A detailed Description of two controlled Experiments concerning the Usefulness of Assertions as a Means for Programming

Matthias M. Müller[1], Rainer Typke[2], Oliver Hagner[3]
Computer Science Department
University of Karlsruhe, Germany
[1]muellerm@ipd.uka.de, [2]rainer@typke.com, [3]oliver.hagner@bigfoot.de
Technical Report no. 2002-2

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 empirically evaluation of the benefits a programmer gains by using assertions in his software development. This paper reports about two controlled experiments to close this gap. Both experiments compared "Programming by contract" to the traditional programming style without assertions. The evaluation suggests that assertions tend to decrease the programming effort and that assertions lead to more reliable programs compared to those programs written without using them.

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 about assertions or the more common principle of "Programming by Contract" describes a never ending story of success. For example, Voas [Voa97] states that "assertions can be a means of boosting testing's value where masking errors are most likely". And according to McKim [McK96], "Programming by contract is a way to provide rigorous specifications in a way that is accessible to a good technical programmer." He concludes: "Therefore programming by contract is a good thing!"
We do not disagree with these statements, but so far, most of the literature about assertions emphasizes their advantages without empirically evaluating the main question: What benefit does a programmer gain by using assertions in his software development? From an

economical point of view, that would be the most important question for every manager who is confronted with a new technique. What do I get from it and at what price? So far, literature has no answer to this question. This is where our experiments come into play.
Meyer's four hypotheses about the advantages of assertions formed the starting point for our study [Mey88].

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

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

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

Meyer's Hypotheses 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 on the first two topics, program correctness and program understanding. Both issues were considered in different situations, i.e., during the development of new software and during maintenance. We compared software development with the use of assertions to development without assertions. Meyer's last two hypotheses were not within the scope of our study.
Both experiments were conducted as part of a practical training course introducing the PSP (personal software process) [Hum97] 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 [Ros92], while those in the second experiment EXP2 used Java with jContract [Stö99]. 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 much with the rest of the program, while the other task of EXP1 required a deeper understanding of the program. EXP2 involved solving one programming task which had the character of developing new software.
The evaluation of both experiments suggests that assertions decrease the programming effort for maintenance, measured as time needed to finish the task, while the programming effort slightly increases during the development of new code. If we assume that the programming effort depends on program understanding, we have to accept Meyer's second hypothesis for maintenance, but we cannot accept it for the development of new code.
EXP2 also evaluated the reliability of the resulting programs. The result is that programs developed with assertions had a slightly better reliability than those programs developed without assertions. But this advantage is only marginal. And at first glance, we also cannot accept Meyer's first hypothesis concerning better program correctness. But, if we look at the programs before the quality assurance stage, i.e., the program versions the subjects considered finished, the programs developed with assertions had a much higher

reliability, though not significant, than the other programs. And for these intermediate programs, Meyer's first hypotheses holds.
This paper is organized as follows: the next section presents related work about evaluation of assertions. Section 3 describes the used assertions tools APP and jContract. Section 4 shows the experimental settings. In section 5, the results are discussed.

2 Related Work

Meyer's integration of assertions into Eiffel as native language constructs [Mey88] is not the only example of assertions being added to programming languages. Other examples are the assert keyword in Java 1.4 [jav], ANNA (ANNotated Ada) [LST91], Tcl [Coo97], iContract [Sys], jContract [Stö99], APP [Ros92], xUnit [xun], and the assert-library in C.
In an empirical study, Leveson et al. [LCKS90] compared software error detection self checks with N-version voting. They noted that there are great differences in the ability of individual programmers to design and place effective checks. And generally, specification-based checks alone were not as effective as combining them with code-based checks. The comparison of self checks and N-version voting revealed that both techniques identified the same number of defects, although the observed defects were not the same. In fact, self checks detected errors caused by faults that had not been detected by N-version voting in one million of randomly generated input cases. The authors conclude that self checking may have important advantages over voting.
Rosenblum developed the "Annotation PreProcesser" (APP) for C, see 3.1 for a description, and presented a classification of faults found with and without APP [Ros95]. He showed that almost 75% of all observed faults could be found with assertions written with APP.
Other work concerning assertions mention their benefits but lack empirical evidence about the assumptions made. The following list is an excerpt and does not claim to be complete.
Luckham, Shankar, and Takahashi [LST91] propose a method called "two-dimensional pinpointing". They insert annotations into the code which check conformance with specifications during runtime. When violations are found, they try to pinpoint the subunit that caused the inconsistency. Their search varies with the structure level of the software and the test sequence length.
McKim [McK96] emphasizes the benefit of using assertions. He does not rely on a special tool, instead, he developed rules that guide the insertion of assertions to get the most valuable benefit.
Schneider [Sch98] argues in the same manner as McKim when he discusses how assertional reasoning can be used in the analysis and development of concurrent programs.

