

A Controlled Experiment for the Assessment of Aspects Tracing in an Industrial Context

Pascal Dürr, Lodewijk Bergmans, Mehmet Akşit

University of Twente,
PO Box 217,
7500 AE Enschede,
The Netherlands
`{durr,bergmans,aksit}@ewi.utwente.nl`

Abstract. For years the Aspect Oriented Software Development community has promoted aspects as a solution to complexities introduced by implementing crosscutting concerns. And while a lot of research focuses on advancing the state of the art of aspect oriented programming and design, there is still no (reported) wide-spread industrial adoption of Aspect Oriented Programming. In this paper we report on an experiment to quantify the aspect-based approach to *Tracing*. The experiment was performed in an industrial setting at ASML. We believe that the example is a stereotypical aspect-oriented problem and as such the results of this experiment are very likely generalizable to other aspects. Participants of the experiment were requested to carry out five simple maintenance scenarios, related to *Tracing*. The result of this experiment demonstrates 6% and 77% reduction in development effort and errors, respectively. The statistical significance analysis has confirmed these reductions for a subset of the individual scenarios.

1 Introduction

Aspect Oriented Software Development (AOSD) has been promoted as a solution to complexities introduced by implementing crosscutting concerns. However, there is still no (reported) wide-spread industrial adoption of Aspect Oriented Programming (AOP).

Numerous papers [1,2,3,4] have stated that lack of industrial success stories and lack of experimental validation of the claims are two important barriers to wide-spread adoption of AOP by industry. For example, in the *Informal Workshop Proceedings of the first Workshop on Assessment of Aspect-Oriented Technologies* [3], the following comment was made. “Several workshop participants raised the issue that empirical evidence based on industrial-strength applications is crucial to convince the industry to adopt new AO techniques, and that there is a lack of such experiments in the AOSD field.”. Similarly, in *How to Convince Industry of AOP*, by Wiese et. al. [2], the following quote can be found: “We see a kind of a vicious circle here: Industry needs large scale success stories to

be convinced. But, to produce such success stories you have to apply AO in industrial projects.”. With this paper, we aim at providing reliable evidence of the advantages and possible limitations of the AOP languages so that adoption of this technology is based on more rational arguments but not locked up in vicious reasoning. We also explain how this experiment was designed and setup. The participants of the experiment were 20 software developers at ASML, which is a leading company in producing chip manufacturing machines.

The experiment was based on dealing with concern *Tracing*. This concern was selected for three reasons. Firstly, concern *Tracing* is widely required in the machines manufactured by ASML, for example in realizing error handling and control. This has provided a realistic context for the experimentation. Secondly, although it looks like a simple concern, *Tracing* shows sufficient complexity due to its divergence. Thirdly, *Tracing* shows the typical characteristics of a crosscutting concern and as such the results of the experiments are likely to be generalizable to other concerns. The result of this experiment demonstrates 6% and 77% reduction in effort and error in implementing and maintaining concern *Tracing* with respect to the conventional techniques.

1.1 Contributions

This paper contributes in the following ways:

- A detailed description of a real-world aspect: *Tracing*. Although this is usually considered simple, we show that this aspect is complex (Section 2).
- A design of a controlled experiment that can be used to quantify the benefits of using an aspect-based approach to *Tracing* in an industrial setting (Section 3).
- The results of a controlled experiment with 20 professional software developers, using an aspect-based approach to *Tracing* (Section 4).

2 Tracing

AOP is suitable to address many (complex) crosscutting concerns besides *Tracing* and *Logging*, as motivated in [5]. *Tracing* was chosen as the main driver for the adoption of AOP by ASML. The concern accounted for around 7% [6] of LLOC, for one specific component. Although this varies per component, the amount of scattered and replicated code of idiom *Tracing* is large. Idiom *Tracing*, at first glance, seems to be a ”simple“ and ”trivial“ aspect. However, in practice this is not the case. The functionality implemented by concern *Tracing* is required for effective diagnostics of the machines. As such, moving to an AOP solution should exceed, or at least match, the requirements stated for concern *Tracing*. We only elaborate on those details of concern *Tracing* and its aspect implementation, which are relevant to the developers of the base code. First, we show the idiomatic solution as implemented before the introduction of *WeaveC*. Second, we present aspect Tracing, which is now in use at ASML and which was used for the experiment.

2.1 Concern Tracing

We show an example of idiom *Tracing* as it is currently implemented in C. The current tracing implementation uses the so-called THXA framework. This is a tracing framework developed internally at ASML.

```
1 int CCXA_change_item_ids(ITEM_DEF *item_def_1, ITEM_DEF *item_def_2)
2 {
3     int result = OK;
4     int item_id = LO_UNDEFINED_ITEM_ID;
5
6     THXAtrace("CC", TRACE_INT, __FUNCTION__, "> (item_id_1 = %d, item_id_2 = %d, active = %b)",
7             item_def_1 != NULL ? item_def_1 -> item_id : 0,
8             item_def_2 != NULL ? item_def_2 -> item_id : 0, MACHINE_IS_ACTIVE);
9
10    if(result == OK && ( item_def_1 == NULL || item_def_2 == NULL )
11    {
12        result = CC_PARAMETER_ERROR;
13    }
14
15    if(result == OK && MACHINE_IS_ACTIVE )
16    {
17        item_id = item_def_1 -> item_id;
18        item_def_1 -> item_id = item_def_2 -> item_id;
19        item_def_2 -> item_id = item_id;
20    }
21
22    THXAtrace("CC", TRACE_INT, __FUNCTION__, "< (item_id_1 = %d, item_id_2 = %d) = %R",
23            item_def_1 != NULL ? item_def_1 -> item_id : 0,
24            item_def_2 != NULL ? item_def_2 -> item_id : 0, result);
25
26    return result;
27 }
```

Listing 1. Tracing example in C

Listing 1 declares one function called: `CCXA_change_item_ids`. This function changes two identifiers of two items. Lines 17 to 19 show the implementation of these two item identifiers. This function also implements error handling, see lines 3, 10, 12, 15 and 26. Idiom *Parameter Checking* is implemented at lines 10-13. More interestingly are lines 6-8 and 21-24. At these lines two trace calls are stated, using function `THXAtrace`.

Trace call `THXAtrace` at lines 6-8, has several arguments. The first is a textual representation of the component in which this file is located, in this case `CC`. Secondly, a constant is passed to indicate whether this is an internal or external tracing call. For this paper, this distinction is not relevant. Next a GCC meta variable (`__FUNCTION__`) is used that is substituted with the name of the function at compile-time. The fourth argument is a format string which controls how this trace entry should look. In this case we are tracing the item identifiers of both arguments and a global variable called `MACHINE_IS_ACTIVE`. The next two arguments are the actual argument values which should be inserted in the appropriate places in the format string. Note that we also verify that the arguments are not `NULL`. If they are `NULL` and we do refer to the field `item_id`, this could cause a segmentation fault in the system, effectively shutting down the machine. Implementing a guard ensures that if the arguments are null, a zero is printed in the trace file. Since global variable `MACHINE_IS_ACTIVE` is being read

inside this function, we pass this variable as the last argument of the `THXATrace` call.

