Two controlled Experiments concerning the Usefulness of Assertions as a Means for Programming

Matthias M. Müller[1], Rainer Typke[2], Oliver Hagner[3]
Fakultät für Informatik
Universität Karlsruhe, Germany
[1]muellerm@ipd.uka.de, [2]rainer@typke.com, [3]oliver.hagner@bigfoot.de

Abstract

Assertions, or more generally "Programming by contract", have gained widespread acceptance in the computer science community as a means for correct program development. However, the literature lacks an empirical evaluation of the benefits a programmer gains by using assertions in his software development. This paper reports two controlled experiments that close this gap. Both experiments compare "Programming by contract" to the traditional programming style without assertions.
The evaluation of the first experiment suggests that assertions decrease the programming effort for the extension of existing software, measured as time needed to finish the task, while the programming effort slightly increases during the development of new code. The second experiment shows that the programming effort tended to be larger with assertions than without. In addition, it shows that the reliability of the written programs slightly increases with the usage of assertions compared to the programs written without assertions.

1 Introduction

Assertions have gained widespread acceptance in the computer science community as a method for correct program development. They are used in numerous different application domains where program quality is of major concern. The literature concerning assertions, or the more common principle of "Programming by Contract", describes a never ending story of success. For example, Voas [12] states that "assertions can be a means of boosting testing's value where masking errors are most likely". And according to McKim [6], "Programming by contract is a way to provide rigorous specifications in a way that is accessible to a good technical programmer."
We do not disagree with these statements, but so far, most of the literature about assertions emphasizes their advantages without empirically evaluating the benefit a programmer gains by using assertions in his software development. This question is particularly important from an economical point of view as every manager confronted with a new technique must weigh costs versus potential benefits. So far, the literature has no answer to this question. We conducted the experiments in the hope to provide some answer.
Meyer's four hypotheses about the advantages of assertions [7] formed the starting point for the study.

Meyer's Hypothesis 1 Software is correct as it is developed along with its specification.

Meyer's Hypothesis 2 Usage of assertions leads to a better understanding of the solution and the program.

Meyer's Hypothesis 3 The development of documentation is easier with assertions.

Meyer's Hypothesis 4 Assertions form a base for structured testing and correction.

These hypotheses cover four areas that might be influenced by the use of assertions: program correctness, program understanding, documentation, and testing. Since it is almost impossible to consider all these issues in one experiment, we concentrated only on the first two topics, program correctness and program understanding. Thus, Meyer's last two hypotheses were not within the scope of our study. The first two hypotheses were considered in differing situations, i.e., during the development of new software and during the extension of existing software. We compared software development with the use of assertions to development without assertions.
Both experiments were conducted as part of a practical training course introducing the PSP (personal software process) [5] held during the winters of 1999 and 2000 at the University of Karlsruhe. Participants were computer science graduate students. The subjects in the first experiment

EXP1 used C and APP [10], while those in the second experiment EXP2 used Java with jContract [11]. Since we were interested in both the effects of assertions when writing new software and the effects on maintainability, one task of EXP1 was to write new functions that did not interact with the rest of the program, while the remaining task of EXP1 and the only task of EXP2 required a deeper understanding of the program.

The evaluation of EXP1 suggests that assertions decrease the programming effort for the extension of existing software, measured as time needed to finish the task, while the programming effort slightly increases during the development of new code. Assuming that the programming effort depends on program understanding, Meyer's second hypothesis for program maintenance has to be accepted, but it cannot be accepted for the development of new code. Evaluation of EXP2 also suggests to reject Meyer's second hypothesis, because the programming effort tended to be larger with assertions than without.

EXP2 also evaluated the reliability of the resulting programs. The finding is that programs developed with assertions had a slightly better reliability than those programs developed without assertions. However, this advantage is only marginal. And at first glance, Meyer's first hypothesis concerning better program correctness cannot be accepted either. However, looking at the programs before the quality assurance stage, i.e., the program versions the subjects considered completed, the programs developed with assertions had a higher reliability, though not significant, than the other programs. So, for these intermediate programs, Meyer's first hypotheses holds.