3 APP and jContract

Before we describe the experiment settings in section 4, we shall describe the assertion tools that were used for the experiments, APP and jContract.

3.1 APP

APP ("Annotation Preprocessor for C Programs") allows the programmer to add preconditions and postconditions to functions in C programs. Also, 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.
The APP tool was developed by David S. Rosenblum [Ros92]. A detailed description of all aspects of this tool can be found in its man page (see [Ros]). Our description will mainly focus on the syntax and semantics, which should be enough for an overview of how it can be used.
APP assertions are written between the special comment delimiters `/*@ ...@*/`. Within APP assertion regions, comments can be placed between "`// `" and the end of the line.
For APP assertions, C expressions are used with the convention that zero is interpreted as "false", while non-null pointers and numbers are interpreted as "true". In addition to the standard C keywords, the following keywords can be used for writing APP assertions:

- **assume** `<condition>;`
  This keyword denotes a precondition. The programmer can place an arbitrary number of **assume** conditions between the head and body of a function. Whenever the function is called, all these conditions are checked before the first line of the function is reached. If a check fails, the user-defined action takes place. The default action is to print a message to `stderr` which includes the type of the violated assertion and the line number.

- **promise** `<condition>;`
  Postconditions are specified with this keyword. Like preconditions, they are placed between the head and body of a function. The only difference between the **assume** and **promise** keywords is the point of time when these assertions are checked. Postconditions are checked after the function has been completely executed, but before control is returned to the caller.

- **return** `<identifier>` **where** `<condition>;`
  The **return** keyword can be used to specify a property of the return value. Therefore, this kind of assertion is implicitly a postcondition. Like the **promise** and **assume** keywords, it is placed between the head and body of a function. Multiple **return** conditions are checked sequentially.

- **assert** `<condition>;`
  This keyword can be used to check a condition anywhere within the body of a function.

- **in** `<expression>`
  Expressions that are preceded with "**in**" are evaluated with the value they had when the current function was called, even if the expression is somewhere inside the current function or part of a postcondition. In Eiffel, the same effect is achieved with the

"old" keyword. The APP "in" should not be confused with the "in" keyword in jContract.

Examples for the keywords that were mentioned so far:

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

The precondition is: $x \geq 0$.
Postcondition: The return value is ≥ 0*, its square is* $\leq x$*, but* $x <$ (*return value* $+1)^2$.

```
void swap(int *x, int *y)
/*@
   assume x && y && x != y;
   promise *x == in *y;
   promise *y == in *x;
@*/
{
  *x = *x + *y;
  *y = *x - *y;
  /*@
    assert *y == in *x;
  @*/
  *x = *x - *y;
}
```

Here the precondition is that neither x nor y are null pointers and that they do not point to the same location.
Postconditions: the values that x and y point to have switched places.
The assertion within the function body marks the line where one of the postconditions should already hold, provided that there was no overflow.
In addition to the keywords shown above, there are the quantors

- **all** `(Initialisation; loop condition; iteration) <expression>;`
  and
- **some** `(Initialisation; loop condition; iteration) <expression>;`

which are convenient for specifying properties of many array elements in a single statement.

3.2 jContract

With `jContract`, class invariants can be defined as well as pre- and postconditions for methods. The example below illustrate the syntax of `jContract` assertions. All assertions are part of a `JavaDoc` comment. The `jContract` preprocessor transforms these special assertion tags into Java source code. The result of the preprocessing is then the original source code merged with the Java code resulting from assertions. The pre-processed code can then be compiled with a normal Java compiler. It is easy to disable the assertions by just skipping the preprocessing step before compiling the source code.
The example shows a greeting depending on the current time. The parameter of the method must be a valid object reference (precondition) and the returned greeting string is either "hello" or "good night" (postcondition).

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

Listing: jContractExample.java

If an assertion is violated, the program is normally stopped with a runtime exception saying which assertion (method name, pre- or postcondition) is violated. The `jContract` preprocessor can also be used for checking the assertions without stopping the program when an assertion is violated. The violation will then only be reported on the standard error output. We call these kind of assertions "silent assertions".

4 Experimental settings

EXP1 used a counterbalanced design, while EXP2 used a single-factor, posttest-only, inter-subject design [Chr94].

4.1 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 male Computer Science graduate students who had just participated in a one-semester graduate lab course introducing the PSP (personal software process) [Hum97].
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. The participants had to take part in the experiment in order to get their course credits.

4.2 Hypotheses

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

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

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

Thus, we obtained the following null-hypotheses.

$\mathbf{H_{0,Reliability}}$ The programs developed with assertions are at most as reliable as programs that aren't developed with assertions.