The trace call at lines 21-24 is similar to the former trace call. One difference is that since global variable `MACHINE_IS_ACTIVE` has not been changed in this function, we no longer have to trace this variable. Another difference is that we trace the return value of this function, the return value indicates whether the function executed successfully. Similar to the previous trace call, the two item identifiers are traced, as these may be to be modified in this function. Again, we check whether the two arguments are not `NULL`

In general all functions within the codebase are to be traced, in the above illustrated manner. There are some exceptions, e.g. there is a subset of functions which cannot be traced, as the tracing framework has not been initialized yet when these functions executed. All parameters of all traceable functions should be traced. The parameters of a function include the following:

- Function arguments that are either accessed, manipulated, or both in this function.
- Global variables that are either accessed, manipulated, or both in this function.
- The return value of a function.

Furthermore, we distinguish between parameters that are input, output or both:

- *Input* parameters are those parameters which are only read.
- *Output* parameters are those parameters which are only manipulated, this includes those parameters passed to other functions.
- *InOutput* parameters are those parameters which are read and manipulated,

Concern Tracing should behave as follows:

- At the start of a function, trace:
 - the name of the component,
 - an internal or external trace specifier,
 - the name of the function,
 - *Input* and *InOutput* parameters.
- At the end of a function, trace:
 - the name of the component,
 - an internal or external trace specifier,
 - the name of the function,
 - *Output* and *InOutput* parameters.

Thus just described behavior shows that idiom *Tracing* is far from trivial. Even modern AOP languages have problems addressing this effectively. Implementing this concern requires detailed control and dataflow analysis to determine the *Input*, *Output* and *InOutput* classification for all arguments and variables that are used inside a specific function. Most AOP approaches do not support this detailed analysis.

2.2 Aspect Tracing in WeaveC

Now that the requirements of concern *Tracing* have been elaborated, we present the details of the aspect specification that are required to understand the ex-

periment. We use an industrial-strength C weaver developed by ASML, called *WeaveC*, and an AOP language, called *Mirjan*, these are detailed in [7].

The requirements of concern *Tracing* above can be implemented solely in a single aspect specification. However, there are cases where developers need to deviate from the idiom. To allow such deviations, *WeaveC* supports annotations. The following annotations have been defined for aspect *Tracing*:

- $\$trace(TRUE | FALSE)$: controls if a module(file), function, parameter, variable or type should be traced.
- $\$trace_as(fmt = "...", expr = "...")$: traces a parameter, variable or type in a different manner.

In short, aspect *Tracing* can be defined as:

- All functions should be traced
 - Except for those functions, which are annotated with $\$trace(FALSE)$ or whose module is annotated with $\$trace(FALSE)$.
- For each traceable function, trace the input parameters at the start and the output parameters at the end of the function.
 - Except for those parameters, which are annotated with $\$trace(FALSE)$ or whose type is annotated with $\$trace(FALSE)$.

3 Experiment Setup

We first informally present the setup of the experiment. For the statistical results we used the theory and guidelines as presented in [8] and [9]. The experiment was conducted in combination with a training about *WeaveC*. Figure 1 presents an overview of the training and the experiment.

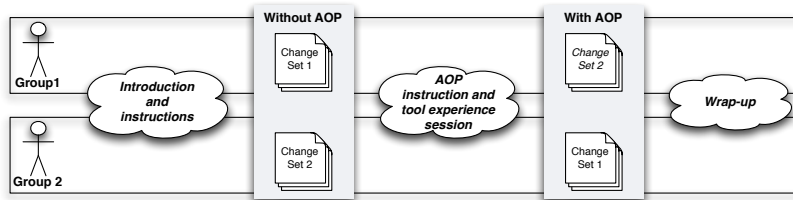


Fig. 1. Overview of the experiment and training. Note that Change Set 1 and Change Set 2 are the same type of change requests on different source codes.

The goal of the experiment was to determine whether using an aspect-oriented approach to *Tracing* helps to reduce the development and maintenance effort. We determined this by measuring both the time it takes to implement tracing related change scenarios, and the errors introduced while implementing these scenarios.

The subjects were split into two groups while executing the change scenarios. To prevent possible differences between the subjects in the groups we, all subjects were present at the instructions. The introduction included not only the overview of the training but also a brief explanation of idiom *Tracing*. This ensured that all subjects had the same notion of tracing. During the introduction a balanced selection of groups was made, based on the years of software engineering experience and the years of working at ASML. Section 5 provides a discussion about the differences between the groups.

After the introduction, the groups were split. Group 1 executed change scenario set 1, while group 2 executed change scenario set 2. Both groups had to implement 5 change scenarios using the current, manual, way of tracing. Change sets 1 and 2 contained similar scenarios. However, the code base on which the scenarios were executed was distinct. The subjects were allowed to spend at most 45 minutes on this part of the experiment.

After the first session, a general introduction into AOP and a specific explanation on the way of working with *WeaveC* was presented, again all subjects were present. Next, the subjects had the opportunity to familiarize themselves with the new process and tooling, using a set of simple exercises.

Subsequently, group 1 had to implement change scenario set 2 using *WeaveC*, while group 2 implemented change scenario set 1 using *WeaveC*. Again the subjects executed up to 5 scenarios within 45 minutes. We used this so-called *cross-over design* to prevent or minimize learning effects between sessions without *WeaveC* and with *WeaveC*. Section 5 discusses the implications of this choice on the validity of the experiment.

For each completed scenario, we determined the time and the error classification. We used the command logs and version control information.

3.1 Subjects

The experiment was conducted as part of a training of *WeaveC*. As such we had no control over the selection of the subjects. The possibility of a training was published within the company to the software group leaders. Subsequently, subjects could register themselves for the training.

We have gathered some characteristics of the subjects. We asked the subjects to fill in the characteristics before the training started. These were:

- Years of experience in software development
- Years working at ASML
- Gender (0=Male, 1=Female)
- Age
- Education Level
- C Level
- Regular THXA user

Note that *THXA* is the current tracing framework.

We used the characteristics from two reasons. Firstly, it allowed us to find possible interesting correlations between the performance of the subjects and these characteristics. Secondly, we distributed the subjects into two groups. We used the sum of the years of software development and years working at ASML

to distribute the subjects into two groups. We made an ordered list with these sums. Next we assigned subjects to groups in an alternating fashion. This process provided a somewhat balanced selection of groups.

3.2 Environment and Tooling

The experiment and training were conducted at an external location. The two groups were both located in different rooms. During the presentations and instructions all persons were in the same room.

We used an external location to prevent the impact of the working environment on the experiment. The subjects could use their own build environment and tools, using remote log-in. This ensured that the build performance would not influence the experiment as the subjects used the build farm at the company. We restricted the usage of shell aliases, to prevent overriding our command logger and timing facilities.

