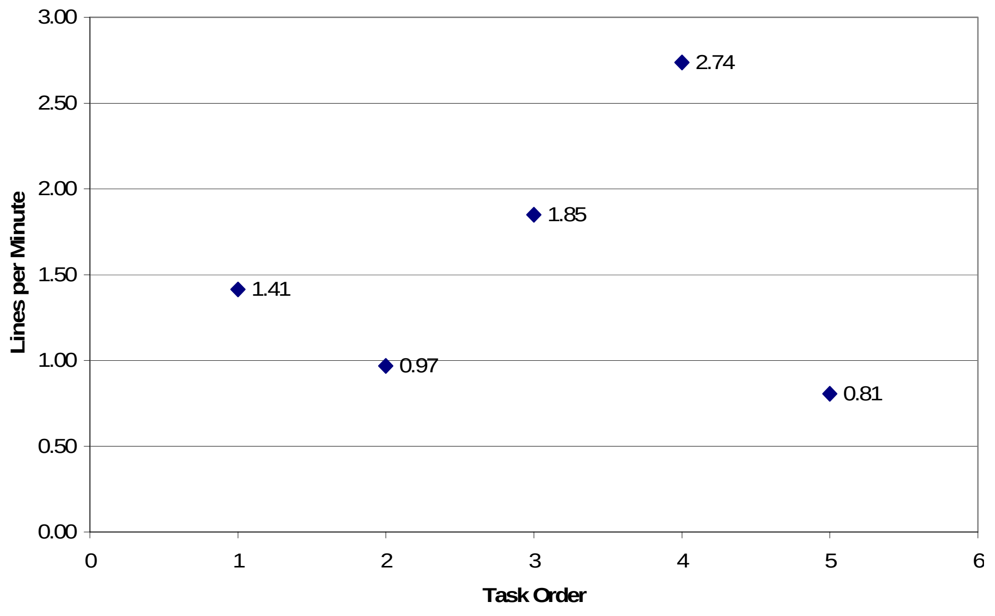

Figure 7: Java Productivity