$\mathbf{H_{0,Effort}}$ The programming effort does not decrease when using assertions.

4.3 First Experiment

4.3.1 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 people 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.
While we were looking for suitable tasks, it became obvious that not all programs are equally suitable for the use of assertions:

- Programs that are split into very small functions are in danger of requiring more space and effort for the assertions than for the program itself. This does not seem to make much sense since the probability of mistakes within the assertions would be unreasonably high, and the effort of writing the program would be increased considerably.

- On the other hand, programs that are split into very large and complex functions do not seem to be good candidates for the use of assertions either. Such functions would require complex assertions using auxiliary functions that are about as complex as the main function itself. When a complex assertion fails, the high probability of mistakes within complex assertions would raise the question whether the assertion

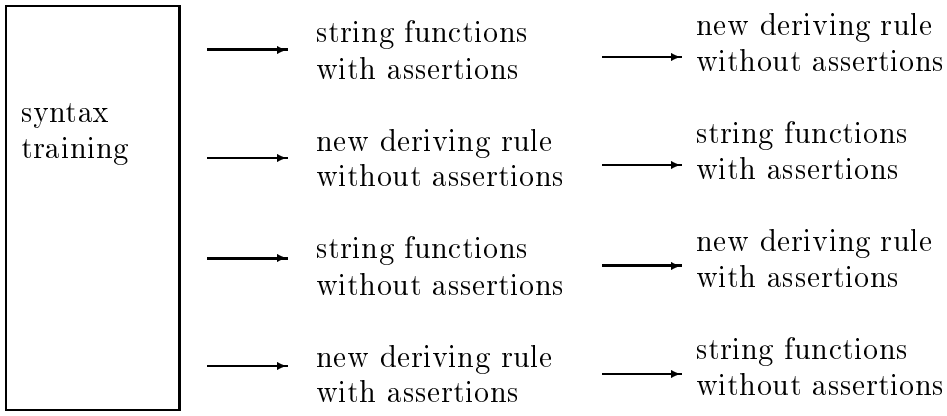
with its auxiliary functions or the program is correct. If this question cannot be answered quickly and with little effort, assertions do not seem to be too useful.

- Programs whose behaviour is difficult to describe with assertions, e. g. graphical user interfaces that have to fulfill conditions like "the new window has a green border and is completely visible", do not seem to be promising candidates for the use of assertions.

The program we chose as a basis for the tasks symbolically derives functions and lists intermediate steps. This program was not written for the purpose of this experiment.
Since we were interested in both the effects of assertions when writing new software and the effects on maintainability, one of the two tasks was to write new functions that did not interact much with the rest of the program, while the other task required a deeper understanding of the program.
The first task was to write equivalents for the `insert` and `delete` string 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.
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 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 to quickly detect wrong ways of reusing existing functions can help here.

4.3.2 Procedure

Since the number of participants was small, every participant was used as a member of two groups, a control group and a group using assertions. 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.
It is to be expected that the order of the two tasks matters. After completing one task, a programmer already knows the program, has gotten used to the programming environment, and is probably more tired than he was at the beginning. These and other reasons can lead to unwanted sequence effects. In order to counterbalance these effects, the participants were split into four groups of similar size that went through the experiment on four different paths:



The group sizes were as follows: three people first worked on the chain rule task using assertions, followed by the string function task without assertions. Three other people did the tasks in the same order, but used assertions only for the string function task. One person first solved the string function assignment without using assertions before working on the chain rule using assertions (we had to discard the data of the second member of this group because he did not finish the tasks), and two people started working on the string functions using assertions before working on the chain rule without using assertions.
Every participant was given a 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, the participant was 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.
- First task: chain rule or string functions.
- Second task: string functions or chain rule. If the first task was done using assertions, the second task is done without and vice versa.

4.4 Second Experiment

The second experiment is a repetition of the first one. The differences were that we used a different task and that it was only an inter-subject design. The rest of the design was the same: single-factor and posttest-only. The controlled independent variable was wether the experimental subjects were given code to reuse that already contained assertions and were allowed to write new assertions with `jContract` (experiment group) or the code to reuse contained the information of the assertions only in the form of `JavaDoc` comments in natural language (control group). 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 (in particular program lines, program reliability, number of reused methods and quality of reuse).

4.4.1 Task

The task to be solved in this experiment is called "GraphBase". It consists of implementing the main class of a given graph library [Gol98] 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 are only accessor methods, e.g. for showing the number of vertices or edges, for finding an edge between two given vertices or for testing if the graph is empty, a weighted or a directed graph.
Each subject is told that the original code of GraphBase was lost and, because there is no backup, that it should be reimplemented by using the rest of the given graph library. The requirements for this task were descriped thoroughly in natural language. The subjects were asked to work and to test on their own until they thought they had finished the task.

