THE COMBINED EFFECTS OF THE USE OF APPLICATION GENERATORS AND CHOICE OF DEVELOPMENT METHOD ON PRODUCTIVITY

Marikka Pihlaja

Helsinki School of Economics, MarikkaPihlaja@emailaddressnotknown

Follow this and additional works at: http://aisel.aisnet.org/sjis

Recommended Citation
Available at: http://aisel.aisnet.org/sjis/vol2/iss1/5

This material is brought to you by the Journals at AIS Electronic Library (AISeL). It has been accepted for inclusion in Scandinavian Journal of Information Systems by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.
THE COMBINED EFFECTS OF THE USE OF APPLICATION GENERATORS AND CHOICE OF DEVELOPMENT METHOD ON PRODUCTIVITY

MARIIKA PIHLAJA
Business Information Systems at the Helsinki School of Economics
Runeberginkatu 14-16, SF-00100 Helsinki, Finland

Abstract

The promised productivity improvements resulting from using application generators have been described by several authors. Some experiments have also been conducted, but they differ substantially in their research settings and conduct of the experiments. This paper examines these differences by collating the main factors affecting the results of the experiments into an overall framework. Previous research efforts are then analyzed according to this framework.

Using the developed framework, experiments were conducted to evaluate the combined effects of the use of application generators and of development strategy on productivity. The results of these laboratory cases showed that, compared with third generation languages, 30 to 1 productivity improvements are possible in the development of medium-sized administrative systems, when application generators are used together with a prototyping strategy. In addition, programmers experienced more satisfaction in prototyping projects than in linear projects.

Keywords: application generator, prototyping.
1 Introduction

A severe problem facing data processing departments is the backlog of applications. A number of studies have shown this backlog to involve several years work (Alloway & Quillard 1983, Sumner & Benson 1988). Unfortunately the maintenance of existing software takes up most of the available DP resources, leaving only a small amount of effort for developing new applications. As a consequence many useful new software products are not developed.

Methods of coping with this shortage of DP resources include the use of ready-made software packages and end user application development. But DP departments are currently still not able to satisfy the demand. Application development productivity has become an issue affecting the profitability and competitiveness of the organization.

Technological change and the management of new technology are the two basic means to improve productivity (Jeffery 1987). Application development is still highly labour intensive, and since the costs of hardware resources are dropping while labour costs keep rising, productivity can be improved by trading developer labour for more intelligent software (Boar 1984, Boehm & Papaccio 1988). Repetitive and monotonous work can be automated or totally eliminated by using software tools (Boehm & Papaccio 1988). According to Boehm (Boehm 1981, Boehm & Papaccio 1988) productivity can be further improved by re-using components, building simpler products and prototyping. In addition to these methods, management should pay special attention to human factors such as staffing and motivation (Boehm & Papaccio 1988).

Significant productivity improvements have been made with the adoption of fourth generation languages for writing applications. According to Martin (1985) they require less human effort, less skill and less time, because they are easy to use, developer friendly and have various automatic features. Furthermore, the amount of reworking is usually reduced, because the code contains fewer errors and applications are easy and quick to change.

In this paper we concentrate on the productivity of application generators when used by DP professionals to develop administrative systems. According to several case reports, by using these tools DP professionals have been able to significantly improve application development productivity (Holtz 1979, Sumner & Benson 1988). Some experiments have been conducted to find out how productive these application generators really are (Harel & McLean 1985, Kuva 1988, Misra & Jalics 1988, Verner & Tate 1988, Rudolph 1983a). The findings vary considerably providing little evidence to support Martin's (1985) claims for order of magnitude productivity improvements. However, when these experiments were studied more closely, we found that they differed substantially in their research settings.

In this paper, a framework is first proposed, showing the main factors affecting productivity. Next, this framework is used to review the productivity experiments described in the literature. Finally, the results are presented of some empirical
experiments which were conducted to evaluate the combined effects of application generator and development strategy on productivity.

2 Measures of Productivity

Productivity means the amount of goods or services that can be produced for a given input of labour or expense (Jones, 1986c). Productivity in application development is taken as the resulting application, measured in number of lines of code or function points, divided by work effort. Although quality is a very important aspect of the product, in productivity experiments the work product includes only the size and complexity of the application. Further, despite the fact that maintenance work takes up half (Muukari et al. 1983) or even 80% (Martin 1985) of DP-resources, none of the productivity studies cover the maintenance phase. They either study the development life cycle from the preliminary study to acceptance of the product, or only a limited part of the life cycle—the programming and testing phase. In order to make the difference between these two perspectives clear, the former is referred to as development and the latter as programming.

