Does Aspect-Oriented Programming Increase the Development Speed for Crosscutting Code? An Empirical Study

Stefan Hanenberg, Sebastian Kleinschmager, Manuel Josupeit-Walter
University of Duisburg-Essen
45117 Essen, Germany
stefan.hanenberg@icb.uni-due.de
sebastian.kleinschmager@stud.uni-due.de
manuel.josupeit-walter@stud.uni-duisburg-essen.de

Abstract

Aspect-oriented software development is an approach which addresses the construction of software artifacts that traditional software engineering constructs fail to modularize: the so-called crosscutting concerns. However, although aspect-orientation claims to permit a better modularization of crosscutting concerns, it is still not clear whether the development time for such crosscutting concerns is increased or decreased by the application of aspect-oriented techniques. This paper addresses this issue by an experiment which compares the development times of crosscutting concerns using traditional composition techniques and aspect-oriented composition techniques using the object-oriented programming language Java and the aspect-oriented programming language AspectJ. In that way, the experiment reveals opportunities and risks caused by aspect-oriented programming techniques in comparison to object-oriented ones.

1. Introduction

A typical argument for aspect-oriented software development (AOSD, [4]) is that aspects permit a better modularization of code that crosscuts other modules (see [10]): code which is caused by so-called crosscutting concerns. Such arguments focus on the readability and maintainability of software constructed from the aspect-oriented approach (see further [22]).

While there are studies available that seem to support that aspect-orientation possibly improves maintainability (see for example [5]), it is still not clear what costs are caused by aspect-oriented programming techniques: it is not clear whether the application of

aspect-oriented techniques increases the development time or whether developers already benefit from aspect-orientation by a reduced development time.

However, knowledge about benefits and possible risks of aspect-orientation with respect to development time is necessary not only for the academic world but also for the software industry. From a software manager's point of view knowledge about opportunities and risks is necessary in order to decide whether aspect-orientation should be applied either in general or to solve a given crosscutting problem.

A first step into that direction is to concentrate only on those pure programming tasks aspect-orientation was built up for: the specification of code fragments that would crosscut other modules if specified using traditional composition techniques. In case aspect-orientation turns out to reduce the pure coding time it strengthens the argument to use aspect-orientation (if it is assumed that the maintainability costs are also reduced). In case it turns out to increase the coding time, it needs to be considered whether the (expected) benefit in the maintainability phase is larger than the costs caused by additional coding time. However, it is unclear how the application of aspect-oriented constructs influences the development time on the programming level.

This paper introduces an experiment conducted on 20 subjects that studies the influence of aspect-oriented programming language constructs on the development time of crosscutting concerns which was originally the main focus of aspect-oriented programming (see [10]).

Section 2 motivates such a study by discussing possible pros and cons of aspect-oriented programming constructs. Section 3 introduces the experiment by describing the experimental design and its execution. Section 4 analyses the experiment by providing

descriptive data and then performing significance tests. After discussing related work in section 5, section 6 discusses and concludes the paper.

2. Motivation

One example for aspect-oriented techniques that is quite often cited in literature is logging (c.f. [4] among many others): invocations to the logger need to appear in a large number of modules. Furthermore, the pieces of code to be specified need to be adapted within each module in order to pass for example the method name or the actual parameters. Without using techniques that permit to program the logging behavior, hand coding such a feature possibly requires a large amount of development time and causes in that way additional development costs. From that point of view it seems clear that aspect-oriented programming techniques significantly decrease the development time.

From a different perspective, one can argue that aspect-oriented programming brings additional abstractions and therefore additional complexity into the development of software (cf. e.g. [20]). Such additional complexity reduces the development speed in such a way that the possible advantage of the technology turns out to be rather a burden for the developer and rather increases the development time.

Although both arguments seem to be reasonable they contradict each other. From the scientific point of view this situation is not satisfactory, because both arguments rely on speculations. According to the appeal formulated in [21] empirical methods (cf. [8, 19, 24]) are an approach to address this problem. This would permit to strengthen (or weaken) arguments based on observed data. This is the chosen approach of this paper.

The here described experiment relies on crosscutting code which does not require any dynamic condition fulfilled at runtime in order to determine whether it should be executed. Spoken in aspect-oriented terms, we study only aspects with underlying static join points (cf. e.g. [7]). Although the main focus of research in the area of aspect-oriented programming language constructs is in the area of dynamic constructs (cf. e.g. [6, 15]) we reduce our view on aspect-orientation to static elements because static join points can be unambiguously determined, i.e. it can be clearly stated where and how they appear in the code. Furthermore, by reducing the question to static join points we reduce the number of different solutions provided by the subjects in the experiment.

The experiment's intention is to study the impact of aspect-oriented techniques on the development time for

crosscutting code. Further issues such as readability or maintainability of the code are not considered.

3. Experiment Design

3.1. Research Question

Our intention is to identify whether (and when) the application of aspect-oriented programming turns out to have a positive impact on the development time. We assume that the application of aspect-oriented techniques turns out to be time saving for a large number of code clones caused by a crosscutting concern. For a rather small number of code clones we expect that the development speed decreases. Hence, we assume that developers using aspect-oriented techniques require more time for solving a programming tasks if the aspect is applied to rather few places in the code.

The intention of the experiment is to approximate the number of code places at which the application of aspect-oriented constructs turns out to be useful. Therefore we defined an experiment where subjects performed the same programming tasks using an object-oriented and an aspect-oriented programming language. We used Java as a representative of an object-oriented (OO) language and AspectJ (see [12]) as a representative of an aspect-oriented (AO) language.

