Estimating the Impact of the Programming Language on the Development Time of a Software Project

Frank Padberg*
Fakultät für Informatik
Universität Karlsruhe, Germany
padberg@ira.uka.de

Abstract

An empirical and quantitative comparison of C, C++, and Java is carried out. The impact of each language on the development time of a fixed sample project is estimated. A new estimation method for the development time is applied which is based on probability theory. For the sample project, C++ and Java yield better estimates than C. Without training in object-oriented programming, the C++ estimates for the sample project are worse than the C estimates. The impact of the programming language on the development time cannot be described by a scale factor.

1 Introduction

In this paper, empirical programmer productivity data are used to compare the programming languages C, C++, and Java. The question being studied is:

Does a particular programming language save development time?

The empirical data of this study fall into three datasets, one for each language. For each dataset, the development time of a fixed sample project is estimated. To compute the estimates, a new estimation method is applied. Comparing the estimates for the three languages, a manager should favor C++ or Java against C as the language for the sample project. He can expect a 12 % faster sample project completion with C++ or Java.

The empirical data are also used to measure how training in object-oriented programming improves the estimated development time for the sample project. Some of the programmers in the C++ dataset have had no OO training. When including the productivity data for the untrained C++ programmers, the time estimate for the sample project is 25 % higher than for C.

Finally, the empirical data are used to measure how outliers among the data points change the estimated development time for the sample project. Unexpected long delays are part of the risk inherent to a project. If outliers are disregarded by the manager, assuming that the next project won't run that bad again, the manager might underestimate the effort for the next project. Disregarding an outlier in the Java dataset improves the expected average development time of the sample project by 12 %.

The estimation method is based on a probabilistic model for software projects. The model reflects that the development time of a project depends on which course the project takes. From a manager's point of view, the course of a project describes "what happens at what time". From his experience with past projects a manager knows that some courses are more likely to occur than others. The idea behind the model is to compute for each possible course of a project the probability that the project will take that particular course. To compute the probabilities of the courses for a project, statistical input data are required about the courses of past projects. The model used in

*supported by a postdoctoral fellowship of the Deutsche Forschungsgemeinschaft DFG at the Graduiertenkolleg Informatik, Karlsruhe

this paper is an enhanced version of [9]. For other approaches to cost estimation such as machine learning, analogy, and neural networks, see [3][10][11][12].

Application of the model yields a probability distribution for the project completion time. This distribution reflects the uncertainty inherent to the project: a particular completion time will occur only with a certain probability. From the distribution of completion time it is straightforward to compute various kinds of estimates for the development time of the project, as well as estimates for the risk of exceeding deadlines.

The results for the sample project indicate that the impact of the programming language on the development time of a software project is non-linear. This is in clear contrast to more simplistic estimation models where some scale factor ("cost driver") is used for the programming language. The impact of the language is non-linear in the probabilistic model because the model combines statistical productivity data for the individual teams with high-level design data for the estimated project. The high-level design data measure the degree of coupling between the components of the software. Differences between the programming languages result in different productivity data for the teams, but even if the impact of the language on the productivity of a team could be described by a scale factor the structure of the software enters the model in a non-linear way. As a consequence, the impact of the programming language on the development time must be studied again with every new project.

It is unlikely that the specific numerical results of this study can be transferred to other application domains, companies, or projects: the *method* can be transferred. Therefore, a large part of the paper explains ...

- how the required empirical data looks like;
- how to compute the input distributions from the empirical data;
- how to obtain the time and risk estimates;
- how to visualize and interpret the results.

The empirical data come from university programming classes. In the classes, 17 students programmed with C, 25 with C++, and 9 with Java. Dependencies between some of the programming exercises in the classes provided the motivation for the sample project estimated in this paper. To validate the time estimates obtained from the probabilistic model, one must run the sample project for each of the languages several times with different programmers. Since finding enough volunteers for a controlled validation experiment is a concern, we are currently changing the setup of the class so that we can collect validation data over the next academic semesters.

There seem to be only few empirical studies which assess how the programming productivity depends on the programming language. In [1] Ada is compared against Fortran. Studying ten Ada projects, the authors find no significant change in effort distribution but an increase in reuse leading to lower delivery cost for Ada. In [2] one small system is developed eight times by different teams using C++. The authors find that reuse reduces the defect density and the amount of rework, and increases productivity. In [4] SML is compared against C++. One programmer develops twelve different programs, all from the same application domain, in each of the two languages. The authors find much more reuse for SML than for C++. In [5] maintenance data for two large systems are compared, one system written in C and the other in C++. The author finds that fixing faults takes much longer for C++ than C. In [6] one system is prototyped using ten different languages. The authors find that the Haskell prototype took less time to develop and was easier to understand than the Ada and C++ prototypes. In [8] one system is developed in both Scheme and Java. The authors report that the coding times, testing times, and debugging times were similar for the two languages.