2.1 Measuring the Product

A count of the number of lines of code produced is the most frequently used measure of product size. The advantage of this measure is its simplicity and cost effectiveness. But, due to the large variety of ways of counting the results can sometimes differ as much as 500% (Jones 1986c). Thus, if the line counting system is not known, the method is not suitable for comparing the results of different experiments. Nor is it suitable for comparing the productivity of different generations of programming languages, because it penalizes high-quality languages (Jones 1986b). It has also been noted that good programmers write more structured, more compact code in less time than poor programmers (Jeffery & Lawrence 1979). Thus measures of numbers of lines of code fail to show the productivity differences between programmers. Still another disadvantage is that the complexity of the program is not reflected in the number of lines of code (Jones 1986c).

Function point analysis (FPA) is a method of characterizing the size and complexity of applications. The product is measured by counting the number of external user inputs, outputs, inquiries and master files produced by the development project. These counts are weighted with numbers designed to reflect their functional value to the customer and their complexity level (Albrecht 1979, Albrecht & Gaffney 1983). The sum of the weighted counts gives the total number of raw function points. The number of raw function points is finally adjusted according to the level of influence of 14 adjustment factors. This adjustment can modify the number of raw function points by up to 35%.
The analysis is based on the user’s external view of a program rather than on the inspection of source code. The weakness of function point analysis is that it underestimates applications of high complexity (Symons 1988). Another disadvantage is that function points cannot be counted automatically; personal judgement has to be used in determining the complexity and number of functions. This leaves room for subjectivity and error, the upper limit of the error range is 30%. However, the error is less than 10% when the application measured is in operation or has detailed design specifications (Rudolph 1983a).

Function point analysis is considered to be independent of the programming language (Rudolph 1983a) and development methodology (Navlakha 1986). Although technological factors have a minor effect on the function point measure, it can be used to compare the productivity effects of different factors and different generations of programming language.

### 2.2 Measuring Effort

Effort is measured in terms of ‘time spent’ rather than of “monetary units that are unequal between different organizations and even more unequal between different countries (Rudolph, 1983a)”. Furthermore, in many situations the time taken is more critical than the financial costs of the application development (Rudolph 1983a). Most productivity experiments have measured effort in terms of work hours recorded by the programmers during the development task.

### 2.3 A Selected Measure for Productivity

In this study the work product is measured in function points covering the size and complexity of the product. Because application development is highly labour intensive, the human resources are usually the limiting element in DP work. We measure work effort in terms of work hours to avoid the difficulties of monetary units.

The results of those studies which used only the number of lines of code as a measure (Boehm et al. 1984, Harel & McLean, 1985, Misra & Jalics 1988), were transformed into function points. This was done using Jones’ (1986c) notion of the approximate number of source statements required to code one function point in different languages (Cobol - 106 LOC/FPA, Pascal - 91 LOC/FPA).

### 3 Framework for Productivity Factors

In order to be able to improve productivity it is essential to recognize the factors affecting it. Indeed, several factors influencing the productivity of the application development process have been specified (Jones 1986c, Benbasat & Vessey, 1980, Chrysler 1978, Boehm 1981). In two studies (Harel & McLean 1985, Kuvaja 1988) Chrysler’s (1978) framework was applied to the reporting of the research settings. Our framework is primarily based on that of Benbasat and Vessey (1980),
in which software engineering characteristics were added to the factors used in Chrysler's framework (programming mode, organizational operations, computer hardware, source language, programmer, programming problem). Unfortunately most of the productivity studies were not carried out in real organizations but in laboratory settings, or the information in the research reports on user and developer organizations was inadequate. Thus the characteristics of the organization could not be studied in this paper. Also, programming mode was left out of our framework. There are studies showing both a productivity advantage and no advantage from the programming in on-line environments. But it appears that in the working environment other performance effects eliminate this factor (Jeffery 1987). Our framework is shown in Figure 1. The factors included are: programming language, size and complexity of the application, developers' experience, development strategy and hardware environment. The Appendix shows the productivity experiments in terms of our framework.

Figure 1: Framework for analyzing productivity factors.

3.1 Programming Language

Increased productivity and ease of use are characteristics of higher generations of programming language. For this paper fourth generation languages and application generators are of particular interest.

Fourth generation languages (4GLs) were introduced towards the end of the 70's. They permit applications to be generated with fewer lines of code and less work effort than would be needed with third generation languages (e.g. COBOL, PL/1, and FORTRAN). Maintenance costs are expected to decrease, because changes in the applications are easy and quick to make (Martin 1985). In addition to employing sequential statements, as do third generation languages, fourth generation tools use a range of other mechanisms, such as filling in forms and panels, screen interaction and computer-aided graphics. On the other hand, the use of fourth generation tools takes up a considerable amount of CPU-time. Also the final products made with 4GLs usually require more CPU-time than products made with third generation languages. The variations in the efficiency of application generators are vast—some being more efficient and others much less efficient than conventional languages (Matos & Julics 1989).