4.4.2 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 an paper and consisted of four parts. The experiment group started with a web course to learn the syntax of `jContract`. The control group did not go through the course because they did not need `jContract` knowledge. Then, in part two, members of both groups were handed a task description and allowed to work on the task. The third part started when the experimental subject thought it had finished. At the end of the experiment, a questionnaire was handed out to every subject. 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 specific Unix account that provided the automatic monitoring infrastructure. It nonintrusively protocoled login/logout times, all compiled source versions and all output 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 subject 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 be done.

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 was to be implemented on paper and in electronic form. The control group was given a description for every method in natural language. The subjects were asked to check their implementation with this additional information and correct it if necessary.

4.5 Power analysis

Cohen [Coh77] stresses the importance of power analysis to get a closer look at the quality of a statistical hypotheses test.
The power of a statistical test of a null hypothesis is the probability that it will yield statistically significant results. It is defined as the probability that it will lead to the rejection of the null hypothesis, i.e., the probability that it will result in the conclusion that the phenomenon exists under the premise that the phenomenon is really existent. Statistically speaking, $1 - power$ is the probability for an error of the second kind.
EXP1 uses groups with $n = 4$ and $n = 5$ subjects. Due to this small number of data points, we restrict our analysis to large effects. In this case, Cohen suggests an effect of the size $ES = 0.8$. We set the significance level of the one sided test to $\alpha = 0.1$. Thus, the power analysis with a t-distribution yields a power of 0.410 [IG96]. The power analysis for EXP2 with $n = 6$ yields a power of 0.518. That is, we have only a 41% and 51% chance, respectively, to find a difference between the groups!
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 to assume 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 live with this drawback.

4.6 Threats to internal validity

The control of the independent variable is threatened by the possibility of an imbalanced group assignment – we might compare one group with faster programmers to one group with slower programmers. To avoid this effect, the group assignment was based on the PSP course productivity (the number of lines of codes programmed per hour in the PSP

course) of each subject. For both experiments, the division resulted in groups with similar productivity.

4.7 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 our 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.

5 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 associated with the dashed line points to the mean within a range of one standard error on each side.
Significance was calculated with the Wilcoxon-Test where the significance p denotes the probability that the observed difference is due to chance.

5.1 Results of first experiment

5.1.1 Number of assertions

Before reading about the effects of assertions, it might be interesting to look at how willing the participants of the first experiment were to use assertions, and how often the checks of these assertions failed. See table 1 for these data.

5.1.2 Durations of the work

We first present the amounts of time needed for the tasks separately for the groups using APP and those not using APP. In order to eliminate the effect of different programming

Participant Number	Number of preconditions	Number of postconditions	Number of `assert` conditions	Failed checks for chain rule task	Failed checks for string task
23	1	3	3	504	0
24	2	5	0	0	0
25	5	3	13	0	0
26	2	3	0	27	0
27	4	3	0	33	0
28	3	1	0	0	20
30	1	3	0	0	0
31	1	2	0	15	0
32	0	0	0	0	0

Table 1: Assertions and failed checks

speeds, we then present the amounts of time measured in multiples of the time needed for completing the APP training instead of in minutes.
For calculating how long the participants worked on their tasks, the beginning and end times were recorded automatically by the same scripts that provided the participants with the required files and decided whether their solutions were correct. Only for the time that had to be subtracted for interruptions in the work, the handwritten notes of the participants were used.
An unexpected phenomenon when using assertions was that for the group using assertions, the distribution of durations was more dense. The use of assertions therefore might make software development more predictable. See table 2 for details.

	$\frac{\text{upper quartile}}{\text{lower quartile}}$ *with assertions*	$\frac{\text{upper quartile}}{\text{lower quartile}}$ *without assertions*
chain rule	1,82	2,24
string functions	1,43	5,44

Table 2: Quartile ratios for durations

Here the quotients of the 75 % quantile and the 25 % quantile are list ed. A quotient close to 1 is desirable because then the duration of the software development is quite predictable.

Figures 1 and 2 do not show significant differences that would be caused by the use of

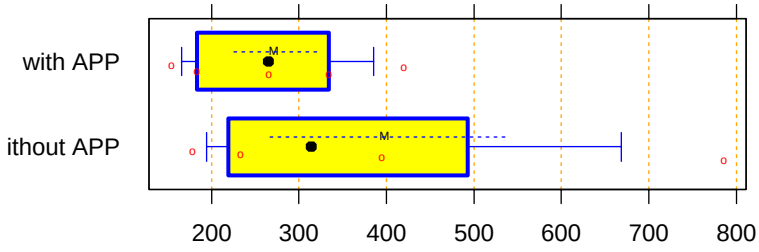


Figure 1: Duration for the chain rule task with and without assertions, measured in minutes.

