An empirical study of software development productivity in C and C++[1]

Ingunn Myrtveit*, Erik Stensrud*

*) Norwegian School of Management - BI

Abstract

The object-oriented language C++ is claimed to be a better C. We analyzed data extracted from a database of 90 commercial, industrial software projects developing client-server type business applications in C++ and C, respectively. We investigated whether C++ improved the productivity of software development. We did not find any empirical evidence of improved productivity. This is in line with previous studies that to date have found no convincing empirical support in favor of object-oriented development.

INTRODUCTION

Software development is labor-intensive. The main motivation for the continuous development of new software development technologies is presumably to develop software more cost-effectively. Object-oriented technology (OOT) is considered one of the major contributions to software engineering. There have been massive investments both in academia and in industry to develop OOT since the 1960'ies. More than 30 years after Simula67 appeared, we expect to see a return on this investment in terms of higher productivity on OOT projects than on non-OOT projects. In this study, we empirically investigate whether the object-oriented language C++ has led to improved productivity compared with its predecessor C.

It is the commonly held view, and widely claimed, that object-oriented technology (OOT) improves the productivity and quality of software projects (e.g. Jacobson et al. 1992; Madsen et al. 1993; Guerraoui et al. 1996), especially of large, complex software (Moser and Nierstrasz 1996; Benett et al. 2002, ch. 3.5; Booch 1991; Martin and Odell 1995; Bruegge and Dutoit 2004). Some even consider OOT a revolution (Cox 1986). As for industry, we also observe that it has adopted OOT, and that C++ seems to have replaced C. C++ is furthermore claimed to be a "better C" (Stroustrup 1986).

Software systems are generally complex conceptual structures, and it is claimed that OOT better handles this complexity. The main reason for the belief in the superiority of OOT is that its concepts are closer to human thinking and natural languages (Mathiassen et al. 1993). The main OO concepts are objects, classes and class hierarchies. As for classes, they improve abstraction (abstract data types) and are perceived to provide better modularization than procedures, that is, a class module may provide higher cohesion and lower coupling than a procedure module and thus increase reuse. Regarding class hierarchies, they further improve abstraction and generalization and thus further reduce

[1] *This paper was presented at the NIK-2008 conference. For more information, see* `//www.nik.no/`

conceptual complexity. The human brain usually handles complexity by classifying and generalizing. Class hierarchies are well known from botany and zoology. We would therefore expect OOT to add value in handling software complexity.

Dynamic binding poses new challenges to software testing (Briand et al. 1999), thus potentially hampering productivity. Also, the use of the inheritance mechanism may be difficult and therefore limited in practice (Cartwright and Shepperd 2000). It is also difficult to see how class modeling adds value to entity-relationship modeling when data storage is to be realized in a relational database management system. Furthermore, it is claimed that OO programming languages like C++ may be hard to learn (Hardgrave and Doke 2000). For others, it may be the transition from structured analysis and design to OO analysis and design (OOAD) that may constitute a radical and difficult change (Sircar et al. 2001).

Given the strong analytical advocacy of OOT and, at the same time, the lack of empirical evidence of its alleged benefits, more empirical studies are called for. In this study, we therefore investigate how OOT impacts productivity analyzing data from a database of 90 industrial, recent, commercial software development projects using C++ and C as the primary language, respectively. The majority of the projects have delivered business applications, e.g. Management Information Systems (MIS) and transaction applications. Specifically, we investigate whether we can reject the null hypothesis that the sample of C++ projects is drawn from the same population as the C projects in terms of productivity. We test the following null hypothesis:

1. H0: productivity(C++ projects) = productivity(C projects)

The paper is organized as follows. Section 0 presents studies that investigated how OOT impacts productivity in a commercial environment. Section 0 describes the data used in this study. Section 0 presents the research method and formalizes the research hypothesis. Section 0 presents the results. Section 0 discusses the validity of the results, and section 0 concludes.

PREVIOUS FINDINGS ON OOT PRODUCTIVITY

There are few studies of OOT in industrial software projects, and those that exist have several weaknesses, most importantly, they suffer from small sample sizes. Other studies have larger sample sizes but used students as subjects on toy projects rather than professional software developers on real projects.