Application generators (e.g. Clarion, Focus, Ingres, Linc, Paradox) belong to the category of fourth generation tools. There are various definitions for appli-
cation generators. In general they are defined as interactive, integrated software systems intended to improve information systems development and maintenance by automating parts of the system development life cycle (Kuvaja 1988). They consist of a number of features such as a screen generator, report generator, database management system and a non-procedural language. Yet, they vary in their power and capabilities. In this paper we study only those application generators which enable creation of a complete application.

By $3{\frac{1}{2}}$GL we mean modern third generation environments which can take advantage of one or more 4GL features. The main functional difference between $3{\frac{1}{2}}$GLs (e.g. JSP-Cobol, Fortran with a relational DBMS including SQL, and a screen-definition system) and application generators is—as Kuvaja (1988) states—the integration of the different functions. In application generators all the functions are integrated: this gives them powerful new possibilities such as prototyping and efficient application modification aids. This differentiates them from modern third generation development environments.

### 3.2 Application Size and Complexity

The development of large systems is in general more expensive and more time consuming than that of small systems. Yet it is difficult to distinguish the productivity effects of size from other factors. Usually as the application grows in size, it also becomes more complex, and the number of interfaces and the need for communication between project participants increases. According to Jones (1986c) the effort used for activities like managing, documenting and planning grows quicker than the effort used for other activities. Thus the percentage of non-productive work increases causing productivity to decrease.

Experiments studying the effect of application size on productivity have mainly been carried out using third generation languages. Chrysler (1978) found the number of input files, control breaks and totals, input edits and input fields to have a statistically significant impact on productivity, but Jeffery and Lawrence (1979) found no evidence of this. Some results show that productivity increases as the number of code lines increases (Vessey 1986, Jeffery & Lawrence 1979). On the other hand productivity has been found to decrease as size in function points increases (Albrecht 1979, Behrens 1983). Differences in the measures used may explain some of the inconsistency of the results. Jeffery & Lawrence (1979) suggested another explanation for these confusing results: The relationship between application size and productivity is more complex than expected. It might be that it is approximately linear over small intervals only.

The fact that an application development process always includes some fixed overhead effort might explain differences in productivity. In small applications the total effort may consist mostly of this overhead, giving low productivity figures. As the application grows in size, the overhead is spread over a greater work product, and productivity increases. But, as the application grows even more, more non-productive work is required, resulting in decreased productivity.
Figure 2: The productivity figures for different language generations with respect to the size of the application. (Rudolph 1983a and 1983b, Albrecht 1979, Boehm et al. 1984, Harel & McLean 1985, Kuvaja 1979, Misra & Jalics 1988, Verner & Tate 1988).
Figure 2 shows the productivity figures for different language generations (3GL, 3 1/2 GL and 4GL) with respect to the size of the application (Rudolph 1983a, 1983b, Albrecht 1979, Boehm et al 1984, Harel & McLean 1985, Kuvaja 1988, Misra & Jalies 1988, Verner & Tate 1988). Both programming productivity and development productivity are shown in the figure. The productivity of application generators (4GLs) is better than that of traditional third generation languages. The difference between 3 1/2 GL and 4GL is not so clear-cut, but in general 4GLs were more productive, especially in terms of development productivity.

Application Size Categories

Jones (1986a) divided applications into eight size ranges. The size ranges were measured in number of lines of code. Even though it was not actually stated by Jones, it was assumed that the code lines refer to Cobol code (Kuvaja 1988). Thus we calculated corresponding size ranges in function points, using a ratio of 106 Cobol code lines to one function point. Because of the small number of experiments (8 experiments, 71 applications) and the fact that there were no studies on large or super-large applications, we were compelled to use a condensed categorization as follows:

0 - 20  FPA small
20 - 300  FPA medium
300 - 4800  FPA large

The following statements are based on Jones' (1986a) claims for the specific tools and methods giving the best productivity depending on the size range. In the 0 - 20 function point category the best productivity has been attained with spreadsheet languages and database query languages. In the 20 - 300 function point category the most productive way to develop programs is to use application generators, automated design tools and good programming languages.

When the application is larger than 300 function points, the best results can be achieved by using a combination of good requirements, design, prototyping and defect removal. Application generators usually reduce the need for human resources at the expense of hardware resources. If application generators are used for systems with high transaction volumes or complexity, close attention must be paid to machine performance. Most 4GLs are not designed for heavy-duty computing (Martin, 1985). Thus, as the application gets very large and the performance constraints become more severe, languages closer to machine language may be a better choice than 4GLs. Jones (1986a) recommends utilization of re-usable code modules and libraries of standard design.