2 Probabilistic model

2.1 Projects

In the model, a software project consists of several development teams. Based on the high-level design of the software, each team is assigned one component to work on. The teams start working

at the same time and keep working until their components are completed. The teams work in parallel, but not independently: problems with the software's high-level design which are detected in one component may lead to rework in other components.

2.2 States

As a project advances, the software's design will be revised from time to time because of problem reports. In the model, the time span between two consecutive designs is called a *phase.* Each phase lasts one or more *time slices.* ThUs, time is discrete in the model. For industrial projects, one slice might correspond to a day or a week.

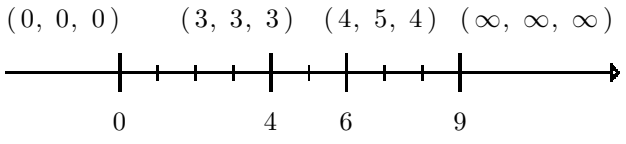
At any point of (discrete) time the *state of a team* is represented by the net development time that the team has spent working on its component, measured in slices. The net development time for a team is obtained from its total development time by substracting all rework times that the team has spent fixing high-level design problems. A value of infinity indicates that the team has finished developing. Note that the state of a team must be measurable in practice. A metric such as "x percent completed" would be hard to measure.

The *state of the project* at the end of a phase is a vector

$$\zeta \;=\; (\,\zeta_1,\ \zeta_2,\ \ldots\ \zeta_N\,)$$

where ζ_i is the state of team i at the end of the phase, and N is the number of teams. The *course of the project* is modelled as a sequence $\zeta(1),\ \zeta(2),\ \ldots$ of states, where $\zeta(j)$ is the project's state at the end of phase j. The sequence of states is supplemented by the numbers $d_1,\ d_2,\ \ldots$ specifying the length of each phase.

For example, this might be the course of a project with three teams:



The state at the beginning is $(0,\ 0,\ 0)$. In the first phase, a problem gets detected after 4 slices. The problem involves all three teams. Each team has already worked for four slices, but the teams are set back by one slice because they will need the next slice for fixing the problem. Therefore, the state of the project after the first phase is $(3,\ 3,\ 3)$. In the second phase, another problem gets detected after 2 slices. The problem involves the first and the third team. Both teams are set back by one slice again. Therefore, the state after the second phase is $(4,\ 5,\ 4)$. In the third phase, all teams complete their component, the last team after 3 slices. Thus, the state after the third phase is $(\infty,\ \infty,\ \infty)$.

2.3 Probabilities

The *transition probability*

$$\mathrm{P}_{\zeta}(\,d,\ \eta\,)$$

is the probability for ending the next phase after d time slices with state η given that the previous phase ended with state ζ.

To compute the transition probabilities, statistical data and design data are required as input; see the following subsections. As soon as the transition probabilities are known for all possible values of ζ, d, and η, one can compute the probability $\mathrm{P}(\omega)$ that the project will take a particular course ω by multiplying the transition probabilities which correspond to that course. Recall that a course of the project is a sequence of state vectors. For example,

$$\mathrm{P}_{(0,0,0)}(4,\ (3,\ 3,\ 3))\ \cdot\ \mathrm{P}_{(3,3,3)}(2,\ (4,\ 5,\ 4))$$

is the probability that a three-team project will advance in its first and second phase as described in the preceding subsection.

The probability that the project will need exactly x time slices to complete is equal to

$$\varphi(x) \;=\; \sum_{\{\,\omega:\ f(\omega)=x\,\}} \mathrm{P}(\omega)\,.$$

The sum runs over all successful courses ω whose length of time $f(\omega)$ is x time slices. A course

is successful when all teams have completed their component. The length of a course is computed by summing up the lengths d_j of the phases. The probability distribution φ of the project completion time is given for the sample project in subsection 5.1.

2.4 Statistical data

The statistical data required as input to the model are a measure of the pace at which the teams have made progress in past projects. The *base probabilities*

$$\mathrm{P}(\mathrm{E}_k^i) \quad \text{and} \quad \mathrm{P}(\mathrm{D}_k^i)$$

are the probabilities that team number i will report a problem (event E_k^i) or will complete its component (event D_k^i) after a net development time of exactly k slices. In addition, the *probability of rework time*

$$\mathrm{P}(\mathrm{R}_k^i)$$

is the probability that team number i will need exactly k slices to fix a design problem. For later use, set

$$p_i = \sum_k \mathrm{P}(\mathrm{E}_k^i) \quad \text{and} \quad q_i = \sum_k \mathrm{P}(\mathrm{D}_k^i).$$

Depending on the size of the database, a manager will distinguish between different team productivity levels and component complexity classes when computing the base probabilities from the records of past projects. For a particular team and component, the manager will ...