A detailed description of the two experiments and their results can be found in [8].

The next section of this paper gives an overview of the assertions tools used, APP and jContract. Section 3 shows the experimental settings. Finally, section 4 contains a discussion of the results.

2 APP and jContract

2.1 APP

APP ("Annotation Preprocessor for C Programs") allows the programmer to add preconditions and postconditions to functions in C programs. This tool was developed by Rosenblum [10]. A detailed description of all aspects of this tool can be found on its "man" page [9]. With APP, assertions can be included at any place within the functions. The programmer can control whether these conditions and assertions are checked during runtime and what should happen if the checks fail. Assertions are written between the special comment delimiters `/*@ ... @*/`.

The following listing shows the embedding of pre- and postconditions into the program code. Pre- and postconditions are indicated by the keywords "assume" and "return", respectively.

```
int square_root(int x)
/*@
  assume x >= 0;
  return y where y >= 0;
  return y where y*y <= x
           && x < (y+1)*(y+1);
@*/
{ ... }
```

Listing. APP example.

2.2 jContract

`jContract` allows the specification of pre- and postconditions for methods as well as class invariants. The example listing below illustrates the syntax of `jContract` assertions. Pre- and postconditions start with "require" and "ensure", respectively.

```
/**
 * @require time != null
 * @ensure return.equals("Hello")
 *           || return.equals("Good night")
 */
String welcome(String time)
{ ... }
```

Listing. jContract example.

All assertions are part of a JavaDoc comment. The jContract preprocessor transforms these special assertion tags to Java source code and inserts them into the original code. This pre-processed code can then be compiled with a normal Java compiler. Assertions can be disabled if the preprocessing step is skipped. If an assertion is violated, the program is stopped indicating the violated assertion (method name, pre- or postcondition).

The jContract preprocessor can also be used for checking the assertions at runtime without stopping program execution if an assertion is violated. In this case, the violated assertion is reported on the standard error output. These kind of assertions are called "silent assertions".

3 Experimental settings

EXP1 used a counterbalanced design, while EXP2 used a single-factor, post-test-only, inter-subject design [2].

3.1 Subjects

Overall, 22 students participated in the experiments, 9 in EXP1 and 13 in EXP2. While in EXP1, all subjects solved a task with and without assertions, the subjects in EXP2 were divided into an experimental group (7 subjects) and

a control group (6 subjects). All participants were Computer Science graduate students who had just participated in a one-semester graduate lab course introducing the PSP (personal software process) [5].
During the PSP course, the participants were introduced to assertions. After this introduction, they were told to use them during their remaining program assignments of the PSP. Also, the experiment started with a web-based training about how to use assertions (see section 3.3.2). The participants had to take part in the experiment in order to get their course credits.

3.2 Hypotheses

Based on Meyer's first two hypotheses, we investigated the following hypotheses in the experiments.

$H_{Reliability}$ Using assertions results in more reliable programs.

H_{Effort} Using assertions reduces the programming effort of development or maintenance tasks.