We gathered data from two data sources. The first was a list of commands which were entered in the terminal, with time stamps. In addition the exit codes of the build commands were logged. An example is shown here:

```
1 Tue Jul 3 09:34:37 MEST 2007, cleartool co -nc CCEWdata.c
2 Tue Jul 3 09:44:40 MEST 2007, > ccmake CCEWdata.osparc
3 Tue Jul 3 09:45:13 MEST 2007, < ccmake 1
4 Tue Jul 3 09:45:39 MEST 2007, > ccmake CCEWdata.osparc
5 Tue Jul 3 09:46:04 MEST 2007, < ccmake 0
6 Tue Jul 3 09:46:18 MEST 2007, cleartool ci -nc CCEWdata.c
```

Line 1 states a checkout of file CCEWdata.c. After one minute the subject proceeded to build the file (line 2). This build fails after 30 seconds (exit code 1). After 20 seconds the subject builds the file again, see line 4. This build executed successfully (exit code 0), see line 5. Finally, at line 6 the user checks in the file.

The second data source was the check-in and checkout information from a version control system. We used this as a backup, if for some reason the key logger did not work. Here is a line from this data source, corresponding the above mentioned data source of file CCEWdata.c:

```
1 CCEWdata.c;20070703.115123;20070703.094618;20070703.093438;20070703.093438
```

This line has to be read from right to left. One can see the same time stamps as in the previous example. To determine the error classification, we examined the committed version of the edited source files to classify the changes the subjects had made to the functions in the files.

3.3 Treatments

We used change requests as treatments to determine the possible gain of using *WeaveC*. We only selected change requests which affect *Tracing*. Since there was only limited time to execute the scenarios, we selected the top four requests. This selection was made with input from ASML to decide which requests were the most frequently occurring and most valuable, according to ASML. These were the selected change requests:

1. Add tracing to a (traceless) function.
2. Change the parameters of a function.
3. Remove tracing from a function.

4. Selective tracing - only trace a specific parameter.
5. Remove a function.

We added a fifth scenario, that was executed first, to account for an initial overhead (see section 5). Note that we did not include a scenario related to changing idiom *Tracing*. We only focus on the scenarios which impact the developers of the base code. Only a small set of developers will write or change aspects. The vast majority will write base code that is subjected to aspects. Also, manually changing idiom *Tracing* is not often done. With AOP this becomes a lot easier but this is out of the scope of this experiment. For each scenario we will now show what is required to fulfill these requests without and with AOP.

1. Add tracing to a traceless function:

Without WeaveC: Add the appropriate tracing code to the function, accounting for parameter usage and null pointer checks.

With WeaveC: Nothing

2. Change the parameters of function: (from Input to InOutput)

Without WeaveC: This requires adding null pointer checks and adding or altering tracing at the end of a function. The subjects were instructed not to reflect this change in the code.

With WeaveC: Nothing

3. Remove tracing from a function:

Without WeaveC: Remove trace statements from the function.

With WeaveC: Attach a `$trace(FALSE)` annotation to the function.

4. Selectively tracing - Only trace a specific parameter:

Without WeaveC: Alter the trace statements to reflect this situation, removing all the other parameters from these statements.

With WeaveC: Attach a `$trace(FALSE)` annotation to all parameters except the one which should be traced.

5. Remove a function: This is used to reduce the initial overhead and to measure possible learning effects between scenarios.

Without WeaveC: Remove function body.

With WeaveC: Remove function body.

Note that in the last scenario, we cannot remove the entire function, since the call sites would have to be adjusted then as well to enable a successful build.

3.4 Objects

The code we used in our treatments is code taken from the ASML codebase. We used the largest component(CC) as a source for objects. We measured several metrics of the code to ensure that we selected representative objects. We used three metrics to select representative functions: *Number of parameters*, *Lines of Code* and *McCabe cyclometric complexity*. The first as the complexity of tracing directly relates to the number of parameters. The latter two are well established metrics for the perceived complexity of source code[10].

Number of Parameters Figure 2 shows the distribution of the number of parameters within component CC. There are a lot of functions which have one parameter. This is mostly caused because the return value of the functions is

used for error handling and even simple “getter” functions use a parameter. We chose to select only functions with between 2 and 4 parameters.

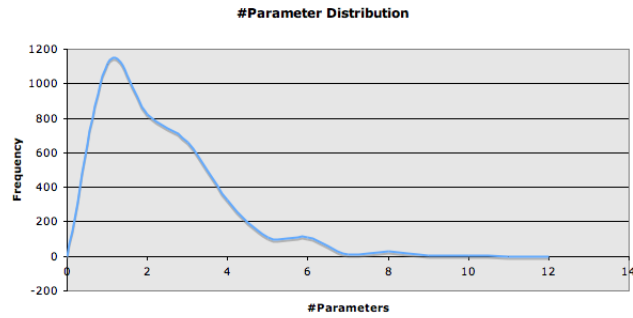


Fig. 2. Parameter Distribution

Lines of code(LOC) Figure 3 shows the distribution of LOC within component CC. We chose to select only those functions that have between 40 and 55 lines of code. This was based on the graph and the average (47.08),

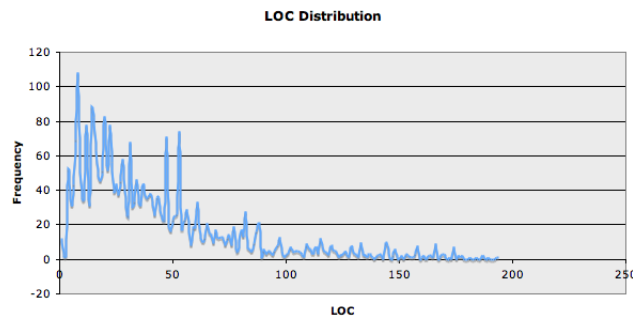


Fig. 3. LOC Distribution

McCabe cyclometric complexity Figure 4 shows the distribution of the McCabe complexity of functions within component CC. Based on the graph and the average(6.16), we chose to select only functions with McCabe complexity between 5 and 10.

Table 1 provide the values for these metrics for each object and treatment combination. This combination is called a *Change Scenario*.

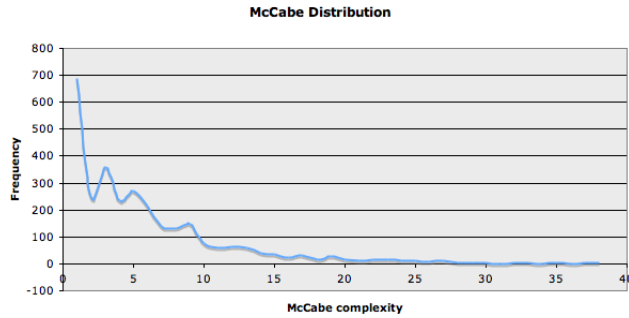


Fig. 4. McCabe Distribution

Change Set	Change Scenario	Description	#Params	#LOC	McCabe
CS1	CS1.1	Add tracing to a function	3	44	9
CS1	CS1.2	Change the parameters of a function	2	49	6
CS1	CS1.3	Remove tracing from a function	3	44	5
CS1	CS1.4	Selectively trace a parameter	3	53	8
CS1	CS1.5	Remove a function	3	53	5
CS2	CS2.1	Add tracing to a function	2	50	6
CS2	CS2.2	Change the parameters of a function	4	48	8
CS2	CS2.3	Remove tracing from a function	2	48	5
CS2	CS2.4	Selectively trace a parameter	2	51	8
CS2	CS2.5	Remove a function	3	52	7

Table 1. Metric values for each change scenario.