The Wilcoxon test shows that the difference of medians is not significant (probability for an accidental difference: 0.55). The means are different, though: With `APP`*, 271 minutes are needed on average, while without assertions, 398 minutes are needed.*

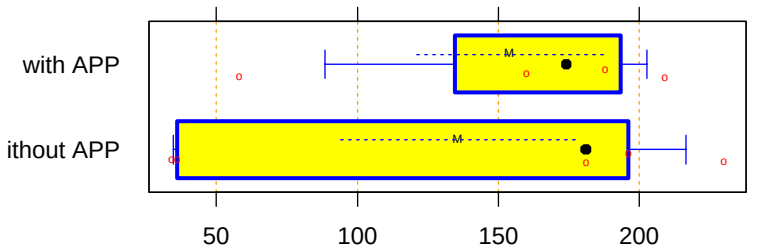


Figure 2: Duration for the string task with and without assertions, measured in minutes.

Result of the Wilcoxon test: with a probability of 0.9, the difference of medians is only accidental.

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, therefore the time spent on the APP training is a good measure for the programming speed. 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 differences in programming speeds is lowered and the influence of the use of assertions becomes more visible.

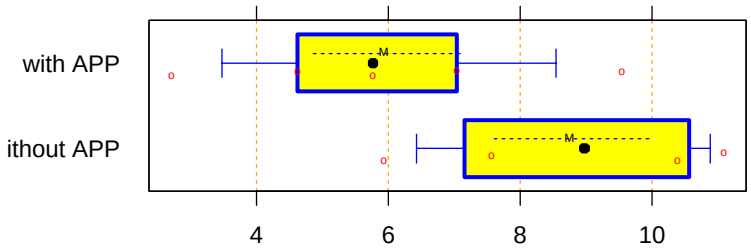


Figure 3: Relative durations for the chain rule task with and without assertions

For example, among those who did not use `APP`*, the fastest participant needed 6 times longer for completing the chain rule task than for completing the APP training. His data point is the leftmost one in the lower half.*

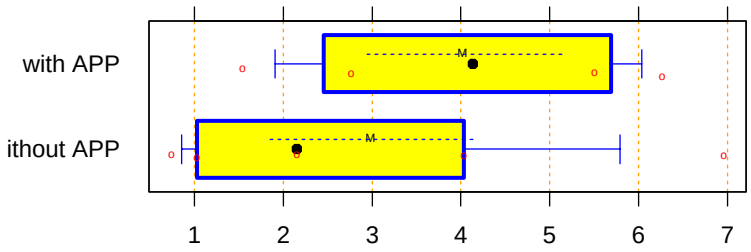


Figure 4: Relative durations for the string function task with and without assertions

The difference visible in figure 3 is significant. The Wilcoxon test shows that the probability for an accidental difference is 0.063. Therefore, assertions seem to save time when software

is maintained, while they tend to increase the effort needed for writing new software. The difference visible in figure 4 is not significant (Wilcoxon test result: 0.556).

5.1.3 Reuse of functions

Since we were interested in the number of reused functions as opposed to the number of function reuses – the research question is how the use of assertions contributed to reusing many different functions, not how reuseable the reused functions were – they were counted in the following way: For each participant, the final version of the extended program was compared to the version they started with 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 task with maintenance character, the chain rule assignment, is shown in figure 5. The difference is significant: the probability for an accidental difference is 0.0688. Users of APP reused 8.6 functions on average, while programmers who did not use assertions found only an average of 6.75 functions that seemed worth reusing. With a probability of 0.8, the use of assertions increases reuse of existing functions by 15 %.
The described counting method for function reuse included functions that were reused only within assertions. If only functions that were reused outside assertions are counted, users of APP reused 7.6 functions on average.

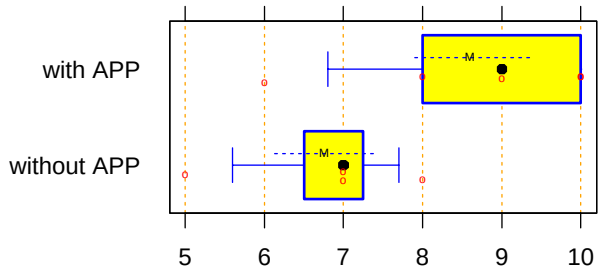


Figure 5: Number of reused functions for the chain rule task ($p = 0.07$).

5.2 Results of second experiment

5.2.1 Number of assertions

Table 3 shows the number of assertions each of the subjects wrote during the experiment.