The research experiments show rather low productivity with 3 1/2 GLs and 4GLs when the application size is small. As the size grows a little, productivity increases. But it begins to decrease again as the application grows even larger. This phenomenon is in accordance with our assumption of the effects on productivity of overhead effort in relation to size and complexity.
Figure 3: *Average programming productivity of languages according to size category.* (Harel & McLean 1985, Boehm et al. 1984, Misra & Jalics 1988, Kuwaja 1988, Verner & Tate 1988).

Figure 4: *Average development productivity of languages in size categories.* (Boehm et al. 1984, Rudolph 1983a, Albrecht 1979, Verner & Tate 1988).
If the effects of application size and complexity are examined more closely by dividing the applications into size categories, this phenomenon is easy to observe (Figures 3 and 4). The productivity advantages of 4GLs over 3GLs, in programming and development, are greatest in medium-sized applications (20 - 300 FPA). It also shows that the productivity of application generators decreases noticeably when the application gets large. But, we would like to point out the productivity differences between programming and development. In the range of 20 - 300 FPA the productivity ratio between 31/2GLs and 3GLs seems to be the same, 6:1, regardless of the life cycle phases examined. But with application generators the productivity is even better when preliminary work and design are also studied. These development productivity results are based mainly on the findings of Rudolph (1983a). The results of Rudolph's studies are noticeably better than those of other studies. Rudolph attributes this to the experience of the programmers in the problem area, and to the small project groups. Unfortunately there is not enough information about other factors, such as development strategy, to analyze their impact on these results.

In addition, the productivity of 31/2GLs does not seem to drop as application size increases. On the contrary, the ratio with traditional languages improves further. The results support Jones's claim about tools giving the best productivity in different size categories. Unfortunately there are no results for the productivity of application generators in the size range 20 - 75 FPA. This is the category in which, according to Jones, application generators have proved to be the winners in productivity. In any case, application generators tend to be at their best in medium-sized applications, but as the size increases further, modern third generation environments challenge their superiority.

3.3 Developers' Experience

The productivity impact of experience has been studied by several researchers. Again the results are contradictory. Programmer's experience has been found to improve productivity (Chrysler 1978, Cheney 1984, Vessey 1986), while others have found no such relationship (Behrens 1983, Jeffery & Lawrence, 1979 and 1985). In any case, it seems to be difficult to examine and quantify human factors. Personal characteristics cannot be totally eliminated. They along with the different measures applied, obscure the research findings.

Jones (1986c) divided programmers' experience into experience in the programming language (tool), the problem area and in structured programming. According to Jones, a programmer experienced in all these subcategories is about 20 times more productive than an inexperienced programmer. The difference between the best and worst-case situation would be even more extreme. Martin (1985) claims that a good programmer can easily achieve 10 times the productivity of a poor programmer with any language, but with 4GLs the difference can rise to 100 to 1. In addition, it is easier to achieve a higher level of skill with a 4GL than with conventional languages, notes Martin (1985).
Even though the effects of experience have been observed in almost every experiment, they are not explicitly reported, making it difficult to draw any conclusions about the impact of experience on productivity. Tentatively, it seems that the larger the application, the more important experience becomes. In programming small applications with application generators experts are only somewhat more productive than novices, in fact, in the small-applications category the novices seem to benefit more than experts from using application generators instead of third generation languages. But, as the application grows in size, experienced programmers achieve about twice the productivity of novices. Both experienced and inexperienced programmers are about 50% more productive when they are using application generators instead of 3.5-languages.

Application generators seem to be most productive when experienced programmers are using them to program a medium-sized application—the ratio between 4GLs and 3GLs is over ten to one.

3.4 Development Strategy

Successful project management and control is an important factor in application development (Jeffery 1987). Every development life cycle consists of preliminary work, design, implementation, and installation and maintenance. Depending on the approach employed these main activities can be divided into more precise phases. The main categories of application development are linear strategy and prototyping strategy (Davis 1982).

Linear and Prototyping Strategies

Conventional approaches, like the Cascade model (Birrel & Ould 1985), manage the process with the help of rigorous specification and control. The process is carried out in a linear fashion: A phase is started only after the previous phase has been completed. Returning to a previous phase is avoided.

The basic idea of prototyping is to generate a working model of the application instead of writing detailed specifications for it. According to Naumann & Jenkins (1982) prototyping neither requires nor permits the use of prolonged rigid specifications during the development process. The prototype is generated quickly and is tested by the user. Any undesirable or missing features identified must then be corrected. Several iterations will undoubtedly be required before the application is finished.