During the design of the experiment we had a rough idea of the average time of the execution of all five scenarios. This was 42 minutes. There was a change the subjects would take a lot more time than anticipated. We used a paired sample test for analyzing the experiment. Therefore, we had to ensure that we had the most paired scenarios from the subjects. We also wanted to ensure that we had results for the most important set of scenarios. This selection was made with input from ASML, to get the most valuable information out of the experiment.

We used a process called clustered randomization to randomize the scenarios. *CS5* was always put first, this captures the initial overhead. Next *CS1* and *CS2* were randomly selected. Finally, *CS3* and *CS4* were randomly selected. The following permutations were possible:

- CS5 CS1 CS2 CS3 CS4
- CS5 CS1 CS2 CS4 CS3
- CS5 CS2 CS1 CS3 CS4
- CS5 CS2 CS1 CS4 CS3

We assigned the order within a change scenario set to the subjects at random.

An example scenario We show part of an instruction sheet for one scenario, in this case CS1.1: Change Set 1 - Adding tracing to a function.

1. Navigate to directory: `CC/CCQM/int/bin`:

```
1 cd CC/CCQM/int/bin
```

2. Check out file: CCEWdata.c:

```
1 cleartool co -nc CCEWdata.c
```

3. Locate the following function: CCEWDA_set_wafer_state,

4. Generate tracing code for this function.

5. Build the file:

```
1 ccmake CCEWdata.osparc
```

6. In case of build errors, please go to step five and fix these and rebuild the system, until there are no more build errors.

7. Check in file: CCEWdata.c

```
1 cleartool ci -nc CCEWdata.c
```

3.5 Variables

Factors WeaveC is the only factor of this experiment. This factor is measured in the nominal scale, at two levels: without WeaveC and with WeaveC.

Independent Variables The independent variables of our experiment are:

- **Years of experience in software development:** Numeric
- **Years working at ASML:** Numeric
- **Gender (0=Male, 1=Female):** Numeric
- **Age:** Numeric
- **Education Level (4=PhD, 3= MSc, 2=BSc, 1=Practical Education):** Numeric
- **C Level (5=Expert, 1=None):** Numeric
- **Regular THXA user (1=Yes, 0=No):** Numeric

Dependent Variables The dependent variables of our experiment are:

- **Time** to execute a change scenario, in secs: Numeric
- **Error** classification of a change scenario: Numeric

Error classification We chose to use an error classification in stead of simply counting the number of errors. Most of the time only one error is introduced into the idiom code. Also, the impact of the errors differ. Therefore, we used the following error classification:

- 0** : No errors,
- 1** : Typo in tracing text string,
- 2** : Tracing less than is required,
- 3** : Wrong tracing,
- 4** : No parameter checking code.

An error of class 4 can result in a segmentation fault, and is as such considered the worst case. A higher number indicates a more severe error. In case of two errors we used the worst case, this only occurred once in the dataset.

3.6 Hypotheses

We defined the following null hypotheses:

- *WeaveC* does not reduce the development time, while developing and maintaining code related to *Tracing*: $H_0^{time} : time_{without_WeaveC} \leq time_{with_WeaveC}$
- *WeaveC* does not reduce the severity of errors, while developing and maintaining code related to *Tracing*: $H_0^{error} : error_{without_WeaveC} \leq error_{with_WeaveC}$

For this experiment we defined the following alternative hypotheses:

- *WeaveC* does reduce the development time, while developing and maintaining code related to *Tracing*: $H_a^{time} : time_{without_WeaveC} > time_{with_WeaveC}$
- *WeaveC* does reduce the severity of errors, while developing and maintaining code related to *Tracing*: $H_a^{error} : error_{without_WeaveC} > error_{with_WeaveC}$

These hypotheses will be tested in the next section for each of the five scenarios.

4 Experiment Results

We split the total group into two groups. The training was scheduled for one day with a morning and an afternoon session.

4.1 Subjects

Table 2 presents to so-called descriptives of the subjects, these contain; the number of scenarios, the minimum values, the maximum value, the mean and the standard deviation.

Characteristic	N	Min	Max	Mean	Std. Dev
Age	17	26	45	34.88	5.73
Education	17	2	3	2.41	0.51
JarenASML	17	0	7	3.40	2.78
Years in SW development	16	1	20	9.12	5.62
Clevel	17	1	5	3.71	0.99
THXA	17	0	1	0.59	0.51

Table 2. Descriptives of the Subjects

In the morning session, two subjects were accidentally included in group one. This is why, in the morning session, group one has more subjects than group two. Section 5 discusses this discrepancy more thoroughly. Furthermore, we excluded three subjects and a series of data points, see section5 for more details.

4.2 Initial processing

Since the number of subjects is low, it is hard to get statistical significant results. Before we present these statistical results, we will first present our observations based on these results.

Table 3 presents, for the development effort, the number of data points and the average effort of each scenario respectively without, and with, *WeaveC*, as well as the delta for these scenarios. Note that a negative delta indicates that the subjects executed the scenario faster with *WeaveC*.

Table 4 presents, for severity of error reduction, the number of data points and the average of each scenario without and with *WeaveC*, as well as the delta for

Scenario	without WeaveC		with WeaveC		Effort Δ
	#	Average	#	Average	
Adding Tracing	15	824	17	328	-60.2%
Changing Parameters	14	523	15	422	-19.4%
Removing Tracing	9	143	11	230	60.6%
Selectively Tracing	8	292	13	227	-22.2%
Remove Function	17	250	17	185	-26.2%

Table 3. Development Effort

these scenarios. Note that a negative delta indicates that the subjects introduced less errors with *WeaveC*.

Scenario	without WeaveC		with WeaveC		Error Δ
	#	Average	#	Average	
Adding Tracing	15	0.67	17	0.00	-100.0%
Changing Parameters	14	2.57	15	0.20	-92.2%
Removing Tracing	9	0.00	11	0.00	0%
Selectively Tracing	8	1.38	13	0.92	-32.9%
Remove Function	17	0.00	17	0.00	0%

Table 4. Errors

Observations Based on the results in tables 3 and 4, we observe the following:

- Overall, users were able to implement 10 more change scenarios with the use of *WeaveC*.
- Overall, users were 6% faster with the use of *WeaveC*.
- Overall, users made 77% less severe errors with the use of *WeaveC*.
- Most scenarios require less effort with *WeaveC* than without *WeaveC*.
- Removing tracing from a function requires more effort with *WeaveC* than without. This is probably caused by the lack of experience in using the newly introduced annotations.
- Adding tracing and changing parameters introduces almost no errors with *WeaveC*, whereas manually implementing these change resulted in more severe errors.
- Selectively tracing a function with *WeaveC* introduces less severe errors than without *WeaveC*.
- *WeaveC* introduces no new errors while removing tracing and a function.
- The manual implementation contained 8 critical errors, which could have resulted in a segmentation fault during run-time.