3.3 First Experiment EXP1

3.3.1 Task

Since the number of participants for the first experiment was rather small, we needed tasks that allow the use of each participant as a member of a group that uses assertions as well as a member of a control group that doesn't. Also, the problems to be solved by the subjects had to be sufficiently complex for any effects to be visible, while at the same time still being solvable within the limited amount of time participants were willing to spend on the experiment.
In order to fulfill these constraints, the participants were assigned two tasks that were based on the same C program. These tasks were unrelated so that people could be used both as a member of the control group and as a member of the group using assertions. Since they had to get to know only one program, the tasks could be more complex than tasks dealing with two different programs could have been.
The program chosen as a basis for the tasks of EXP1 symbolically derives functions and lists intermediate steps. This program was initially written for a purpose unrelated to this experiment.
Since we were interested in both the effects of assertions when writing new software and the effects on the extension of existing software, one of the two tasks was to write new functions that did not interact with the rest of the program, while the other task required a deeper understanding of the program.
The first task "String" was to write equivalents for the `insert` and `delete` string library functions of Pascal. Since the symbolic derivation program had been ported from Pascal to C, it made use of these string functions, which are not part of the standard C library. Therefore, these library functions did not need to call functions in the given program, and no knowledge about the program was required or even useful for writing the new functions.
Assertions can be useful here because they make it easy for the programmer to check whether the preconditions he builds on when writing the new functions are met, and whether the new functions always do what they are supposed to do. If the program fails, it should therefore be easy to determine if the cause lies in a new function or the rest of the program.
This task had the character of writing new software since these string functions have nothing to do with deriving functions, therefore it was not necessary to look at the rest of the program for solving the problem.
The second task, "Chain", was to extend the program so that it could apply the chain rule, i. e. $(f(\varphi(x)))' = f'(\varphi(x)) \cdot \varphi'(x)$.
Assertions can be useful here because the programmer can save effort by reusing existing functions. This also enhances the quality of the resulting software. Both the documentation character of assertions, which makes it easier to find reusable functions, and the fact that assertions can help one quickly detect erroneous ways of reusing existing functions can help here.

3.3.2 Procedure

Every participant completed one of the two tasks described above as a member of one group and the other task as a member of the other group.
Table 1 shows group sizes, group tasks, and the order of the tasks. Originally, there were two subjects in group 3, but the data of the second member had to be discarded because he did not finish.

Group	Size	1. Task	2. Task
1	2	Str, APP	Ch, nAPP
2	3	Ch, nAPP	Str, APP
3	1	Str, nAPP	Ch, APP
4	3	Ch, APP	Str, nAPP

Table 1. Group sizes and tasks during EXP1. *Str* **= String function task.** *Ch* **= Chain rule task.** *APP* **= with APP.** *nAPP* **= without APP.**

Every participant was given syntax training before working on the tasks. This training was a web-based introduction to APP. The web-based script presented the APP syntax, asked the participant to write APP assertions for given functions,

and commented on the correctness of the participant's input. Only when at least half of the training assignments were solved correctly was the participant allowed to start working on his first task. Because the syntax training was completely automated, it was identical for every participant.
Even though the string functions were necessary for extending the program by adding the chain rule, the string function task could be done after the chain rule task. The participants were simply given the string functions in the form of object files so that the program could be compiled and tested even though the participants did not have access to the source code of the string functions.
The sequence of participants' tasks was as follows:

- The participant is handed a paper-based form for an experiment protocol. This form contains a questionnaire, the assignments, and space for keeping track of the time spent on its different parts, as well as a description of APP.
- The participant fills in the questionnaire.
- The participant reads about APP.
- The participant goes through the web-based APP training.
- Solving of first task.
- Solving of second task.

3.4 Second Experiment EXP2

The second experiment was a repetition of the first one but with a different task and a different design. It was a single-factor, post-test-only, inter-subject design. The controlled independent variable was whether the subjects used jContract with program code annotated with assertions or not. The subjects of the experiment group were allowed to write new assertions with jContract and they got program code to reuse that was annotated with assertions. The subjects of the control group got the same program code without assertions but the information of the assertions was provided for them in the form of JavaDoc comments in natural language. Each subject of either group solved the same task and worked under the same conditions. The observed dependent variables for each subject were a variety of measurements of the development process (in particular working time), and various measurements of the delivered product (program reliability, number of reused methods, and quality of reuse).