Recommendations and Praxis

Burns & Dennis (1982) introduced a framework for suitable development strategies depending on the complexity and uncertainty of the project. A conventional linear strategy should only be used with projects of high complexity and low uncertainty. When both complexity and uncertainty are high, a mixed strategy of
linear and prototyping approaches is more appropriate. Otherwise prototyping is recommended.

Martin (1985) illustrates (Figure 5) the productivity improvement gained with fourth generation languages when a linear approach is used. Productivity is improved only in the programming and testing phase. He argues that if application generators are to be used most effectively, the development strategy should also be adapted to the technology used. Application generators are at their best when a prototyping approach is used. The distribution of work effort into different phases changes, and the total effort decreases (Figure 6).

![Figure 5: Use of application generators with linear strategy.](image)

![Figure 6: Use of application generators with prototyping strategy.](image)

However, the current systems development life cycle is usually not deliberately altered for application generators (Saarinen & Sääksjärvi 1989, Sumner & Benson 1988). In addition, Saarinen & Sääksjärvi (1989) found that in Finland the success of those projects which used application generators together with a linear approach was not better than average. However, projects using application generators with a prototyping approach were generally more successful. This does not mean that application generators cannot improve productivity by themselves, but that their impact might be even more significant when a prototyping method is used.

Although prototyping is the best strategy for 4GLs, most experiments have only studied the impact of application generators on productivity when using a linear development approach. Figure 7 shows the development strategies used in the experiments. If we compare the research experiments with Burns' and Dennis' recommendations, we note that most of the experiments used a development strategy suitable only for projects with high complexity and low uncertainty. Harel & McLean (1985) defined their test projects as 'simple or not so simple'. Also, the specifications for the test application used by Misra & Jalics (1988) gave the impression of being of low complexity. Thus, according to Burns and Dennis the prototyping approach should have been used. For the project of Verner & Tate (1988) either the prototype or a mixed strategy is recommended. Due to the hypothetical application of Kuvaja's (1988) experiment it is difficult
to determine a suitable approach. It might have been better to use a prototyping strategy, though.

On the other hand, prototyping is encouraged only in situations where suitable technology and tools are available. If the program is developed with a conventional language, a mixed strategy is recommended instead of prototyping (Burns & Dennis, 1982). Even though the experiment of Boehm et al. (1984) did not follow this advice, it provides evidence for significant productivity effects of prototyping: prototyping groups needed about 45% less time than linear groups to develop the same application with Pascal.

![Diagram]

Figure 7: Development methods used in the experiments and recommendations of Burns & Dennis (1982).

If a prototyping approach can noticeably improve the productivity of conventional languages, the improvement in productivity would undoubtedly be even more outstanding with application generators. In Verner’s and Tate’s experiment a group of programmers inexperienced in application generator, developed a large application (863 FPA) using a prototyping approach. The productivity was compared with the hypothetical productivity of the project implemented with Cobol: The productivity ratio was estimated to be about 4:1.

In addition, Verner and Tate noted that the use of the application generator together with prototyping reduced the work effort, mainly in the implementation phase but also in the design phase. Thus, in order to study the real productivity effects of application generators, examination of programming productivity is inadequate. Attention should be paid to the development strategy and to the whole application development life cycle, from the preliminary study to the completion of the application, or even to the maintenance phase. Apart from the study of Verner & Tate (1988), only Rudolph has examined the development productivity of application generators (1983a). All the other studies of application generators have examined only programming productivity. They used prespecified applications with a linear strategy, so that in contrast to prototype projects, no productivity improvements are possible in the design phase. Moreover, it is
possible that the productivity of the plain programming and testing phase is better when using application generators with a prototyping approach than with a linear approach.

3.5 Hardware Environment

The effects of the hardware environment on productivity have not received much attention in the studies. It is, however, assumed that productivity improves as the environment becomes simpler (Thadhani 1984). Thus, in microcomputer environments productivity should be better than in mainframe or minicomputer environments.

Hardware response time does have a certain effect on programmer's productivity. Productivity has been shown to improve with 60% (measured in FPAs) simply by cutting response time from 2.22 to 0.84 seconds (Lambert 1984).

3.6 Summary of the Framework Analysis

The results of the productivity studies reported in the literature indicate that the programming language and the application size are the most important productivity factors. Regardless of the size of the application, productivity is improved when higher generations of programming language are used. Figure 8 shows a rough estimate of how size affects the productivity of different language generations; it appears that for small applications productivity is relatively low, for medium-sized applications it is at its best, and it then drops off again.

The results indicate that the effects of experience increase as the application grows in size. An expert creating a medium-sized application will most probably achieve the best productivity.