Table 5 presents the distribution of the errors for the five scenarios, without and with *WeaveC*. For each error class the number of errors for this class is presented in the corresponding cells.

Scenario	without WeaveC				with WeaveC					
	0	1	2	3	4	0	1	2	3	4
Adding Tracing	7	1	1	2	0	15	0	0	0	0
Changing Parameters	3	3	0	0	7	15	0	0	0	0
Removing Tracing	9	0	0	0	0	11	0	0	0	0
Selectively Tracing	4	0	2	1	1	9	0	0	4	0
Remove Function	17	0	0	0	0	17	0	0	0	0

Table 5. Error Distribution over the scenarios

4.3 Development Effort

We now present the statistical results from the experiment. We used SPSS Version 13.0 for Windows[11] to process the raw data and to execute the tests. We have removed all statistical outliers from the data-set, see section 5.

Descriptives Table 6 presents several details for all scenarios, namely; the number of scenarios, the minimum values, the maximum value, the mean and the standard deviation.

Scenario	N	Min	Max	Mean	Std. Dev.
Adding Tracing w/o <i>WeaveC</i>	11	254	902	583	227
Changing Parameters w/o <i>WeaveC</i>	12	76	648	412	177
Removing Tracing w/o <i>WeaveC</i>	9	91	210	143	45
Selectively Tracing w/o <i>WeaveC</i>	7	142	339	249	72
Remove Function w/o <i>WeaveC</i>	15	84	347	189	80
Adding Tracing with <i>WeaveC</i>	13	106	379	218	79
Changing Parameters with <i>WeaveC</i>	15	153	910	422	243
Removing Tracing with <i>WeaveC</i>	10	59	333	217	86
Selectively Tracing with <i>WeaveC</i>	13	89	421	227	120
Remove Function with <i>WeaveC</i>	15	106	253	179	46

Table 6. Descriptives of Effort

Significance We calculated the significance through the use of a standard paired sample test with a confidence interval of 95%. Table 7 shows the results for each scenario.

Scenario	N	Mean	Std. Dev.	Sig.(2-tailed)
Adding Tracing	8	500	179	0.00
Changing Parameters	11	-47	370	0.68
Removing Tracing	6	-91	75	0.03
Selectively Tracing	6	95	78	0.03
Remove Function	13	16	69	0.41

Table 7. Significance of Effort

Figure 5 presents the differences of each change scenario.

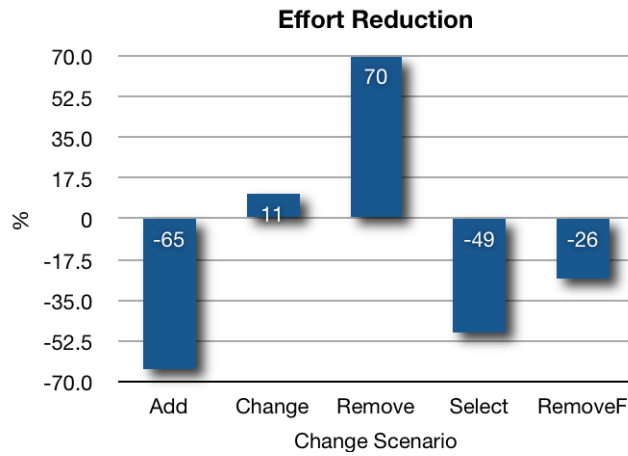


Fig. 5. Effort Reduction

Observations The following observations can be made with a statistical significance of 95%:

- Adding tracing to a function takes less time with *WeaveC* than without.
- Removing tracing from a function takes more time with *WeaveC* than without.
- Selective tracing the parameters of a function, takes less time with *WeaveC* than without.

Correlations We found the following correlations between the subjects and their performance without and with *WeaveC*:

- There was a strong, positive correlation between the difference in effort while changing a function and the number of years in software development [correlation factor=.624, number of measurements=11, significance \leq .05]. In other words: subjects with more years experience in software development, gained more when using *WeaveC*.

4.4 Errors

Descriptives Table 8 presents several details for all scenarios, namely; the number of scenarios, the minimum values, the maximum value, the mean and the standard deviation.

Significance We calculated the significance through the use of a standard paired sample test with a confidence interval of 95%, see table 9.

Figure 6 presents the error reduction of each change scenario.

Observations The following observations can be made with a statistical significance of 95%:

- Changing the parameters of a function manually introduces more errors than with *WeaveC*.

Scenario	N	Min	Max	Mean	Std. Dev.
Adding Tracing w/o <i>WeaveC</i>	11	0	3	0.82	1.25
Changing Parameters w/o <i>WeaveC</i>	12	0	4	2.58	1.78
Removing Tracing w/o <i>WeaveC</i>	9	0	0	.00	.00
Selectively Tracing w/o <i>WeaveC</i>	7	0	4	1.57	1.62
Removing Function w/o <i>WeaveC</i>	15	0	0	.00	.000
Adding Tracing with <i>WeaveC</i>	13	0	0	.00	.000
Changing Parameters with <i>WeaveC</i>	15	0	0	.00	.000
Removing Tracing with <i>WeaveC</i>	10	0	0	.00	.000
Selectively Tracing with <i>WeaveC</i>	13	0	3	0.92	1.44
Removing Function with <i>WeaveC</i>	15	0	0	.00	.000

Table 8. Descriptives of Errors

Scenario	N	Mean	Std. Dev.	Sig. (2-tailed)
Adding Tracing	8	.038	0.74	0.20
Changing Parameters	11	2.50	1.81	0.00
Removing Tracing	6	0	0	-
Selectively Tracing	6	0.50	2.60	0.66
Removing Function	13	0	0	-

Table 9. Significance of Errors

Correlations We found the following correlations between the subjects and the difference between without and with *WeaveC*.

- There was a strong, positive correlation between the difference in errors while changing the parameters of a function and the number of years in software development [correlation factor=-.723, number of measurements=11, significance \leq .05]. In other words: the subjects gained more, with *WeaveC*, if they were more experienced in software development.
- There was a strong, positive correlation between the difference in errors while changing the parameters of a function and the age of subjects [correlation factor=-.624, number of measurements=11, significance \leq .05]. In other words: the subjects gained more, with *WeaveC*, if they were older.
- There was a strong, positive correlation between the difference in errors while selectively tracing a parameter of a function and the number of years in software development [correlation factor=-.879, number of measurements=7, significance \leq .01]. In other words: the subjects gained more, with *WeaveC*, if they were more experienced in software development.
- There was a strong, positive correlation between the difference in errors while selectively tracing a parameter of a function and the age of the subjects [correlation factor=-.879, number of measurements=7, significance \leq .05]. In other words: the subjects gained more, with *WeaveC*, if they were older.
- There was a strong, positive correlation between the difference in errors while selectively tracing a parameter of a function and the experience with C [correlation factor=-.789, number of measurements=7, significance \leq .05]. In other words: the subjects gained more, with *WeaveC*, if they had more experience in C.