- look at all components developed by the team in past projects;
- classify the components according to their complexity, using a complexity measure of his choice;
- in each complexity class, look at the records to find out the development times and rework times;
- in each complexity class, compute the net times and the probability distributions;
- choose the probability distributions which are best suited for the given component.

The base probabilities for the sample project are computed from the empirical data in subsections 4.4 to 4.6.

2.5 Design data

The design data required as input to the model are a measure for the strength of coupling between the software's components. The stronger the coupling is the more likely it is that problems detected in one component will lead to rework in other components. For nonempty subsets K and X of the set of components the *dependency degree*

$$\alpha(K, X)$$

is the probability that changes in the software's design will extend over exactly the components X given that the problems causing the redesign were detected in the components K. For example,

$$\alpha(\{3\}, \{1, 2, 3\})$$

is the probability that a problem detected in the third component will lead to changes in the first, second, and third component. The dependency degrees for the sample project are computed from its high-level design in subsection 4.7.

3 Sample project

3.1 Database

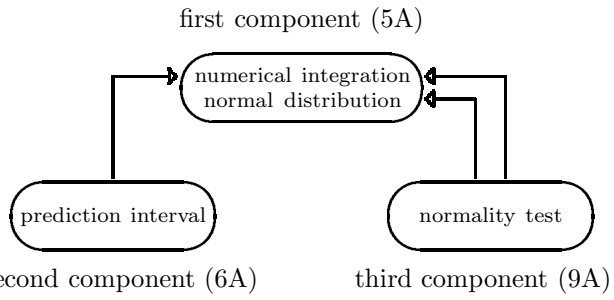
The empirical data were collected during programming classes at Karlsruhe in the years 1996 through 1999. The classes were offered to computer science students at the graduate level. The classes followed the lines of the "Personal Software Process" [7]. The goal of the classes was to help the students identify their programming weaknesses, such as their typical programming errors, and improve their effort estimation skills.

There were ten programming exercises in each class, numbered 1A through 10A, which had to be completed one exercise a week in ascending order. Each student was free to choose the programming language for the exercises. Besides C, C++, and Java, other programming languages were used as well. The students worked at home using the programming environment they were familiar with.

With each exercise, the students were required to fill out estimation forms during planning, time and defect logs during programming, and summary forms after completing the program. The students had to submit the forms together with the source code to the instructor for evaluation. The students were assured that the productivity and defect data they provided were not used for grading, so they had no reason to fake the data.

3.2 Components

The idea for the sample project came from looking at the requirements and the solutions for exercises 4A, 5A, 6A, and 9A. The programs offer simple statistics functions. Program 4A is a linear regression program. Program 6A computes the prediction interval for the linear regression estimate. Program 9A performs the χ^2 test on normality. Program 5A offers numerical integration of real functions, which is required for the computation of both the prediction interval and the normality test. Program 5A also offers the integral of the standard normal distribution, which is required by the normality test. Corresponding to exercises 5A, 6A, and 9A, the sample project gets divided into three components:



An arrow means "uses". There is no direct link between the second and the third component.

In the sample project, each component is assigned to a different team and the components are developed all at the same time. Since the components are small, a team consists of only one programmer in the sample project, but this doesn't make a difference to the probabilistic model. The requirements do not change during the project. There are no assumptions made about the software engineering methods used. How long would such a project take to complete?

3.3 Redesigns

In some cases, a change to program 5A became necessary while a student was working on 6A. The numerical integration code in exercise 5A accepted a pointer to the integrand function f as a parameter. The function f was called from inside the integration code at various points. Here is a stripped version of the corresponding C code:

```
numint(f)

float (*f)(float);
{

  ... (*f)(x) ...
```

In exercise 6A, however, several functions had to be integrated which were all from a family $(f_n)_{n=1,\,2,\,\ldots}$ of similar functions. Instead of having a different pointer for each f_n it was natural to have a single pointer to a function F which, depending on the value of a variable n, behaved as one of the functions f_n. The students found two ways of programming a solution.

The first way was to hide the variable n from the numerical integration code, for example as an external variable in C or in the class definition for F in Java, and to set n to the right value before calling the integration code. In that case, the numerical integration code wasn't changed.

The second way was to provide the value of n as an additional parameter to the numerical integration and then call F from inside the integration code with n as a parameter:

```
numint(F, n)

float (*F)(int, float);
int n;
{

  ... (*F)(n, x) ...
```

In that case, the numerical integration code had to be changed. In the sample project where all three components would get developed at the same time, the need to change the code of the first component would lead to a problem report and later a redesign of the software.