There is no convincing quantitative evidence to claim that OOT has delivered (Briand et al. 1999; Potok et al. 1999; Jones 1994; Cartwright and Shepperd 2000). Most studies have investigated isolated object-oriented features which gives somewhat indirect insights into OOT itself (Cartwright and Shepperd 2000), and most studies designed experiments using students (Potok et al. 1999). For example, Zweben et al. (1995) investigated encapsulation (or information hiding, a black box approach (Dennis et al. 2002)), a central OOT feature, using students. Basili et al. (1996) analyzed the effect of reuse on eight medium-sized MIS systems developed by students applying OOT.

Only one previous study has investigated the overall impact of OOT on productivity on commercial software projects (Potok et al. 1999). They analyzed 19 software projects; 11 used OOT; 8 used non-object-oriented development techniques. They assumed a log-linear model and used the double-dummy variable approach to test the significance of OOT.

They did not find that object-oriented technology improved the productivity. They did unfortunately not report which programming languages were used.

Port and McArthur (1999) analyzed four projects. These projects used C/Java, C++, Ada83, and Ada95 programming languages; three projects used OOAD. Productivity was measured as unadjusted function points (UFP) per person-month. They found that the Java/C project had 20 times higher productivity than the Ada83 project, with the Ada95 and C++ projects in between. They concluded that OOT improved on productivity, and that reuse of software components was the main contributor. However, their study was a case study, and the types of applications developed seemed very different. Thus, it seems invalid to generalize from their study.

Houdek et al. (1999) conducted an experiment comparing structured and OO methods applied to embedded software systems. The results identified only minor differences in development time and quality of the developed systems.

Cartwright and Shepperd (2000) analyzed one object-oriented system, 133,000 lines of code (=roughly 1000 function points), developed with C++. They found little use of OOT constructs such as inheritance and polymorphism. To the extent that inheritance was used, it contributed to more fault-prone software, thus potentially impacting negatively on the productivity. This finding is contrary to common belief, but nevertheless in line with the study of Wood et al. (1998) who found that inheritance in OOT software may inhibit software maintenance.

Finally, the large IT consultancy, Andersen Consulting (now Accenture), found that inheritance was often counterproductive, resulting in fragile, hard-to-maintain code on commercial software, not better than the Cobol programs they were replacing (Adamczyk and Moldauer 1995).

There have been published several studies on productivity of software projects in general, for example in *Management Science*, but none on OOT productivity (Banker and Kauffman 2004).

DATA

The ISBSG (International Software Benchmarking Standards Group) database Release 9 is, to our knowledge, the largest database of software projects (ISBSG 2005). It contains 3024 software projects. Projects have been submitted from 20 different countries. The major contributors are Japan (28%), the United States (26%), and Australia (24%). Major organization types are communications (15%), insurance (12%), banking (7%), manufacturing (6%), government (6%), and business services (6%). Major business areas are telecommunications (26%), banking (13%), insurance (13%), finance (9%), manufacturing (8%), accounting (5%), sales and marketing (4%), and engineering (4%). The applications are mainly business applications using a client-server architecture: Management Information Systems (18%), transaction/production systems (40%).

C/C++ projects account for 14% of the data. The projects were implemented between 1990 and 2003. The software size is measured in Function Points. The sample used in this study contains 90 observations (see Table 1). 52 were developed with C as the primary language and 38 with C++ as the primary language. The C and C++ projects are comparable as they are of similar size and have a similar mix of application types (MIS, Transaction Systems, Office Information Systems, and Electronic Data Interchange). Furthermore, they are all New Development projects.

Table 1. Descriptive statistics

Variable	N	Mean	Median	StDev	Min	Max
C and C++						
Function Points	90	602	394	625	25	3354
Work Effort (hours)	90	6035	3040	7379	62	34560
C++						
Function Points	38	557	364	507	59	2529
Work Effort (hours)	38	5611	3431	5446	263	22920
Implementation Year	37	1998	1999	2	1994	2002
C						
Function Points	52	627	470	703	25	3354
Work Effort (hours)	52	6345	2467	8555	62	34560
Implementation Year	49	1998	1999	3	1990	2003

METHOD

We test hypotheses using regression analysis with dummy variables. We assume that the relationship between labor, y, and software size, x, is given by the following, commonly used, multiplicative model:

$$y = e^{a} x^{b} e^{u} \qquad (1)$$

where a and b are coefficients, and $u \sim N(0, \sigma^2)$ is the stochastic error term. The coefficients (parameters) are estimated with linear least squares regression applied to the loglinear form of (1):

$$\ln y = a + b \ln x \; + u \qquad (2)$$

To test hypothesis 1, we applied (2) adding a single dummy variable:

$$\ln y = a + b \ln x \; + \sum a_i D_i + u \qquad (3)$$

The single dummy approach assumes that the slope of model (2), b, is constant, and allows only the intercepts, a_i, to vary. The a_i's are the dummy parameters to be estimated. D_i is a dummy variable for programming language i, C++ and C, respectively. For an account of the dummy variable approach, consult an econometrics textbook, for example Gujarati (1995, p. 512.)