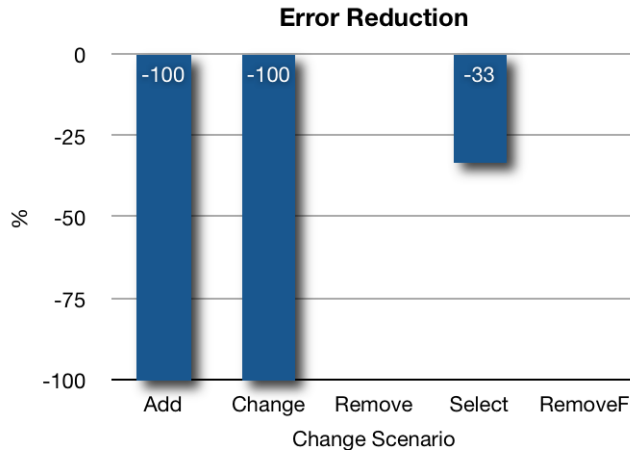


Fig. 6. Error Reduction

4.5 Verification of the Hypotheses

We now verify the hypotheses that were stated in section 3.6.

We can reject the H_0^{time} for scenarios Adding Tracing and Selectively Tracing. We can thus accept alternative hypothesis H_a^{time} for these scenarios.

We cannot reject H_0^{time} for scenarios Changing Parameters and Remove Function. Also, we have to accept H_0^{time} for scenario Remove Tracing. Therefore, we cannot accept alternative hypothesis H_a^{time} for these scenarios.

We can reject H_0^{error} for scenario Changing Parameters. We can thus accept alternative hypothesis H_a^{error} for this scenario.

We cannot reject H_0^{error} for scenarios Adding Tracing, Remove Tracing, Selectively Tracing and Remove Function. Therefore, we cannot accept the alternative hypothesis H_a^{error} for these scenarios.

Concluding, the results from the experiment showed with statistical significance of 95% that adding tracing to a function and selectively tracing (only one parameter) takes less time to implement with the use of *WeaveC*. The results also showed that changing the parameters of a function reduces the severity of errors substantially with the use of *WeaveC*.

5 Validation

There are many possible threats to the execution and results of an experiment. In this section we present some these threats which could have had the most impacts, discuss our counter measures and the possible impact of these threats. Although, we use an informal question answer style, all questions can be mapped to the categorization proposed in [12]. These categories are: construct, internal and external validity and reliability. We used the guidelines proposed in [9] to cover all validity threats.

You combined the results from both groups, is this valid? Table 10 shows the descriptives of the subjects in both groups.

Group	Characteristic	N	Min	Max	Mean	Std. Dev
1	Age	12	26	45	35.42	5.66
	Education	12	2	3	2.50	0.52
	Years SW	11	1.0	20.0	9.00	5.80
	Years ASML	12	0	7.0	4.21	2.71
	Clevel	12	1	5	3.50	1.09
	THXA	12	0	1	0.58	0.52
2	Age	5	26	43	33.60	6.35
	Education	5	2	3	2.20	0.45
	Years SW	5	1.0	17.0	9.60	5.86
	Years ASML	5	0	5.0	1.46	2.03
	Clevel	5	4	5	4.20	0.45
	THXA	5	0	1	0.60	0.55

Table 10. Groups

From table 10 we can observe the following:

- Subjects in group one had, on average, worked longer at ASML.
- Subjects in group two had, on average more experienced in C.

We do not feel that this impacts the results of the experiment a lot, especially since we did not find any significant correlation between this variable and the performance of the subjects. Also, intuitively these two differences may compensate each other. As one can see there were more subjects in group 1 than in group 2, there are two reasons for this. Firstly, three subjects we excluded were in group 2. Secondly, two subjects accidentally received the instructions of group 1 in the without *WeaveC* session. We chose to move them to group 1 once we discovered this.

Could the separation over two sessions have influenced your results?

Similar to the previous question, we also verified whether there were no major differences between the subjects participating in the morning session and in the afternoon session. Due to space limitations we don't provide a similar table, but only present our observations.

- Subjects in the morning session had, on average, worked longer at ASML.
- Subjects in the morning session had, on average, worked longer in Software Engineering.

We believe that this does not impact the results of the experiment a lot, as we did not find any significant correlation between this time of day variable and the performance of the subjects. More importantly, we take the composed results of the both sessions, this eliminates this threat. The subject executed the same treatments and we did not measure the difference between session.

Were the subjects representative?

We had no control over the selection of the subjects. Therefore, we could not make an representative selection from the population. Also, we do not have any statistics about the characteristics of the

population, therefore we cannot extrapolate our results to a specific population. However, we believe that the characteristics of the subjects are not ASML or industry specific and as such can serve as an indicator for AOP industry wide. But we cannot support this claim with statistical evidence.

Why are not all scenarios significant? There may be several causes why we did not get significance for some scenarios. One of these causes can be that the sample size is too small. Another can be that the expected benefits may not exist in those scenarios, although most scenarios indicate that there are benefits.

Were the without and with WeaveC treatments equivalent?

1. **Adding Tracing:** In one scenario the subjects had to trace four input parameters and one output parameter. In the other scenario the subjects had to trace two input parameters and two output parameters. There is a difference in complexity, we also saw indications in the results that this might have impacted the results. Group 1, which first executed the hard scenario without *WeaveC* and then the simple scenario with *WeaveC*, benefited more from *WeaveC* than group 2. Group 2 executed the simple scenario first and then the hard scenario, and thus benefited less from *WeaveC*. However, we are unable to establish the precise impact of this difference.
2. **Changing parameters of a function:** In one scenario, the subjects had to change one parameter from an input int to an input and output int. The other scenario had a similar change but this required changing two parameters from input to input and output. There is a small complexity difference here that may have impacted the experiment. However, we saw no evidence of this in the results.
3. **Removing Tracing:** Removing trace statements from the functions in the scenarios has the same complexity in both scenarios. Therefore, we do not expect an impact on the results.
4. **Selectively Tracing:** Only tracing one specific parameter is equally complex in both scenarios. Therefore, we do not expect an impact on the results.
5. **Removing Function:** Removing a function is equally complex in both scenarios.

Did you exclude any subjects? We excluded three subjects from the results. The check-in and check-out time stamps for these subjects were not valid. This was either caused by a bug in the monitor tooling, or because the subjects did not check-in the files.

Did you exclude any data points? We removed 8 individual scenarios from the session without *WeaveC*, and 6 individual scenarios from the session with *WeaveC*. These were all statistical outliers related to the development effort. These statistical outliers were determined using SPSS [11]. We removed the individual scenarios, afterwards we verified that there were no more statistical outliers related to the errors. This was indeed the case.