Which solution had been chosen by a student was determined by looking at the source code of exercises 5A and 6A. Both solutions occured with

all three programming languages, even Java. In an industrial setting, the fact that a redesign occured in a project would have to be seen from the project documentation.

4 Empirical input data

4.1 Data points

In the classes, there was a total of 51 students using either C, C++, or Java. One of the students programming with C did not work on exercise 9A, as did two of the students programming with C++. The following table gives the number of different students for each of the programming languages and exercises :

	C	C++	Java
5A	17	25	9
6A	17	25	9
9A	16	23	9

There were much fewer students using Java than C or C++, which must be kept in mind when comparing the project estimates for the three languages. In an industrial setting there frequently will be a different number of data points for the different components when estimating a software project.

4.2 Development times

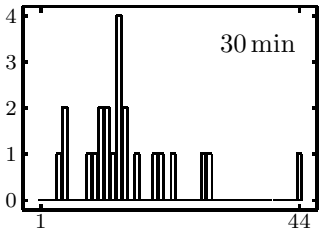
For each exercise, the students had to fill out a "Project Plan Summary" form. The form includes a table which contains the actually measured development time for the exercise :

Time in Phase	Plan	Actual	Sum
Planning			
Design			
Design Review			
Code			
Code Review			
Compile			
Test			
Postmortem			
Total			

In the "Actual" column the student fills in the times that he actually spent in the listed development phases. The other two columns are not of interest here. The times are specified as minutes. The times spent in the development phases are summed up in the "Total" field. The development time input data for this paper is taken from the "Total" field. The "Planning" field and the "Postmortem" field include the time needed for filling out all the forms. In an industrial project, the development time includes all the paperwork, too.

4.3 Time slices

The measured development times for the exercises are in the range of 1 to 22 hours. Suppose for a moment that one time slice in the model corresponds to 30 minutes of real time. A real time of 6:44 then is 14 time slices in the model. Doing this conversion for all measured development times of, say, exercise 9A programmed with C++, yields a bar chart which displays for each number of time slices its relative frequency :



As will be explained in subsection 4.5, the chart is close in shape to the chart we get for the base probabilities of completing the third component when using 30 minutes as the length of the time slices. The problem is apparent now : the bars are scattered over the chart because of variation in the empirical data, and many bars correspond to one data point only. In other words, a time slice of 30 minutes is too small to reasonably bundle up the data points.

To find a better value for the length of a time slice, some experimentation is needed. Since there are only few data points for Java, which leaves the Java charts scattered even for a time slice of 60 minutes, the length of a slice is set to 120 minutes.

It is not necessary to have all gaps closed in the chart. In particular, it is natural for empirical data to show some outliers. There also is a tradeoff here, because the longer one time slice is the more precision will be lost for the project estimates.

Depending on how long one time slice is compared to the measured development times, it may be appropriate to do some rounding when converting the measured times into slices. For example, with a two hour time slice a real time of 4:12 would count as 3 slices although it is much closer to 2 slices. For the remainder of the paper, times that exceed a multiple of two hours by not more than 15 minutes will be rounded down.

4.4 Probabilities of problem reports

In the programming classes, the change to program 5A described in subsection 3.3 was the only redesign which occured and involved exercises 5A, 6A, and 9A. Here is a table specifying for each language how many students worked on 6A and how many times the change occured:

	C	C++	Java
students	17	25	9
redesigns	5	8	1

Note that adapting program 4A while working on 6A does not count as a redesign in the sample project and does not contribute to the probabilities of problem reports. Neither 5A nor 9A use 4A, nor is there a separate team developing 4A in the sample project.

Program 5A doesn't use any functionality from 6A or 9A. Thus it is assumed that no problems with the design will be detected by the first team in the sample project, and p_1 is set to zero.

Program 9A uses numerical integration the same way program 6A does: a family of similar functions gets integrated. In the sample project, the second and third component are developed at the same time. Thus it is assumed that a problem report is as likely to come from the third team as from the second, and p_2 and p_3 are set equal.

Probabilities p_2 and p_3 together amount to the empirically measured relative frequency r of changes to program 5A. For example, the relative frequency r is $\frac{5}{17}$ for C. In the model, components are developed independently during a phase. Elementary probability theory gives the equation $p_2 + p_3 - p_2 \cdot p_3 = r$. Solving the equation for p_2 (or p_3) gives $p_2 = p_3 = 1 - \sqrt{1-r}$. Here are the resulting values for each language, specified as percentages:

	C	C++	Java
r	29.4	32.0	11.1
p_1	0	0	0
p_2	16.0	17.5	5.7
p_3	16.0	17.5	5.7

Splitting the relative frequency r between the second and the third component is necessary for the sample project only because of the special setting in which the empirical data were collected: the exercises were programmed one after another. If the empirical data came from industrial projects one would count separately for each component i of the planned project the relative frequency r_i of problem reports among the data points which are used for estimating that component, and take r_i as the value for p_i.