![Graph showing the productivity of 3GL, 3+GL, and 4GL languages with application size.](image)

Figure 8: *Estimate of the effect of application size on the productivity of different language generations.*
On the other hand, no reliable diagnosis could be made of the effect of the development strategy or hardware environment on productivity.

A review of the literature made it quite obvious that there is still a lot of research to be done in order to get a more complete picture of the productivity effects of application generators on DP work. Further it is clear, that we must not forget to take into consideration the whole application life cycle, as well as the factors affecting productivity, when comparing different productivity results.

4 Empirical Experiments

Our aim was to examine the impact of the combined effects of the application generator and development strategy on productivity. This was because most productivity experiments have studied application development using only linear development strategy, and because application generators should be at their best in a prototyping approach.

4.1 Research Settings

We carried out two empirical experiments (Figure 9): a linear project and a prototype project.

In the linear project a novice programmer programmed an invented application using an application generator. A prespecified test application was used. The specifications were the same as were used in Kuvaja’s (1988) experiment. Thus, we could calibrate our measuring methods by reference to Kuvaja’s. In addition, because Kuvaja’s experiments were conducted in mainframe or minicomputer environments, and we used a microcomputer environment, we could investigate how the hardware environment affects the programming productivity of a novice programmer.

In the prototyping project two versions of an application were created as required by the Society of Forestry in Finland. The application was of low complexity, but did have some uncertainty associated with it. Ac-

<table>
<thead>
<tr>
<th>Projects</th>
<th>P</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of applications</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Life cycle</td>
<td>Preliminary work</td>
<td>Y</td>
</tr>
<tr>
<td>Design</td>
<td>Implementation</td>
<td>Y</td>
</tr>
<tr>
<td>Measurement</td>
<td>Output</td>
<td>X</td>
</tr>
<tr>
<td>~FPA</td>
<td>Lines of code</td>
<td>X</td>
</tr>
<tr>
<td>Input</td>
<td>Work months</td>
<td>X</td>
</tr>
<tr>
<td>Work hours</td>
<td>GL</td>
<td>X</td>
</tr>
<tr>
<td>Language</td>
<td>Modern</td>
<td>3</td>
</tr>
<tr>
<td>conventional</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Size</td>
<td>Small</td>
<td>0-20FPA</td>
</tr>
<tr>
<td>Medium</td>
<td>20-300FPA</td>
<td>2</td>
</tr>
<tr>
<td>Large</td>
<td>300-4800FPA</td>
<td>2</td>
</tr>
<tr>
<td>Experience</td>
<td>in language</td>
<td>N</td>
</tr>
<tr>
<td>in hardware env.</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>in problem area</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>in structured prog.</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Development method</td>
<td>Linear</td>
<td>X</td>
</tr>
<tr>
<td>Prototype</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Hardware environment</td>
<td>Mainframe</td>
<td>X</td>
</tr>
<tr>
<td>Minicomputer</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Microcomputer</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Application gen.</td>
<td>Ingres 5/02a PC</td>
<td>1</td>
</tr>
<tr>
<td>Paradox 2.01 PC</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Clarion Proff.</td>
<td>developer 2.0</td>
<td>1</td>
</tr>
</tbody>
</table>

where

P Prototype project
L Linear project
and
N Novice
E Experienced

Figure 9: Research setting for our experiments.
According to the recommendations of Burns & Dennis (1982), this was an ideal project for prototyping. In addition, the application was medium-sized—the size category in which application generators have shown maximum productivity.

The biggest difference from a linear project was that, in the prototype project, no formal written specifications were made—the developer designed the application while programming. Another important difference was that, in the prototype project, one person did everything from the preliminary work to completion of the application, whereas, in the linear project, the programmer only programmed and tested an application to meet the specifications set by another person(s).

The programmer in the linear project was also one of the developers in the prototyping project. Thus, personal characteristics should not affect the productivity difference between the two applications she produced. Because the programming language and the application area were completely different in these experiments, the fact that the same person performed both experiments should not bias the result. The experience of the programmers is presented in the Figure 10.

<table>
<thead>
<tr>
<th>EXPERIENCE</th>
<th>IN THE LANGUAGE</th>
<th>IN THE HARDWARE ENVIRONMENT</th>
<th>IN THE PROBLEM AREA</th>
<th>IN STRUCTURED PROGRAMMING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Project</td>
<td>The programmer had no former experience with the tool. Before experiment she first looked through a tutorial and some examples and then developed a small program by herself.</td>
<td>Programmer had used microcomputers for several years and was using them at work daily.</td>
<td>No experience in this problem area.</td>
<td>The programmer was a 4th year student of Business Information Systems at Helsinki School of Economics. Thus theory familiar, but little experience.</td>
</tr>
<tr>
<td>Prototype project</td>
<td>Both developers had developed one medium-sized application with the application generator used in the experiment.</td>
<td>Both developers had used microcomputers for several years and were using them daily at work.</td>
<td>The developers had together developed an application of the same type using another application generator. No other experience.</td>
<td>Both were 4th year students of Business Information Systems at Helsinki School of Economics. Thus theory familiar but little experience.</td>
</tr>
</tbody>
</table>

- Novice
- Expert
- Novices

Figure 10: Experiences in our experiments.