3.4.1 Task

The task given in EXP2 was called "GraphBase". It consisted of implementing the main class of a given graph library [4] containing only the method declarations and method comments but not the method bodies. There are methods for adding vertices and edges and for deleting and cloning a whole graph. Other methods include accessor methods, e.g. for showing the number of vertices or edges, methods for finding an edge between two given vertices, or methods for testing if the graph is empty, weighted, or directed.
Each subject was told that the original code of GraphBase was lost and, because there was no backup, that it should be reimplemented by using the rest of the given graph library. The requirements for this task were thoroughly described in natural language. The subjects were asked to work and to test on their own until they felt they had finished the task.

3.4.2 Procedure

The experiment took place between February 2000 and April 2000, mostly during the semester breaks. Most subjects started at about 9:30 in the morning. The experiment materials were printed on paper and consisted of three parts. The first part described a web course that only the subjects of the experiment group had to pass. The second part contained a task description. The third part consisted of a questionnaire that was handed out to every subject at the end of the experiment. It contained questions about the understandability of the documentation and asked for personal ratings concerning program understanding and the reliability of the resulting program.
The subjects worked on the task using their own specific Unix account that provided the automatic monitoring infrastructure. It transparently protocoled login/logout times, all compiled source versions and all outputs from each program run. The subject could modify the setup of the account as necessary. The source code of the graph library except for the GraphBase method bodies was provided to the subjects.
The subjects' work was divided into three phases.

Web course phase (WC), during which the subjects in the experiment group were introduced to the syntax of jContract. The control group skipped this step.

Implementation phase (IP), during which the subjects solved their assignment until they thought that their program would run correctly. This phase ended when they claimed to have finished the task.

Correction phase (CP), during which the subjects were given more details about the expected implementation. The experiment group was given a list of postconditions for every method that had to be implemented. They got this list on paper and in electronic form. The control group was given a description for every method in natural language. All subjects were asked to check

their implementation with this additional information and correct it if necessary.

3.5 Power analysis

Cohen [3] stresses the importance of power analysis to get a closer look at the quality of a statistical hypotheses test. EXP1 and EXP2 have a power of 0.41 and 0.52, respectively [8].

According to Cohen, both experiments have a very poor power. He argues that only experiments with a power of more than 0.8 have a real chance to reveal any effect. Therefore, it is quite reasonable that neither experiment has the chance to show an effect, even if a difference exists. But, as we could not acquire any more subjects for these experiments, we had to abide by this drawback.

3.6 Threats to internal validity

The control of the independent variable is threatened by the possibility of an imbalanced group assignment – one might compare one group with faster programmers to one group with slower programmers. To avoid this effect, the group assignment was based on the subject's PSP course productivity. This productivity was measured as number of lines of codes programmed per hour in the PSP course. For both experiments, the division resulted in groups with similar productivity.

3.7 Threats to external validity

There are two important threats to the external validity (generalizability) of the experiment. First, professional software engineers may have different levels of skill and experience than the participants, which might make the results too optimistic or too pessimistic. Both higher and lower levels will occur, because the students are more skilled than most of the non-computer-scientists that frequently start working as programmers. A higher skill level than the subjects' might leave less room for improvement which might reduce the group differences, but higher experience may also sharpen the eye as to where improvements are most desirable or most easy to achieve. Conversely, lower skill may leave more room for improvement but may also impede applying assertions correctly at all. Second, the subjects used assertions a very short time after being introduced to them. It is conceivable that the assertion usage of these persons had not yet stabilized and the mid-term benefits would be higher than observed in the experiment. Furthermore, work conditions different from the experiment conditions may positively or negatively influence the effectiveness of assertions.

4 Results

Box plots are used to show the results of the measurements. The filled boxes within a plot contain 50% of the data points. The lower (upper) border of the box marks the 25% (75%) quantile. The left (right) t-bar shows the 10% (90%) quantile. The median is marked with a thick dot (•). The M marks the mean, and the dashed line shows the range of one standard error on each side. The variance of a data-distribution is measured as fraction of the 75%- to the 25%-quantile.

Significance was calculated with the two-sample Wilcoxon, Mann, Whitney Test (referred to as Wilcoxon test), where the significance p denotes the probability that the observed difference is due to chance.