When working on an exercise, students usually switched several times between designing, coding, compiling, and testing. The students had to note the time when they switched the development mode on a "Time Log" form. The comments field of the form was used to note the reason for switching. The forms were not filled out detailed enough to read off the time when a student changed 5A while working on 6A. As a makeshift, it is assumed for the sample project that all points in time up to the maximum measured development time for the second component are equally likely for a problem to get reported. The same is assumed for the third component. Thus,

$$\mathrm{P}(\mathrm{E}_k^i) = \frac{p_i}{m_i - 1}$$

where m_i denotes the number of slices corresponding to the maximum net development time for component i, and k ranges from 1 to $m_i - 1$. The values for the m_i are:

	C	C++	Java
m_1	5	10	6
m_2	11	9	7
m_3	7	11	10

To sum up, the base probabilities of problem reports for the sample project are as follows. The percentages are rounded to two decimal digits:

	C	C++	Java
$P(E_k^1)$	0	0	0
$P(E_k^2)$	1.60	2.19	0.95
$P(E_k^3)$	2.67	1.75	0.64

4.5 Probabilities of component completion

The probability that the team working on component i of the sample project will complete the component without reporting a problem, given that the team doesn't get interrupted to do some rework because of design changes, is

$$q_i \;=\; 1 \;-\; p_i\,.$$

The percentages of the q_i for each language are:

	C	C++	Java
q_1	100.0	100.0	100.0
q_2	84.0	82.5	94.3
q_3	84.0	82.5	94.3

In general, the measured development times of a component include rework times spent with fixing high-level design problems. The rework times must be substracted from the measured development times to get the net development times. Among the data points used for estimating component i of the sample project, the data points which had a net development time of k slices are counted and their relative frequency s_k^i is computed. The base probabilities of completion time for component i then have the values

$$P(D_k^i) \;=\; q_i \cdot s_k^i$$

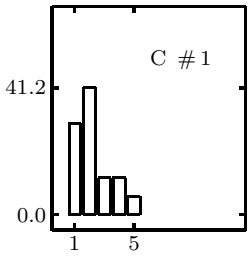
where k ranges from 1 to m_i.

For exercise 5A, the measured development times can be taken as net development times because no high-level design problems occured while program 5A was being developed. Recall that the change to 5A described in subsection 3.3 was made while 6A was being developed. The measured times for exercise 5A, programmed in C, and the corresponding numbers of slices are:

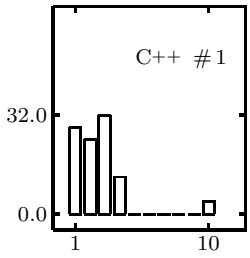
2:11	1	2:15	1	4:32	3	9:25	5
3:50	2	4:11	2	4:15	2	3:00	2
6:26	4	1:40	1	6:01	3	3:43	2
6:28	4	3:26	2	1:10	1	1:59	1
3:04	2						

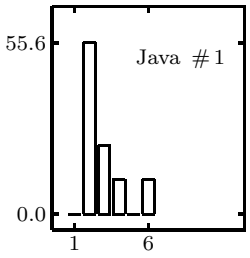
Two of the seventeen data points have (net) development time equal to four slices, so $s_4^1 = \frac{2}{17}$. Since q_1 is one, the relative frequencies s_k^1 already are the probabilities of component completion for the first team in the sample project if the language is C:

k	$P(D_k^1)$
1	29.41
2	41.17
3	11.77
4	11.77
5	5.88



For C++ and Java, the charts showing the base probabilities for the first team look like this:





The empirical data for C++ shows an outlier, $k = 10$. We shall come back to this outlier in subsection 5.2.

For exercise 6A, the measured development times include rework times spent for changing program 5A. The exact rework times could not be taken from the "Time Log" forms because the forms hadn't been filled out by the students detailed enough. As a makeshift, the rework times are estimated to have been between 15 and 30 minutes, and the measured development times are rounded accordingly. With empirical data from industrial projects the rework times would be seen from the project documentation.