Note, if $a_i < a$, the productivity of language i is higher than the baseline productivity. In the following sub-sections, we discuss the model specification and briefly present the Function Point metric.

We empirically examined the model specification, i.e. that the loglinear specification was reasonable for our sample, including linearity, homoscedasticity, outliers, and highly influential observations. Also, the slope coefficients of the two subsamples were almost identical, confirming the visual inspection.

1.1 Measuring software size and complexity with Function Points

Function Points (Albrecht and Gaffney, 1983; IFPUG, 1999) consist of five primitives: External Inputs, External Outputs, External Inquiries, Logical Internal Files, and External Interface Files. The primitives are classified into Low, Average, and Complex, then weighted based on the classification, and the weighted counts are added to a single unadjusted function point (UFP) total. UFP is then adjusted for processing complexity using 14 complexity factors to give the final adjusted function points (AFP). The classification into Low, Average, and High is based on counting the number of data element types (DET) and the number of file types referenced (FTR) that a transaction inputs to, or outputs from, the database, that is, the primitives are sized based on the number data movements.

In this study, we used AFP as the independent variable.

RESULTS

The results suggest that the loglinear model is a reasonable model. This is supported by the fairly high R-sq values in Table 2. From Table 2, we also observe that the regression coefficients of C and C++ are almost equal.

The single dummy results using equation (3) suggest that C++ projects are not more productive than C projects, and that we cannot reject the null hypothesis that they are equally productive (p=0.645 for a(C++), Table 3).

Table 2. Loglinear regression results

Language	N	a	b	R-sq%	S
C	52	3,24556	0,796668	44	1,09
C++	38	3,40123	0,787274	36	0,89

Table 3. Single dummy regression – C vs. C++

Predictor	Coef	SE Coef	P
a	3.26	0.61	0.000
b	0.79	0.10	0.000
a(C++)	0.10	0.22	0.645

One explanation of the lack of positive findings in favor of OOT is that the applications in this sample were client-server type business applications where the client typically is developed in C/C++ whereas the server is developed in SQL which is a non-OOT. The development effort on the server side could be of the same order of magnitude as the client development effort and thus may mask the actual productivity benefits of C++. Furthermore, the developers may suffer from having to cope with an object model on the client side and a non-object model on the server side. However, client-server development using several languages is perfectly normal and representative for real life software projects, in particular of business applications.

EXTERNAL VALIDITY

The applications investigated were mainly MIS and transaction/production systems (60%). The results ought therefore to be valid for MIS and transaction/production

software. However, the impact of C/C++ on the development productivity of other types of systems, e.g. scientific and engineering software (e.g. CAD software) may be different. For example, CAD software models physical objects, often a large number of different object types that lend themselves to modeling using "part-of" and "kind-of" relationships, and it may therefore better lend itself to C++.

CONCLUSION AND FURTHER WORK

In this study, we investigated whether development in C++ is more productive than development in C. We analyzed data extracted from a database containing commercial, software development projects completed in the period 1990-2004. The projects developed mainly client-server type, business applications like MIS and transaction systems. We found no empirical evidence that C++ is a more productive C.

The implications for managers and CIO's are that the choice of C++ vs. C is not obvious. In general, the procedural languages may be easier to learn and use, so, it may take both learning time and a very skilled workforce to reap the benefits of the more advanced features in C++.

Topics for further study include investigating in more depth why no studies have found convincing, empirical evidence in favor of OOT; further empirical analysis to investigate if C++ is more productive for large, more complex projects; further empirical analysis to investigate if C++ is more productive for enhancement projects than for initial development; last, but not least, evolve the existing theories on OOT.

REFERENCES

Adamczyk, J and T Moldauer (1995) "Inheritance and reusable objects? Yeah, right." *Andersen Consulting LLP*, Northbrook, IL.

Albrecht, AJ and SH Gaffney (1983), "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation". *IEEE Transactions of Software Engineering*. vol. 9, no. 6, pp. 639-648.