Was there an impact of the tooling on the results? We had to ensure that the build performance did not influence the experiment results too much, i.e.

approaching the scenario times or an order of magnitude slower than a regular build. We used the command logger to determine the build times. The results are presented in table 11.

#C builds	Average C build time	#AOP builds	Average AOP build time	Δ
65	24.71	88	51.82	210%

Table 11. Build times

From the table we can observe that the average build time is roughly doubled. However, as this is a realistic delay which we expect will only decrease as the tooling is improved, we did include the build times in our experiment results.

Did you address the initial startup effects of the subjects? To prevent an initial startup time from influencing our results, we added a scenario (*Remove Function*) as the first scenario. Any initial startup effects should be captured by this first scenario.

Was there a learning effect between scenarios? We used clustered randomness to prevent learning effects between scenarios. This ensured that even if there was insufficient time for the participants to finish all scenarios, we would have results from the two most important scenarios. The order in which the scenarios were executed in each session was randomly assigned.

Was there a learning effect between sessions? We introduced scenario *Remove Function* not only to determine the initial startup effects, but also to determine whether there was a learning effect between the without *WeaveC* session and with *WeaveC* session. The actions required for executing this scenario are the same in both sessions. However, we are unable to state whether there was a learning curve as there are no significant results for this scenario. Also there is no clear separation between the times of the initial startup effects and the learning curve.

How did you ensure that the users were motivated? As the experiment was part of a voluntary training, we can assume that most subjects were motivated to learn about *WeaveC*. To further ensure the motivation, we promised the best 10 subjects a small gadget (around 10 euro). We explained to the subjects that the definition of “the best” was a function of both time and errors. This should have prevented the users from rushing the scenarios.

How did you ensure that all subjects had a similar notion of tracing? At the start of the course we presented the current tracing idiom to all users. The subjects were all present during the same presentation, to remove the possible influence of different instructors. Similarly, we presented the subjects with the concepts of AOP and the new way of working, before the second session. Again the subjects attended the same presentation, presented by the same instructor.

How did you ensure that the users were familiar with the tooling and process? As part of the training, we included a tool experience session before the session with *WeaveC*. In this session, the subjects had to go through all the steps of the new way of working and use all features of the weaver. Therefore, all subjects were familiar with the tooling and process before they started with the *WeaveC* session.

Did you satisfy the requirements for the T paired sample test? The T paired sample test requires normally distributed results. We verified the normality of effort and severity of errors using the Kolmogorov-Smirnov statistics and histograms produced by SPSS[11]. Almost all scenarios were normally distributed. There were five which were reasonably normally distributed, probably as a result of the low number of measurements and the variation between these measurements. These scenarios related to the effort were: Remove Tracing w/o *WeaveC* and Selectively Tracing with *WeaveC*. Regarding the severity of errors these scenarios, were: Add Tracing w/o *WeaveC*, Change Function Parameter w/o *WeaveC* and Selectively Tracing with *WeaveC*.

Is the raw data available? Due to confidentiality reasons we cannot publish the raw results of the experiment. We can be contacted to provide these results based on a nondisclosure agreement or a similar construction.

6 Survey

A survey was conducted three months after the training and experiment. The purpose of this survey was to determine the current usage and acceptance of *WeaveC*, as well as to determine any issues users faced.

Out of 50-60 current users of *WeaveC*, 26 users responded. In this group were 8 persons who also attended the training and experiment. The survey consisted of several questions about the current acceptance of the tooling and usage of *WeaveC* within ASML. The users were also questioned about the usage, any issues and benefits of *WeaveC*.

The respondents stated that they would like to have more training or documentation about the usage of annotations. They also proposed several new annotations. This strengthens the assumption that the negative development effort while removing tracing, is caused by the newly introduced annotations. Similar, for changing parameters where the subjects also still introduced errors. Both these scenarios use annotations.

The users also stated that they would like to see more (complex) aspects like *Profiling* and especially *Error Propagation*. This implies that the users have confidence in *WeaveC* and want to use it for more aspects than just *Tracing*. The following benefits of *WeaveC* were expressed by the respondents:

- Better handling of crosscutting concerns.
- Lead time reduction.
- Effort reduction.
- Less boring work.
- Better quality: less errors and cleaner code.

The results from the survey confirm our results from the experiment. Overall the users experience the benefits of *WeaveC*. They also mention the need for good and more elaborate documentation and training for those elements of *WeaveC* which are new to them, especially annotations.

7 Related Work

As mentioned in the introduction there are numerous papers about the importance of industrial controlled experiments and success stories, e.g.[2], [3] and [4]. However, actual papers about controlled experiments and success stories are scarce.

In [13], Sjoberg et. al. present a survey of controlled experiments in software engineering. The authors calculated that only 1.9% of journal and conference articles in a representative set of journals and conferences reported on controlled experiments in software engineering. Only 0.2% of the papers reported to use professionals for the experiments. Similarly, 0.3% reported on experiments which measured the effects of changing the code. This indicates that empirical validation of software engineering methodologies which improve development effort is rare. In the area of AOP this is certainly.

There are some success stories about the industrial adoption of AOP. In [14], Bodkin et. al. report an experience where the authors applied aspects to provide feedback on user behavior, system errors, and to provide a robust solution for a widely deployed diagnostic technology for DaimlerChrysler. Aspects are used as a reflective means to gather information about the system. Although the authors present a discussion about benefits and challenges, they do not provide a quantitative assessment of the benefits of AOP.

In [1], Colyer and Clement discuss large scale AOSD for middleware. The authors report on a case-study they conducted at IBM. They present of set of challenges they faced while executing the case-study. One of the aspects tackled in the case study is also *Tracing*. This supports our statement that even a “simple” concern like *Tracing* can be an excellent driver to adopt AOP. The conclusion of the authors is that AOP can be used successfully on a large scale. They do not provide a quantitative assessment of the benefits of AOP.

In [15], Mendhekar, Kizales and Lamping presented the results of a case-study that compared the implementations of an image processing system. The authors compared the performance of a naïve OO implementation, an optimized OO implementation and an AOP implementation. The authors showed that the performance of the AOP solution was comparable to the performance of the optimized OO version. The performance of the naïve OO implementation was much slower. The AOP solution required 88% less lines of code (including the weaver). They only considered the performance and not the development time and effort in a controlled manner.

Walker, Murphy and Baniassad [16,17] present an initial assessment of AOP. Here the authors present several experiments. One experiment to determine whether users of AOP were able to more quickly and easily find and correct faults in a multi-threaded program. The results of this experiment showed that

the AspectJ users were indeed faster for one scenario, while for two other scenarios the difference was smaller compared to the regular Java developers. The intention of the second experiment was to investigate whether the separation of concerns provided in AOP enhances the ability to change the functionality of a multi-threaded, distributed program. The results of this experiment showed that the users of AOP(Cool, Ridl and JCore) were slower than the user which manually changed the program using Emerald. These results seem to contradict our findings in this paper. There are several differences between our experiment and the one described by the authors. The main difference is that the authors conducted semicontrolled empirical studies rather than the “statistically valid experiments” [17] we conducted. Other differences are the smaller set of subjects in the studies and that the subjects are not professionals, but students and professors.