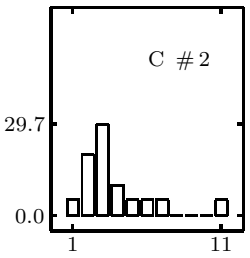
For C and exercise 6A the measured development times are:

∗8:30	1:41	4:48	20:25	∗4:57	9:15	10:45
4:43	12:54	3:59	∗7:40	∗5:29	∗5:16	6:04
2:45	2:23	4:15				

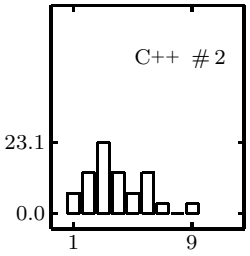
The development times for exercise 6A which include some time for changing program 5A are

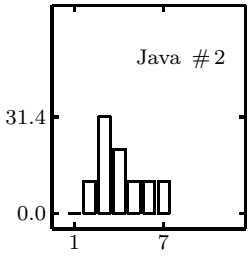
marked with an asterisk. Among these times, the 8:30 is rounded down to 4 slices to take into account the rework time. The corresponding base probabilities for the second team are:

k	$P(D_k^2)$
1	4.94
2	19.77
3	29.65
4	9.88
5	4.94
6	4.94
7	4.94
11	4.94



For C++ and Java, the charts showing the base probabilities for the second team look like this:

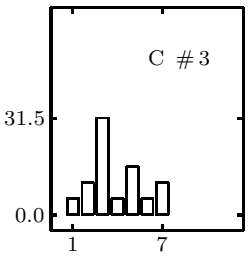




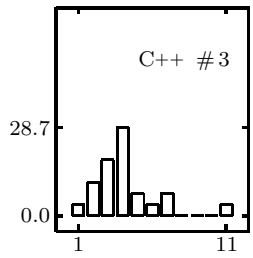
For exercise 9A, the measured development times can be taken as net development times. Recall that the change to 5A described in subsection 3.3 was made while 6A was being developed. The measured times for exercise 9A, programmed in C, and the corresponding base probabilities for the third team are:

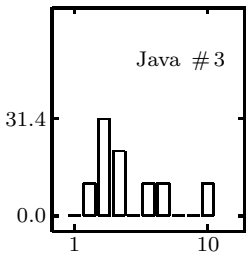
4:21	6:32	13:42	5:58	14:03	9:33	5:09
11:31	3:50	9:14	4:44	10:09	5:17	2:11
3:36	5:03					

k	$P(D_k^3)$
1	5.24
2	10.50
3	31.50
4	5.24
5	15.76
6	5.24
7	10.50



For C++ and Java, the charts showing the base probabilities for the third team look like this:





The data for Java shows an outlier, $k = 10$. We shall come back to the outlier in subsection 5.4.

4.6 Probabilities of rework times

The development times for the exercises are small. Therefore, a team is set back in the sample project by only one time slice if a redesign occurs which involves the team. Thus, the probability of rework time is

$$P(R_1^i) = 100.0$$

for all teams. Since a slice is two hours, this will still overestimate the impact of a redesign.

4.7 Dependency degrees

There are two interfaces in the sample project. One interface is to the numerical integration code which is provided by the first team and used by both the second and the third team. The other interface is to the integral of the normal distribution which is provided by the first team and used by the third team. Thus there are two cases when a problem report occurs in the sample project. If the problem relates to the numerical integration interface, then all three components are involved. If the problem relates to the interface to the normal distribution's integral, then the first and the third component are involved. Assuming that the two cases are equally likely, the table for the dependency degrees is:

	1	2	3	1,2	1,3	2,3	1,2,3
1	0	0	0	0	50	0	50
2	0	0	0	0	0	0	100
3	0	0	0	0	50	0	50
1,2	0	0	0	0	0	0	100
1,3	0	0	0	0	50	0	50
2,3	0	0	0	0	0	0	100
1,2,3	0	0	0	0	0	0	100

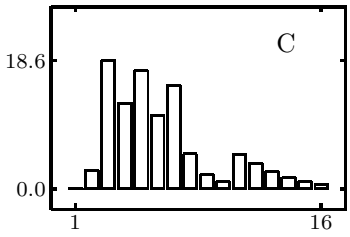
The table contains one line for each K and one column for each X. The set braces are left out.

For example, suppose that both the first and the second team report a problem, so $K = \{1, 2\}$. The problem reported by the first team relates to the first or the second interface, or both. The problem reported by the second team certainly relates to the first interface. Therefore, all three teams will be involved in the redesign. Thus, $X = \{1, 2, 3\}$ with probability 100 %.

5 Results

5.1 Project estimates

Using the probabilistic model, one can compute from the empirical input data for each of the programming languages C, C++, and Java the probability distribution φ for the project completion time of the sample project. For the C language, the model yields the following chart:



The horizontal axis shows the number x of time slices and the vertical axis the probability $\varphi(x)$.