4.1 Results of EXP1

4.1.1 Working time

Figures 1 and 2 show the working times in minutes for Chain and String, respectively. That the distribution of durations was more dense for the group using assertions (indicated "with APP" in the figures) was an unexpected phenomenon. The use of assertions therefore might make software development more predictable.

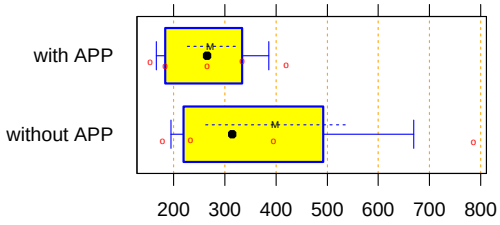

Figure 1. Duration for the chain rule task, measured in minutes. Group variances are $v_{APP} = 1.82$ and $v_{noAPP} = 2.24$.

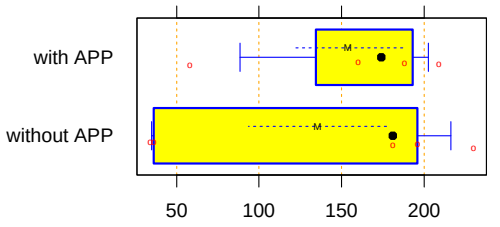

Figure 2. Duration for the string task, measured in minutes. Group variances are $v_{APP} = 1.43$ and $v_{noAPP} = 5.44$.

Figures 1 and 2 do not show significant differences that would be caused by the use of assertions. Because the groups were quite small, differences in the individual programming speeds of participants had a large influence on the results of the Wilcoxon tests. It is possible to lower the influence of individual programming speeds by measuring the time spent on the programming tasks in multiples of the time spent on the APP training instead of in minutes. This is legitimate because there is a correlation between the participants' programming speeds and the time they spent on the APP training. The correlation coefficient is 0.84. This means that subjects who are good at quickly solving the APP training assignments are also good at quickly finishing the programming tasks. If the time spent on the programming assignment is measured relative to the training time, the influence of the participants' differing qualifications is eliminated. Figures 3 and 4 compare the durations measured in multiples of the time spent on the APP training instead of in minutes, so the influence of programming speed differences is lowered and the influence of the use of assertions becomes more visible.

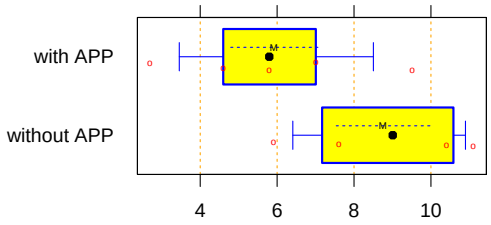

Figure 3. Relative durations for the chain rule task ($p = 0.06$).

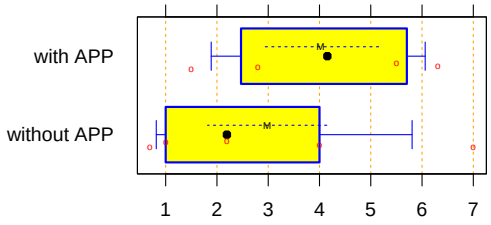

Figure 4. Relative durations for the string function task ($p = 0.28$).

The difference visible in figure 3 is significant. The Wilcoxon test shows that the probability for an accidental difference is $p = 0.06$. Therefore, assertions seem to save time when software is extended, while they tend to increase the effort needed for writing new software, see figure 4. But the difference visible in figure 4 is not significant ($p = 0.28$).

4.1.2 Code reuse

We are interested solely in the number of reused functions as opposed to the number of function reuses, as only the former can indicate how the use of assertions contributed to reusing many different functions. Function reuse was counted in the following way: for each participant, the final version of the extended program was compared to the version with which he started using the UNIX tool `diff`, thereby isolating the code written by the participant. A Perl program was then used to count the number of different functions that were already defined in the original program and called in the new code. The result for the maintenance task, the chain rule assignment, is shown in figure 5. The difference is significant: the probability for an accidental difference is 0.07.