Banker, RD, and RJ Kauffman (2004), "The evolution of research on information systems: A fiftieth-year survey of the literature in Management Science", *Management Science*, vol. 50, no. 3, pp. 281-298.

Basili, V., Briand, L., and Melo, W. (1996), "How reuse influences productivity in object-oriented systems", *Communications of the ACM*, vol. 39, no. 10, pp. 104–116.

S Benett, S McRobb, R Farmer (2002), "Object-oriented systems analysis and design using UML – 2nd ed.", London: McGraw-Hill.

Booch, G (1991), "Object-oriented design with applications", Reading MA: Benjamin/Cummings.

Briand, LC, E Arisholm, S Counsell, F Houdek, and P Thevenod-Fosse (1999), "Empirical studies of object-oriented artifacts, methods, and processes: state of the art and future directions", *Empirical Software Engineering*, vol. 4, no. 4, pp. 387-404.

Bruegge, B and AH Dutoit (2004), "Object-oriented software engineering using UML, patterns, and JavaTM – 2nd ed.", Upper Saddle River, NJ: Pearson Prentice Hall.

Cartwright, M and MJ Shepperd (2000), "An empirical investigation of an object-oriented software system", *IEEE Trans. Software Eng*., vol. 26, no. 8, pp. 786-796.

Cox, B (1986), "Object-oriented programming – an evolutionary approach", Reading, MA: Addison-Wesley.

Dennis, A, BH Wixon, D Tegarden (2002), "Systems analysis & design – An object-oriented approach with UML", Wiley.

Guerraoui, R et al. (1996), "Strategic directions in object-oriented programming", *ACM Computing Surveys*, vol. 28, no. 4, pp. 691-700.

Gujarati, D.N. (1995), "Basic Econometrics", 3rd ed., New York: McGrawHill.

Hardgrave, BC and ER Doke (2000), "Cobol in an object-oriented world: a learning perspective", *IEEE Software*, vol. 17, no. 2, pp. 26-29.

Houdek, F., Ernst, D., and Schwinn, T. (1999), "Comparing structured and object-oriented methods for embedded systems: a controlled experiment". *ICSE'99 Workshop on Empirical Studies of Software Development and Evolution* (ESSDE). Los Angeles, pp. 75–79.

Q Hu (1997), "Evaluating alternative software production functions", *IEEE Transactions on Software Engineering*, vol. 23, no. 6, pp. 379-387.

IFPUG - International Function Point User Group (1999), "IFPUG Function Point Analysis Manual 4.1", www.ifpug.com.

ISBSG Estimating, Benchmarking and Research Suite Release 9, www.isbsg.org. Accessed 1 June 2005.

Jacobson, I, M Christerson, P Jonsson, and G Övergaard (1992), "Object-oriented software engineering – A use case driven approach", Wokingham, England: Addison-Wesley.

Jones, C (1994), "Gaps in the object-oriented paradigm", *IEEE Computer*, vol. 27, no.6, pp. 90-91.

Madsen, OL, B Møller-Pedersen, and K Nygaard (1993), "Object-oriented programming in the Beta programming language", Wokingham, England: Addison-Wesley.

Martin, J and JJ Odell (1995), "Object-oriented methods – a foundation", Englewood Cliffs NJ: Prentice Hall.

Mathiassen, L, A Munk-Madsen, PA Nielsen, J Stage (1993), "Object-oriented analysis" (in Danish), Aalborg, Denmark: Marko.

Moser, S and O Nierstrasz (1996), "The effect of object-oriented frameworks on developer productivity", *IEEE Computer*, pp. 45-51.

Potok, TE, M Vouk, and A Rindos (1999), "Productivity analysis of object-oriented software developed in a commercial environment", *Software – Practice and Experience*, vol. 29, no. 10, pp. 833-847.

Sircar, S, SP Nerur, R Mahapatra (2001), "Revolution or evolution? A comparison of object-oriented and structured systems development methods", *MIS Quarterly*, vol. 25, no. 4, pp. 457-471.

Stroustrup, B (1986), "The C++ programming language", Reading, MA: Addison-Wesley.

Zweben, SH, SH Edwards, BW Weide, and JE Hollingsworth (1995), "The effects of layering and encapsulation on software development cost and quality", *IEEE Transactions on Software Engineering*, vol. 21, pp. 200-208.