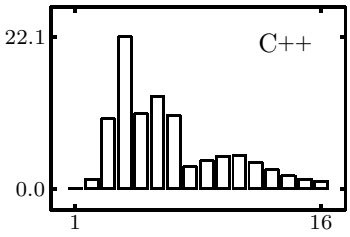
Managers can obtain a lot of information from the chart. For example, the chances of completing the sample project within 12 hours correspond to the sum of the first six bars, which is about 61 %. Recall that one time slice is two hours. In other words, if there were a deadline of 12 hours for completing the project, the risk of not finishing in time were a high 39 %.

An "average case" estimate for the development time of the sample project is obtained as the (probabilistic) expected value for the chart. To picture this, multiply the number x of each bar with its height $\varphi(x)$ and sum up. The expected value for the sample project, programmed with C, is 12.4 hours.

Another development time estimate is obtained when prescribing a maximum acceptable risk for the project. For example, if management is willing to accept a 20 % risk of exceeding the planned development time, it is sufficient to plan for a project duration of 16 hours. This is seen from the chart by successively adding up the bars until the sum exceeds 80 %. The number of bars which were added up corresponds to the development time which must be planned for. If the maximum acceptable risk is 10 %, management must plan for a project duration of 24 hours.

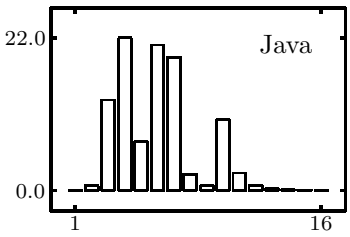
For C++, the chart for the distribution φ and the project estimates are:

	C++
average	13.2 h
20 % risk	20 h
10 % risk	26 h



For Java, the chart for the distribution φ and the project estimates are:

	Java
average	11.8 h
20 % risk	14 h
10 % risk	20 h



The results are compared in subsection 5.3. One must be careful when interpreting the results for Java, because they are based on only a few data points.

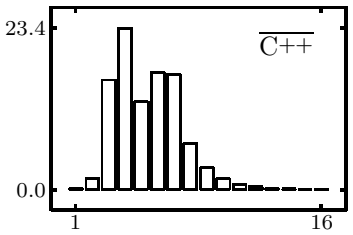
5.2 Training impact

The project estimates for C++ are higher than the estimates for C. For example, the 20 % risk estimate is 20 hours for C++ compared to 16 hours for C. The reason is that six students of the first programming class who used C++ have had no training at the university in object-oriented programming before taking the class. Their code looked much like C code with some object-oriented "overhead" added.

When including only data points for trained C++ students, the relative frequency r of problem reports for C++ drops to 26.3 %. Some long

development times for the exercises also drop out, in particular the outlier data points for exercises 5A and 6A. Because the base probabilities have improved, the estimates for the sample project improve, too:

	$\overline{\text{C++}}$
average	10.6 h
20 % risk	14 h
10 % risk	16 h

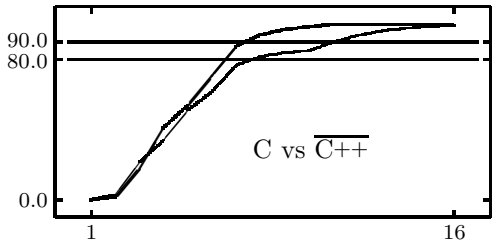


The time estimates have improved by 20, 30, and 38 % compared to the previous results for C++. This clearly shows for the sample project the impact of training when a new technology is introduced into software development. It has previously been reported in [1] that extensive training is needed before the benefits of object-oriented programming turn out.

5.3 Language comparison

For the comparison, the C++ project estimates are based on the input data from the C++ students who had some training in object-oriented programming, see the preceding subsection.

From the probability distribution φ of project completion time one can compute the corresponding cumulative distribution Φ for the sample project. By definition, $\Phi(z)$ is the probability that the project will take at most z time slices to complete, which corresponds to the sum of the first z bars of the distribution φ. The faster the graph for the cumulative distribution increases the greater are the chances to complete the project early. Therefore, the differences between the programming languages can be visualized by comparing the Φ-graphs:



The Φ-graph for C most of the time runs below the Φ-graph for C++, and there is a horizontal gap between the two graphs in the upper range. Thus one can expect faster completion and lower risk for the sample project when developing with C++ instead of C. This is reflected by the project estimates for the two languages:

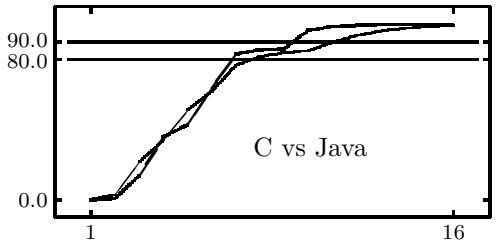
	C	$\overline{\text{C++}}$
average	12.4 h	10.6 h
20 % risk	16 h	14 h
10 % risk	24 h	16 h

There are two ways to compare the estimates. First, one can fix the maximum acceptable risk and look at the corresponding expected development times. For example, if a maximum acceptable risk of 20 % is fixed the expected development time is 16 hours for C but only 14 hours for C++. Second, one can fix a deadline and look at the corresponding risks of exceeding the deadline. For example, if a deadline of 16 hours is fixed the risk of exceeding the deadline is 20 % for C but only 10 % for C++.