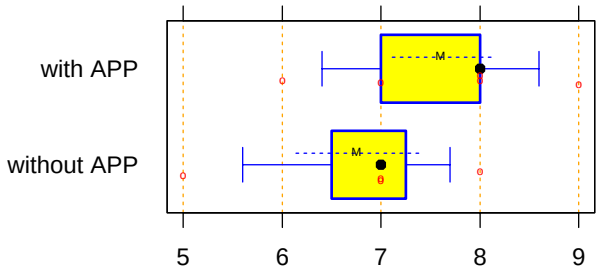


Figure 6: Number of reused functions outside assertions for the chain rule task ($p = 0.19$).

Participant Number	Number of pre-conditions	Number of post-conditions	Number of invariants	Number of assert-conditions	Total
101	43	21	0	0	64
102	3	41	0	0	44
103	17	3	0	0	20
104	0	0	0	0	0
105	30	39	5	0	74
106	0	43	0	0	43
108	17	0	0	0	17
201	0	0	0	0	0
202	0	0	0	18	18
203	0	0	0	0	0
204	0	0	0	0	0
205	0	0	0	0	0
206	0	0	0	0	0

Table 3: Number of assertions

The upper part of the table shows the number of the participants of the experimental group. Two rows are remarkable. Participant 104 ought to use jContract but he did not, and participant 202 of the control group used his own assertion environment.

5.2.2 Reliability

In this experiment, reliability was measured by determining the percentage of the passed assertions among all possible executable assertions in the test. The initial behavior of jUnit had to be adjusted to count all failed assertions. 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.
Reliability was measured for a synthetic test with 727,190 method invocations and about 7.5 million assertions. The reference implementation runs for about 150 seconds for this big 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. Deivations in the structure are caught by subsequent assertions.
First, we look at the reliability of the synthetic test for the final programs, see figure 7.

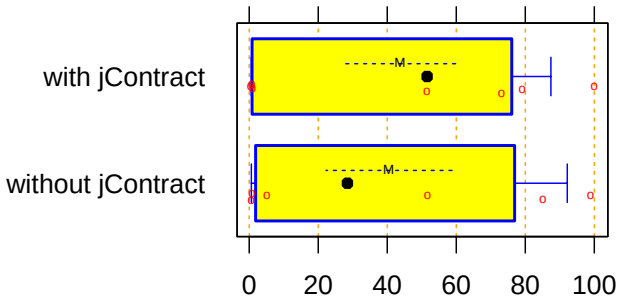


Figure 7: Reliability of the final programs for the synthetical-test in percent.

Here we cannot see any differences between the groups.
Now, the programs right after the IP are examined and the question is asked, 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 the subjects are most confident with concerning accurateness. The reliability of the program versions at the point when the subjects claimed to be done is shown in figure 8.
The reliability of the experiment group is higher $p = 0.112$. 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%.
We can say that the use of assertions is an advantage compared to informal information like the natural language documentation.

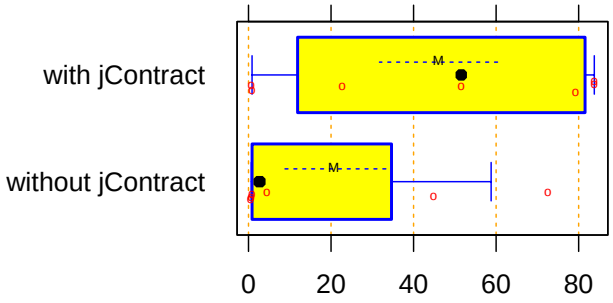


Figure 8: Reliability of programs after IP for the synthetical-test in percent.

5.2.3 Working time

We now present the working time needed for the IP. Because only the experiment group worked on the web course, this time 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 this group. The control group couldn't do this because they got the same information but in natural language, so they looked directly for defects in their program code after reading this information. If we compare only the minimum and the maximum of both groups, we can see that the control group needed between 24 and 55 minutes and the experiment group between 68 and 199 minutes for the CP.

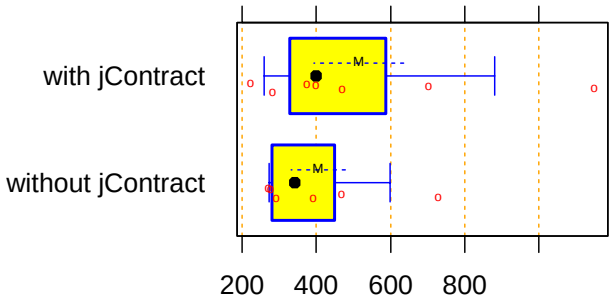


Figure 9: Working time in minutes.

In figure 9, we can see that the experiment group tends to need longer 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. But it worked only very slowly. With all other measures we compared, we could not see such an outlier effect.

5.2.4 Code reuse

Examining code reuse might lead to some results about program understanding. Four measures were collected to get a perception of it. There are (1) the number of reused methods, (2) the number of reused methods without the written 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 3.2). Their output was written to a log file, which the subjects did not notice.