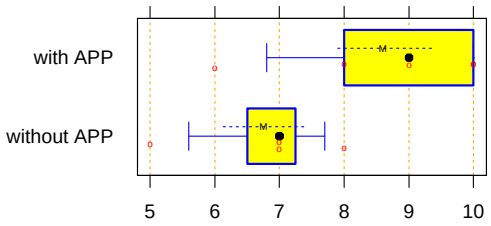

Figure 5. Number of reused functions for the chain rule task ($p = 0.07$).

The described counting method for function reuse included functions that were reused only within assertions. Figure 5 shows the results if only functions that were reused outside assertions are counted. The observed difference is still visible but no longer significant.

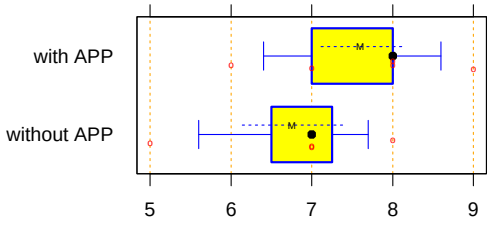

Figure 6. Number of reused functions outside assertions for the chain rule task ($p = 0.19$).

4.2 Results of EXP2

4.2.1 Working time

We now present the working time needed for the IP. As only the experiment group worked on the web course, the time

spent for the web course cannot be part of the working time. There is also a large difference in the duration of the CP: the experiment group got a list of postconditions for every method which they had to implement. All subjects in this group copied these postconditions into their implementation which took a long time for the subjects of this group. The control group couldn't do this because its subjects got the same information only in natural language, so they looked directly for defects in their program code after reading this information. Comparing the minimum and the maximum of both groups, it can be seen that the control group needed between 24 and 55 minutes and the experiment group between 68 and 199 minutes for the CP.

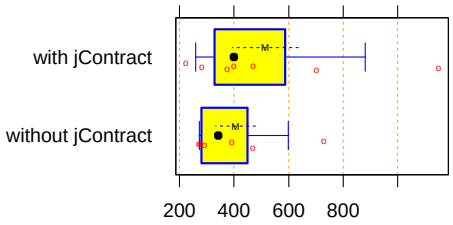

Figure 7. Working time in minutes for the IP ($p = 0.31$).

Figure 7 shows that the experiment group ("with jContract" in the figures) tended to need more time for the implementation than the control group, but the difference is not significant with $p = 0.31$.
The data point at 1269 minutes in the experiment group can be viewed as a outlier with a factor of 2.9 higher than the median. This is reasonable because of the programming experience of the subject: the largest program this person had written before the PSP course was about 300 lines of code, and in the PSP course, the person was one of the slowest measured in lines of code per hour. In other compared measures, there was not such an outlier effect.
In contrast to EXP1 where the usage of APP with assertions decreased the programming effort of the maintenance task, the subjects of the experiment group tended to spend more time for the task when programming with assertions than without.

4.2.2 Code reuse

Examining code reuse might lead to some results about program understanding. Four measures were collected to get a perception of it. These are (1) the number of reused methods, (2) the number of reused methods outside assertions, (3) the number of failed method calls, and (4) the number of method calls that failed at least twice. The last two measures were obtained with silent assertions inserted into the existing graph library (see section 2.2). Their output was written to a log file, without notice of the subjects.
Figure 8 shows the results for the number of reused methods with and without assertions. It shows with $p = 0.23$ no significant difference in the number of reused methods. However, there is a tendency that with assertions the subjects reused more different methods. This tendency disapears if the maximum point at 32 in the experiment group is ignored.
The number of methods reused outside assertions was also examined. The result was that the experiment group reused significantly less methods than the control group. But the observed difference is due to the characteristics of the implementation. For example, the method `add(Edge)` has to test for four different properties in order to properly perform its task to add the edge: are the graph and the edge weighted, are both directed, does the graph already contain the edge, or is the graph empty and thus does this edge define the type of the graph? All programs of the control group perform these queries with if-statements, while the experiment group used assertions instead. Therefore, not counting the code reuse within assertions ignores an important aspect of the implementation. And in this case, this comparison is meaningless.