4.2 Measurements

The developers measured the size and complexity of the applications in function points. The application in the linear project was finally measured as having 171.8 corrected function points, which was very close to the mean value—171.3 FPA—of the estimates given by the programmers in Kuvaja’s experiment. Thus, we confirmed that our way of counting function points (we noticed that the function point estimate could vary easily 30%, which was more than claimed by Rudolph (1983a)) did not affect comparisons of the results of our and Kuvaja’s experiment. Using the same counting method the application in the prototype project was measured as having 133.6 corrected function points.

Working hours were recorded by the programmers during the test. In the first experiment the unit of measure was 5 minutes. In the second experiment
the programming and testing effort was recorded in minutes, the work needed for other activities was estimated in hours (Figure 11).

4.3 Findings

We measured programming productivity in both experiments. In the linear project the programming productivity was 1.02 function points per hour (including some design work, programming and testing). However, in the prototype project the programming productivity was 2.62 (Ingres) and 2.71 (Paradox) FPA/hour (including all the design work, programming and testing). In the prototype project we were also able to measure development productivity (productivity of all phases from preliminary work to completion of the application): with Ingres, the development productivity was 1.61 and with Paradox 1.64 function points per hour. Unfortunately, because we did not know how much time was spent on the preliminary work and the design phase in the linear project, we were unable to measure its productivity in the development process. In any case, productivity was significantly higher in the prototype projects. In comparison with the linear project programming productivity was doubled, and the productivity of the whole development life cycle exceeded that for the programming work in the linear project.

One reason for the much greater productivity in the prototype project was that, in the linear project, rigorous specifications forced the developer to program the application in a way that excluded the use of the features of the application.

<table>
<thead>
<tr>
<th>Life Cycle Phases</th>
<th>WORK EFFORT DISTRIBUTION OVER PHASES</th>
<th>Linear Project Clarion</th>
<th>Prototype project Ingres</th>
<th>Paradox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preliminary work</td>
<td>Unknown, because we created a hypothetical application according to specifications made at the Institute of Information Processing Science, University of Oulu, Finland.</td>
<td>Both developers took about 20 hours. In addition, the representatives of the association and the supervisor of the project spent about 12 hours in meetings and other activities associated with the project.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td>unknown. The specifications included processing logic, information types and databases and screen and report layouts. Programmer spent about 4 hours becoming familiar with the specifications.</td>
<td>27 hours</td>
<td>35.5 hours</td>
<td></td>
</tr>
<tr>
<td>Programming &amp; Testing</td>
<td>169 hours, which includes all activities after familiarization with specifications.</td>
<td>24 hours</td>
<td>14 hours</td>
<td></td>
</tr>
</tbody>
</table>

Figure 11: Work effort distribution over life cycle phases in empirical experiments.
Programming productivity of 4GL
Medium-sized application

OUR EXPERIMENTS

Novice using prototyping in microcomputer environment

Novice using linear approach in microcomputer environment

KUVAJA’S EXPERIMENT

Novice using linear approach in minicomputer environment

Experienced using linear approach in minicomputer environment

Experienced using linear approach in mainframe environment

Productivity (FPA/hour)

Figure 12: The productivity results of our and Kuvaja’s (1988) experiments.

generator. In contrast, in the prototype project, there were no restrictive specifications, and the developers could design and implement the application while getting the best out of the application generator they were using. This benefit was specially noticeable, since the user organization had no former experience in using computer technology in its administration. Thus, the developers were allowed to use their own judgement and creativity in implementing the application.

Another explanation for the productivity difference was that, in the linear project, it was sometimes difficult for the programmer to interpret the specifications. In addition, the specifications—as do all specifications—contained some errors. Thus, a considerable amount of work effort was spent in clarifying the specifications and correcting mistakes. In contrast, because the developer in the prototyping project was assigned to the project from the beginning, she had a better idea of what was to be developed. Moreover, no time had to be spent in making or comprehending formal written specifications.