8 Conclusions

The experiment we report on in this paper has adopted the *Tracing* aspect, explained in section 2. Tracing is sometimes dismissed as a trivial aspect, however, we showed that this is certainly not the case in the actual realization of Tracing at ASML. Additionally, this “simple” aspect can serve as an excellent driver to adopt AOP, as also mentioned by Colyer et.al. [1]. Aspects can be used to address a wide range of crosscutting concerns, but the benefits of adopting AOP are not as clear for heterogeneous or more complex concerns, as with homogeneous concerns like *Tracing* and *Profiling*. As such these more “simple” aspects should not be overlooked, but rather be embraced as effective examples to introduce AOP in industry.

One of the contributions of this paper is the design of a controlled experiment that can be used to quantify the benefits of AOP in other organizations. The design can easily be adopted for other aspects, other base code, and other scenarios. We believe that the proposed integration with an introductory course is an elegant solution to reduce the reluctance of a development organization for spending time on an experiment.

We conducted the experiment with 20 ASML developers. The developers had to execute five simple tracing-related change scenarios twice. First, the developers had to implement the scenarios manually. After this the subjects had to implement the scenarios using *WeaveC*. We have tried to reduce validity threats through careful design of the experiment (see section 3), and offer a detailed discussion of validity threats in section 5.

The results from this experiment, presented in section 4, showed that, overall, the subjects were able to execute the scenarios 6% faster using the AOP solution. There were scenarios where the subjects were slower with AOP. Since the actual amount of work was much less, we *believe* that this is caused by the fact that the subjects were new to the tooling, especially the required annotations. More prominent however, was the reduction in errors when using the AOP solution by 77%. We can safely conclude that in the experiment case, a substantial reduction of errors was achieved with even slightly less effort.

The statistical results of the experiment are limited, as —probably due to the limited number of participants— not all executed treatments give results that are statistically significant (at 95% reproducibility). For two common change scenarios there is a statistically significant benefit in terms of development effort w.r.t. *Tracing*. There is a statistically significant severity error reduction for one scenario.

The profile of the subjects in terms of education and software development experience (see section 4.1) seems representative for ASML and (medium- to large-sized) software development organizations, but we had no comparable profile information about these population to substantiate such a conclusion.

A survey was conducted amongst 26 users of *WeaveC*. The respondents expressed the need for more complex aspects, like errors handling. This indicates that the respondents were confident in *WeaveC*. Secondly, the respondents expressed the need for more explanation about annotations, as this was a new concept. This strengthens our belief that there is still some gain in development effort, which we were unable to measure. In conclusion, the survey confirms that the users do “experience” the benefits of AOP, in line with the —careful— conclusions from the experiments.

The current set of subjects is limited. We hope to conduct more experiments in the future to validate our results and achieve —presumably— statistically more significant results.

9 Acknowledgments

We would like to thank Vincent Buskens and Richard Zijdemans from the Department of Sociology at the University of Utrecht. Their expertise in the area of experiments has had a large impact on the design of the experiment. They also helped with analyzing the results from this experiment. We also would like to thank the *WeaveC* team at ASML, and especially Steven De Brouwer, Istvan Nagy and Remco Van Engelen. Their support and expertise ensured an optimal environment for the experiment. Without the weaver developed by the *WeaveC* team at ASML, non of this work would have been possible. They also conducted the survey as mentioned in section 6.

This work has been partially carried out as part of the Ideals project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter-Novem TS program (grant TSIT3003).. This work is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe).

References

1. Colyer, A., Clement, A.: Large-scale AOSD for middleware. In Lieberherr, K., ed.: Proc. 3rd Int’ Conf. on Aspect-Oriented Software Development (AOSD-2004), ACM Press (March 2004) 56–65
2. Daniel Wiese, Uwe Hohenstein, R.M.: How to convince industry of aop. In: Proceedings of Industry Track of the 6th Conf. on Aspect-Oriented Software Development. (2007)

3. Garcia, A., Soares, S., Eaddy, M., Bartsch, M., Aksit, M.: Informal workshop report on assessment of aspect-oriented technologies. Technical report, AOSD (2007)
4. Bodkin, R., Colyer, A., Memmert, J., Schmidmeier, A., eds.: AOSD Workshop on Commercialization of AOSD Technology. In Bodkin, R., Colyer, A., Memmert, J., Schmidmeier, A., eds.: AOSD Workshop on Commercialization of AOSD Technology. (March 2003)
5. Laddad, R.: AOP@Work: Myths about AOP (2006) <http://www-128.ibm.com/developerworks/java/library/j-aopwork15>.
6. Durr, P., Gulesir, G., Bergmans, L., Aksit, M., van Engelen, R.: Applying aop in an industrial context. In: Workshop on Best Practices in Applying Aspect-Oriented Software Development. (Mar 2006)
7. Nagy, I., van Engelen, R., van der Ploeg, D.: An overview of mirjam and weavec. In van Engelen, R., Voeten, J., eds.: Ideals: evolvability of software-intensive high-tech systems. Embedded Systems Institute, Eindhoven (2007)
8. Wohlin, C., Runeson, P., Horst, M., Ohlsson, M., Regnell, B., Wesslen, A.: Experimentation In Software Engineering. Kluwer Academic Publishers (2000)
9. Kitchenham, B., Pfleeger, S., Pickard, M., Jones, P., Rosenberg, D.H.J.E.J.: Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* **28**(8) (March 2002) 721–734
10. McCabe, T.J.: A complexity measure. *IEEE Transactions on Software Engineering* **2**(2) (1976) 308–320
11. SPSS: SPSS VERSION 13. FOR WINDOWS <http://www.spss.com/spss/>.
12. Cook, T.D., Campbell, D.T.: *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin Company (1979)
13. Sjoberg, D., Hanney, J., Hansen, O., Kampenes, V.B., Karahasanovic, A., Liborg, N., Rekdal, A.: A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering* **31**(9) (September 2005) 733–753
14. Bodkin, R., Furlong, J.: Gathering feedback on user behaviour using aspectj. In: Proceedings of the Industrial Track of the fifth International Conference on Aspect-Oriented Software Development. (2006)
15. Mendhekar, A., Kiczales, G., Lamping, J.: RG: A case-study for aspect-oriented programming. Technical Report SPL-97-009, Palo Alto Research Center (1997)
16. Walker, R.J., Baniassad, E.L.A., Murphy, G.C.: An initial assessment of aspect-oriented programming. In: Proc. 21st Int'l Conf. Software Engineering (ICSE '99). (1999) 120–130
17. Murphy, G.C., Walker, R.J., Baniassad, E.L.A.: Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. *IEEE Transactions on Software Engineering* **25**(4) (1999) 438–455