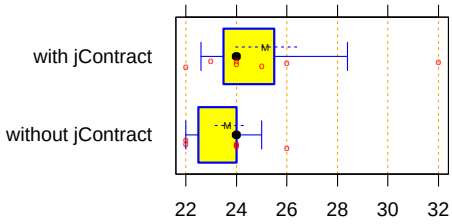

Figure 8. Number of reused methods ($p = 0.23$).

We now examine proper reuse of existing methods. Figure 9 shows the number of assertions that failed at least once. The experiment group tends to use existing methods more erroneously for the first time than the control group.

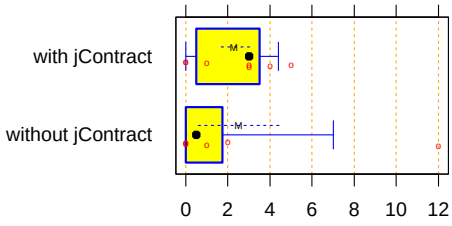

Figure 9. Number of assertions that failed at least once ($p = 0.23$).

This effect decreases when the methods are reused more often. Figure 10 shows the number of assertions that failed at least twice.

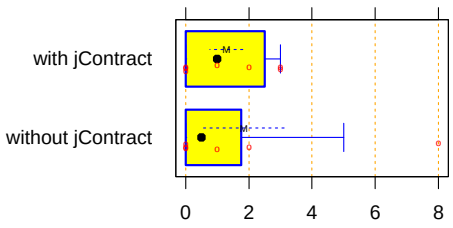

Figure 10. Number of assertions that failed at least twice ($p = 0.44$).

Finally, using assertions slightly increases the reuse of existing code. But assertions do not aid in the proper reuse of existing methods.

4.2.3 Reliability

EXP2 also evaluated the reliability of the written programs. Reliability was measured by determining the percentage of the passed assertions among all possible executable assertions in the test. A random test with 727,190 method invocations and about 7.5 million assertions were used. The reference implementation runs for about 150 seconds for this test. It calls the methods of the implementation randomly, but with different probabilities, and compares the resulting data structure with the one built by the reference implementation. Deviations in the structure are caught by assertions. This random test was written using jUnit [1].

To count all failed assertions the initial behavior of jUnit had to be adjusted. That is, jUnit was modified in such a way that it did not abort a test after a failed assertion. Instead, it continued the test case so that all assertions were executed. The failed assertions were counted and printed out at the end of the test run.

First, the reliability of the final programs after the CP is examined, see figure 11. Almost no difference can be seen between both groups.

We now turn our attention to the programs right after the IP. What would have happened if the CP had been omitted? This question is interesting in as much as these programs represent the output of the subject's process without further modifications or enhancement by any external quality control. These programs represent the versions of whose accuracy the subjects are most confident. The reliability of the intermediate program versions is shown by figure 12.

Though the reliability of the experiment group is higher, the difference is with $p = 0.11$ not significant. Except for two programs, two thirds of the programs in the experiment group are more reliable than the median in the control group, which is only at 3%.

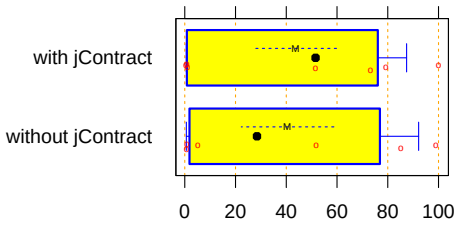

Figure 11. Reliability of the final programs in percent ($p = 0.53$).