Figure 12 shows a comparison of the results of Kuvaja’s and our experiments. The experience of our programmers should have been quite similar to the experience of novices in Kuvaja’s experiment. All were DP students in their 4th - 5th year and all were inexperienced in the application generator they were using. Their experience of structured programming was about the same. The differences in their experience of the hardware environment is not known. These results provide some evidence for better productivity in microcomputer environments than
in minicomputer or mainframe environments. A more important result is that
the productivity in the programming phase seems to be improved with prototyping.
The same novice programmer was able to more than double her productivity
by changing from a linear strategy to prototyping.Regardless of experience and
hardware environment, the productivity of application generators is about 2.4
times greater with prototyping than with a linear approach. In comparison with
the novice 3 1/2GL programmer in a mainframe environment (Kuvaja 1988), the
productivity ratio was 6:1. In comparison with the findings (Boehm et al. 1984)
on the traditional approach to programming applications—third generation lan-
guages with a linear approach—the productivity of application generators in mi-
crocomputers using prototyping is more than 20 times higher. These figures refer
only to medium-sized applications, and analyze only programming productivity.

Analyzing all the phases, from preliminary work to user acceptance of the
product, our experiment shows a productivity increase of about 30 times that of a linear project using third generation languages (Boehm et al 1984, Albrecht
1979, Rudolph 1983a). Further, a productivity ratio of 18:1 was recorded between
our prototyping project and an incremental project carried out by Verner & Tate
(1988). Our application was of medium size and was developed in a microcom-
puter environment, whereas that of Verner & Tate was large and was created in
a mainframe environment. This implies that application size and hardware envi-
ronment have a crucial effect on the productivity of application generators. The
best productivity can be achieved when a medium-sized application is prototyped
in a microcomputer environment.

5 Conclusions

The findings of this study support James Martin’s claims of productivity gains
of between 10:1 and 80:1 with application generators in comparison with third
generation languages. However, since only two empirical experiments were car-
rried out, the personal characteristics of the developers may bias the findings. An
improvement in the development process (i.e. preliminary work, design, program-
ning and testing) of 30 to 1 was achieved by novice programmers using applica-
tion generators with prototyping. In both experiments the applications were
medium-size administrative systems. As the productivity ratio for medium-size
applications is expected to be better for experts than for novices, the improve-
ment would probably be even more marked, if the level of experience had been
higher. In addition, the results support the views that application generators
are most productive in medium-size projects, and that simple, developer-friendly
hardware environments have a positive effect on productivity. The use of a pro-
totyping method more than doubles the productivity of application generators.

Furthermore, an important advantage of a prototyping approach is that the
programmers were more satisfied with the working procedures and the conduct
of the project than those in the project following a linear approach. In prototype
projects the task of a programmer is more motivating and varied allowing programmers to use their own judgement and creativity in the design and implementation of the application. The leverage effects can be significant and far-reaching. Thus the increasing job satisfaction in a prototyping approach is a factor that should by no means be neglected.

Finally, in the light of this study, it is obvious that a formal experiment using a linear strategy and prespecified test applications is inadequate for investigating real productivity effects of application generators. All the productivity effects of the tool throughout the whole application life cycle, from the preliminary work to implementation (or to maintenance), should be observed, and a development strategy suited to the project should be used. In many cases this would mean prototyping using application generators—a combination offering great potential for productivity improvements.

Acknowledgements

This paper is based on my master thesis (Pihlaja 1990) and a seminar report written together with Johanna Pajunen. I am very grateful to her and the representatives of the Society of Forestry in Finland for making the experiments possible. I would also like to thank Pasi Kuvaja for being willing to help me. In addition, I would like to express my warmest thanks to Professor Markku Sääksjärvi, Timo Saarinen and Jukka Heikilä for support and guidance.

References


Appendix:
Research settings of productivity experiments

| EXPERIMENT | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| Number of applications | 11 | 2 | 7 | 16 | 2 | 1 | 1 | 2 | 1 | 3 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| Life cycle | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| Design Implementation | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| Measurements | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| Output | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| Input | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| Language | G | L | A | N | E | M | A | L | E | N | A | C | H | A | E | U | V | A | J |
| Application gen.4 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| Modern | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| conventional | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| Small | 0-20FPA | 2 | 1 | 5 | 2 | 1 | 1 | 2 | 1 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 2 | 1 | 1 | 1 |
| Medium | 20-300FPA | 3 | 2 | 1 | 5 | 2 | 1 | 1 | 2 | 1 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 2 | 1 | 1 | 1 |
| Large | 300-400FPA | 3 | 2 | 1 | 5 | 2 | 1 | 1 | 2 | 1 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 2 | 1 | 1 | 1 |
| Development strategy | Linear | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| Prototype | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| Hardware environment | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| Mainframe | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| Minicomputer | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| Microcomputer | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |

N Novice
E Experienced
2 experienced >= 1 years experience