If the estimates were for an industrial project extending over several months, an expected 12 % decrease in development time, respectively, an expected decrease in risk from 20 % to 10 % would be a good reason for the manager to favor C++ against C as the programming language for the project.

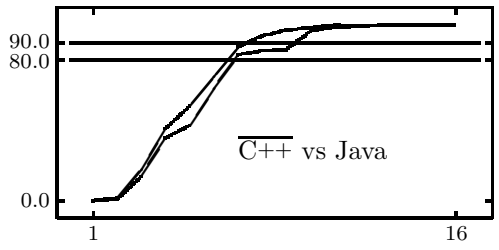
Keeping in mind that the number of Java data points is small, a manager might also favor Java against C for the sample project when looking at the risk estimates:

	C	Java
average	12.4 h	11.8 h
20 % risk	16 h	14 h
10 % risk	24 h	20 h



Again, there is a horizontal gap between the two graphs in the upper range, leading to lower risk estimates for Java. The picture is less clear when comparing C++ with Java:

	$\overline{\text{C++}}$	Java
average	10.6 h	11.8 h
20 % risk	14 h	14 h
10 % risk	16 h	20 h

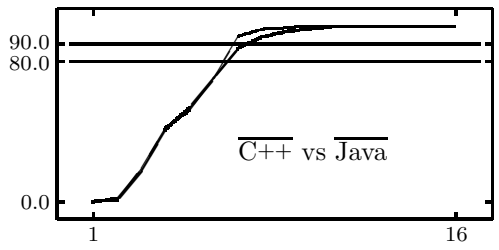


The 10 % risk estimate is better for C++ than for Java, but it is hard to tell whether the horizontal gap between the two graphs at the 90 % range is significant or not. The number of Java data points is small and there is a sharp bend in the graph for Java which looks unnatural, see the next subsection.

5.4 Outliers

Outliers in the input data can have a significant impact on the project estimates obtained from the model. For example, the Java outlier $k = 10$ of subsection 4.5 for exercise 9A is responsible for the sharp bend in the Φ-chart for Java. When disregarding the outlier, the chart and the estimates are much closer to the C++ results than before:

	$\overline{\text{C++}}$	$\overline{\text{Java}}$
average	10.6 h	10.4 h
20 % risk	14 h	14 h
10 % risk	16 h	14 h



The Java chart looks more natural now. The outlier might be explained with an unexperienced student, because the development times of the student for the other exercises were rather long, too.

References

1. Basili, Caldiera, McGarry, Pajerski, Page, Waligora: "The Software Engineering Laboratory – An Operational Software Experience Factory", Proceedings ICSE 14 (1992) 370-381
2. Basili, Briand, Melo: "How Reuse Influences Productivity in Object-Oriented Systems", Communications of the ACM 39-10 (1996) 104-116
3. Gray, MacDonell: "A Comparison of Techniques for Developing Predictive Models of Software Metrics", Information and Software Technology 39 (1997) 425-437
4. Harrison, Samaraweera, Dobie, Lewis: "Comparing Programming Paradigms: An Evaluation of Functional and Object-Oriented Programs", Software Engineering Journal 11-4 (1996) 247-254
5. Hatton: "Does OO Sync with How We Think?", IEEE Software 15-3 (1998) 46-54
6. Hudak, Jones: "Haskell vs. Ada vs. C++ vs. Awk. An Experiment in Software Prototyping Productivity", Technical Report, Yale University, July 1994
7. Humphrey: *A Discipline for Software Engineering*, Addison-Wesley 1995
8. Lloyd, Hartline: "On Language Choice for Software Projects", Preprint, University of Washington, March 1999
9. Padberg: "A Probabilistic Model for Software Projects", Proceedings ESEC/FSE 7 (1999) 109-126, Springer LNCS 1687
10. Shepperd, Schofield, Kitchenham: "Effort Estimation Using Analogy", Proceedings ICSE 18 (1996) 170-178
11. Srinivasan, Fisher: "Machine Learning Approaches to Estimating Software Development Effort", IEEE Transactions on Software Engineering 21-2 (1995) 126-137
12. Wittig, Finnie: "Using Artificial Neural Networks and Function Points to Estimate 4GL Software Development Effort", Australian Journal of Information Systems 1 (1994) 87-94