Figure 10 and 11 show the results for the number of reused methods. In the figure 11 you can see the number of reused methods without the written assertion code. The first figure shows with $p = 0,23$ no difference in the number of reused methods. But there is a tendency that with assertions the subjects reused more methods. This tendency disapears if you ignore the maximum point at 32 in the experiment group. Figure 11 shows the groups if you ignore the assertion code. Now, there is a significant difference between the two groups: the experiment group reused less methods. An explanation of the difference could be that the validation tests for the method parameters are done either by assertions (experiment group) or by if-statements (control group). If you generally don't count the reuse within assertions, you ignore a significant aspect of the implementation. Since the overall reliability of the implementations is not comparable, as we have seen before, this could result in a difference for both figures, too.

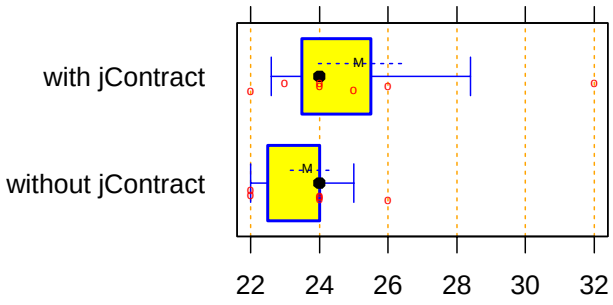


Figure 10: Number of reused methods.

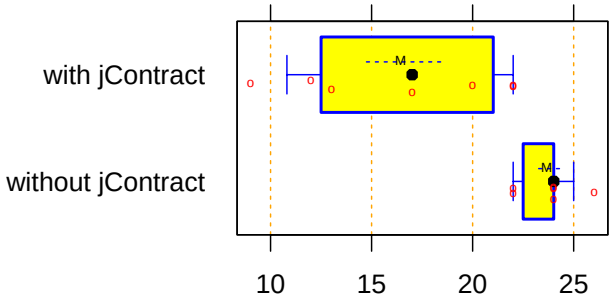


Figure 11: Number of reused methods outside assertions.

Finally, for the number of reused methods, we can not see a difference between the two groups.
Both groups made quite similar errors while reusing a method more than once. If we look at the figure 12 and 13 we can say that wrong reuse does not happen again that often in the experiment group.

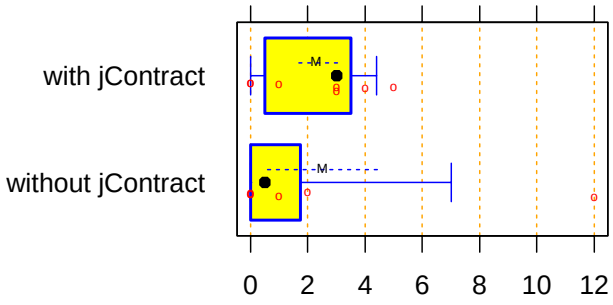


Figure 12: Number of assertions that failed at least once.

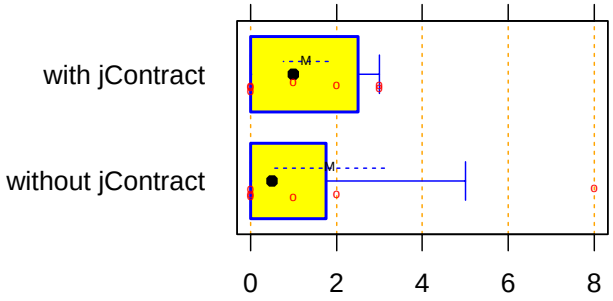


Figure 13: Number of assertions that failed at least twice.

6 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.

- Assertions reduce programming effort in maintenance if the maintenance task is defined as a program assignment that requires a deep understanding of the program to maintain.

- Assertions slightly increase the programming effort for the implementation of new functions that do not interact much with the rest of the program.

- When we look 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. But when we look at the programs after the implementation phase, the programs of the experimental group, i.e., the group that used assertions, were more reliable, though not statistical significant, than those of the control group.

- The Usage of assertions also led to a higher number of reused methods that weren't 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 be a hindrance not to see any 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 show quite different results. But overall, and this is a result applies to programmers who are new to assertions, using assertions decreases the programming effort in maintenance and increases the reliability of newly developed code with only a small amount of extra effort.

References

[Chr94] L. B. Christensen. *Experimental Methodology.* Allyn and Bacon, 1994.

[Coh77] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Academic Press, 1977.

[Coo97] J. Cook. Assertions for the tcl language. In *5th Tcl Workshop*, Boston, Massachusetts, July 1997.

[Gol98] D. Goldschmidt. Design and implementation of a generic graph container in java. Master's thesis, Rensselaer Polytechnic Institute in Tray, New York, April 1998.