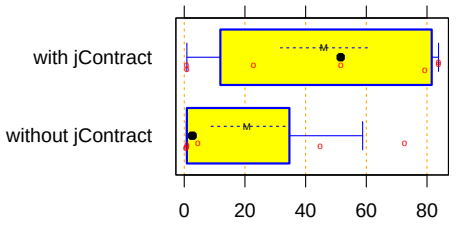

Figure 12. Reliability of programs after the IP in percent ($p = 0.11$).

Concerning reliability, it can be said that the usage of assertions is an advantage compared to informal information like the natural language documentation. Meyer's first hypotheses holds for the intermediate programs after the IP.

5 Conclusions

This paper presented two controlled experiments about the usefulness of assertions as a means of programming. Participants were computer science graduate students who took part in a practical training course introducing the PSP. Both experiments compared programming with assertions to the development without assertions. The study investigated the influence of assertions on programming effort and program reliability. The experiment data led to the following observations.

- The first experiment suggests that assertions reduce maintenance programming effort if the maintenance task is defined as a program assignment that requires a deep understanding of the program. In contrast, the second experiment shows a quite different picture because there, assertions tended to increase the programming effort.
- Assertions slightly increase the programming effort for the implementation of new functions that do not interact with the rest of the program.

- When looking at the final programs of the second experiment, the usage of assertions slightly increased the reliability of the written code compared to the code written without assertions. The effect is only marginal. However, when looking at the reliability of the programs right after the implementation phase, the programs of the experimental group, i.e., the group that used assertions, were more reliable, though not statistically significantly more so than those of the control group.

- The usage of assertions also led to a higher number of reused methods that were not written by the subjects themselves.

Despite the observed results, this study is far from being a complete evaluation of programming with assertions. There are several circumstances that weaken the discussed results. First, the number of subjects was very small, which led to a small power of finding an existing effect. This small power could been a hindrance in seeing sharper results. But, this is also a result from power analysis, some effects that weren't detected with this experimental setting could still be there and wait for their discovery. Second, the subjects have only limited experience with assertions, and it is quite possible that more experienced programmers would show quite different results. Overall, using assertions slightly increases programming effort in the worst case, but this increased effort pays off in a higher reliability of the written software, even for novice assertion-users.

Acknowledgements

We thank our students who participated in the experiments, and also especially Agatha Walczak-Typke for commenting on drafts on this article.

References

[1] K. Beck and E. Gamma. junit. http://www.junit.org/.

[2] L. B. Christensen. *Experimental Methodology*. Allyn and Bacon, 1994.

[3] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Academic Press, 1977.

[4] D. Goldschmidt. Design and implementation of a generic graph container in java. Master's thesis, Rensselaer Polytechnic Institute in Tray, New York, Apr. 1998.

[5] W. Humphrey. *A discipline for software engineering*. Addison-Wesley, 1997.

[6] J. McKim. Programming by contrac: Designing for correctness. *Journal of object oriented programming*, 9(2):70–74, May 1996.

[7] B. Meyer. *Object-oriented software construction*. Prentice-Hall, 1988.

[8] M. Müller, R. Typke, and O. Hagner. A detailed description of two controlled experiments concerning the usefulness of assertions as a means for programming. Technical Report 2002-2, Computer Science Department, University of Karlsruhe, Feb. 2002.

[9] D. Rosenblum. APP. http://www.research.att.com/sw/tools/reuse/.

[10] D. Rosenblum. Towards a method of programming with assertions. In *International Conference on Software Engineering*, pages 92–104, Melbourne, 1992.

[11] J. Störk. Erzeugung effizienter Laufzeitüberprüfungen von Zusicherungen. Master's thesis, Department of Computer Science, University of Karlsruhe, May 1999. http://www-is.informatik.uni-oldenburg.de/∼stoerk/da/diplomarbeit.html. Only available in German.

[12] J. Voas. How assertions can increase test effectiveness. *IEEE Software*, pages 118–122, Mar./Apr. 1997.