[Hum97] W. Humphrey. *A discipline for software engineering.* Addison-Wesley, 1997.

[IG96] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[jav] JavaTM 2 SDK, standard edition, documentation, version 1.4.0. http://java.sun.com/j2se/1.4/docs/.

[LCKS90] N. Leveson, S. Cha, J. Knight, and T. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16(4):432–443, April 1990.

[LST91] D. Luckham, S. Sankar, and S. Takahashi. Two-dimensional pinpointing: Debugging with formal specifications. *IEEE Software*, 2(2):9–23, January 1991.

[McK96] J. McKim. Programming by contrac: Designing for correctness. *Journal of object oriented programming*, 9(2):70–74, May 1996.

[Mey88] B. Meyer. *Object-oriented software construction*. Prentice-Hall, 1988.

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

[Ros92] D. Rosenblum. Towards a method of programming with assertions. In *International Conference on Software Engineering*, pages 92–104, Melbourne, 1992.

[Ros95] D. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[Sch98] F. Schneider. On concurrent programming. *Communications of the ACM*, 40(4):128–128, April 1998.

[Stö99] J. Störk. Erzeugung effizienter laufzeitüberprüfungen von zusicherungen. Master's thesis, Department of Computer Science, University of Karlsruhe, 1999. http://www-is.informatik.uni-oldenburg.de/~stoerk/da/diplomarbeit.html. Only available in german.

[Sys] Reliable Systems. iContract. http://www.reliable-systems.com/tools/iContract/iContract.htm.

[Voa97] J. Voas. How assertions can increase test effectiveness. *IEEE Software*, pages 118–122, MarchApril 1997.

[xun] Xprogramming, software downloads. http://www.xprogramming.com/software.htm.

A Experimental Data of EXP1 and EXP2

Group	A capital C stands for "chain rule with assertions", a lower case c for "chain rule without assertions", a capital S for "string functions with assertions", and a lower case s for "string functions without assertions". The order of the two letters reflects the order of the tasks. For example, participant 23 belongs to group Cs, i. e. he first completed the chain rule assignment with assertions and then the string function assignment without assertions.
#pre	Number of preconditions
#post	Number of postconditions
#assert	Number of "assert" conditions
Fail	Number of failed assertions
DurC	Working time for the chain rule task in minutes
DurS	Working time for the string function task in minutes
RDurC	Relative duration for the chain rule task (DurC divided by the time spent on the web-based syntax course)
RDurS	Relative duration for the string function task (DurS divided by the time spent on the web-based syntax course)
Reused total	Number of reused functions
Reused outside	Number of reused functions outside assertions

Table 4: Data lines in Table 6

Rel1	Reliability of the programs after the IP in percent
Rel2	Reliability of the final programs in percent
Dur	Working time in minutes
Reuse1	Number of reused methods
Reuse2	Number of reused methods outside assertions
Fail1	Number of assertions that failed at least once
Fail2	Number of assertions that failed at least twice

Table 5: Data lines in Tables 7 and 8

subject no.	23	24	25	26	27	28	30	31	32
Group	Cs	cS	Sc	sC	Cs	cS	Cs	sC	cS
#pre	1	2	5	2	4	3	1	1	0
#post	3	5	3	3	3	1	3	2	0
#assert	3	0	13	0	0	0	0	0	0
Fail	504	0	0	27	33	20	0	15	0
DurC	154	233	395	265	334	178	183	420	786
DurS	230	58	209	34	36	188	181	196	160
RDurC	2.7	11.1	10.4	5.8	9.5	5.9	7.0	4.6	7.6
RDurS	4.0	2.8	5.5	0.7	1.0	6.3	7.0	2.2	1.5
Reused total	6	5	7	9	10	8	10	8	7
Reused outside	6	5	7	8	8	8	9	7	7

Table 6: Data of EXP1

subject no.	101	102	103	104	105	106	108
Rel1	51.53	79.24	0.79	0.96	83.70	83.70	22.85
Rel2	51.53	79.24	0.87	0.96	100.00	73.17	0.43
Dur	702	471	399	1148	375	223	282
Reuse1	24	32	24	22	23	25	26
Reuse2	13	17	20	22	12	9	22
Fail1	5	4	3	3	1	0	0
Fail2	3	2	1	3	0	0	0

Table 7: Data for experimental group of EXP2

subject no.	201	202	203	204	205	206
Rel1	0.83	72.58	0.59	44.84	0.96	4.49
Rel2	0.83	85.25	0.60	51.85	5.16	99.12
Dur	270	291	276	469	729	392
Reuse1	26	24	22	24	22	24
Reuse2	26	24	22	24	22	24
Fail1	0	2	0	1	12	0
Fail2	0	2	0	1	8	0

Table 8: Data for control group of EXP2