3.2. Programming Tasks

Within the experiment subjects were asked to write crosscutting code into a target application. As a target application, we used a small game consisting of 9 classes within 3 packages with 110 methods, 8 constructors, and 37 instance variables written in pure Java (version 1.6). Each class was specified in its own file. The game consists of a small graphical user interface with an underlying model-view-controller architecture.

Using this application, nine tasks needed to be done, each one in pure Java (version 1.6) as well as AspectJ (version 1.6.1) whereby the order of the programming language was randomly chosen. We have chosen programming tasks according to the following criteria:

- The programming tasks should be in the domain that AspectJ is designed for. Hence, the programming tasks should not represent crosscutting whose modularization cannot be achieved by AspectJ.

- The tasks should differ with respect to the expected time required to solve it.
- The tasks should not force developers to enumerate all corresponding join point shadows where the crosscutting code occurs.
- The aspect-oriented solutions should not be trivial, i.e. non-trivial constructs such as reflective access on the current join point via `thisJoinPoint`, `around/proceed` blocks as well as context exposure (see further [12]) should be applied in order to solve the tasks.

In the following sections we use the term code target to describe a position in the code where adaptations are required by the object-oriented developer.

```
class C ... {
...
public R m(int i,A a) {
    Logger.log("C", "m", "R",
        new Object[] {i, a},
        new String[] {"int", "A"});
    ...method body...
}
```

Figure 1. Example log-invocation in Java

3.2.1. First Task: Logging. The first task was to add a logging-feature to the application, where each method should be logged. Thereto, a corresponding logger-interface was provided that expects from each method its return type, the name of the class where the method was declared in, an array of the formal parameter names and an array of its actual parameters. An example log statement for a method `m` in a class `C` with parameter types `int` and `A` is shown in Figure 1. Altogether, such a line needed to be added to 110 code targets (i.e. methods).

```
pointcut logging():
    execution(* game.*.*(..)) ||
    execution(* filesystem.*.*(..)) ||
    execution(* gui.*.*(..));
before(): logging() {
    MethodSignature m = (MethodSignature)
        thisJoinPoint.getSignature();
    Logger.log(
        m.getReturnType().getSimpleName(),
        m.getName(),
        m.getDeclaringType().getSimpleName(),
        thisJoinPoint.getArgs(),
        m.getParameterTypes()
    );
}
```

Figure 2. Example log-invocation in AspectJ

For the aspect-oriented solution, the aspect definition, consisting of the keyword `aspect`, an aspect name and the corresponding brackets was given to the subject. A good AspectJ (and short) solution for this task is to write a pointcut that refers to the target classes via their package descriptions and a corresponding advice that reads from `thisJoinPoint` the method signature and the actual parameters. An example for such a piece of code is shown in Figure 2.

3.2.2. Second Task: Nullpointer checks. The second task was to add nullpointer-checks to all non-primitive parameters of all methods in the application. In case one of the non-primitive parameters was null, an `InvalidGameStateException` should be thrown (the exception's class definition was already part of the delivered application).

```
class C ... {
...
public R m(int i,A a, B b) {
    if(a==null || b==null)
        throw new InvalidGameStateException();
    ...method body...
}
...
}
```

Figure 3. Example nullpointer-check in Java

Figure 3 shows an example for the object-oriented solution with a corresponding null-pointer check that needed to be defined by the developer. The example is defined for a method `m` in class `C` with parameter types `int`, `A` and `B` (and the corresponding parameter names `i`, `a` and `b`) Altogether, 36 code targets needed to be adapted.

A possible AspectJ solution is shown in Figure 4.

```
pointcut nullpointerCheck():
    execution(* game.*.*(..)) ||
    execution(* filesystem.*.*(..)) ||
    execution(* gui.*.*(..));
before():nullpointerCheck () {
    MethodSignature m = (MethodSignature)
        thisJoinPoint.getSignature();
    for(int i=0;i<m.getParameterTypes().length; i++) {
        if(!m.getParameterTypes()[i].isPrimitive
            && m.getArgs()[i]==null)
            throw new InvalidGameStateException();
    }
}
```

Figure 4. Example nullpointer-check in AspectJ

3.2.3. Third Task: Synchronization. The third task was to add synchronization statements to a set of methods. The task explicitly demanded to use the Java synchronize-block (instead of using the synchronize keyword in the method header) where the synchronization is achieved on the actual object. Altogether, 52 code targets needed to be adapted.

For the aspect-oriented solution it is necessary to use a proceed-statement within a synchronization block. In order to synchronize the actual executing object, this object needs to be exposed to the advice (see Figure 5).

```
pointcut syncs(Object o):
(
    execution(* GameObject.*(..)) ||
    execution(* Player.*(..)) ||
    execution(* Trap.*(..)) ||
    execution(* GameLevel.*(..));
) && this(o)

Object around(Object o): syncs(o) {
    synchronized(o) {
        return proceed(o);
    }
}
```

Figure 5. Synchronization example in AspectJ

3.2.4. Fourth Task: Player Check. In the fourth task developers needed to check whether a passed parameter corresponds to an instance variable called player of the receiving object whenever certain methods are invoked. Altogether, 4 code targets needed to be adapted.

```
pointcut update(GameManager g, Object o) :
    execution(* GameManager.update*(..)) &&
    this(g) && args(o);

before(GameManager g, Object o) : update(g, o) {
    if(g.player != o)
        throw new InvalidGameStateException();
}
```

Figure 6. Player check in AspectJ

In AspectJ this can be achieved by exposing the object receiving the method and the corresponding parameter to an advice which checks whether the second exposed element corresponds to the player instance variable of the receiving object (see Figure 6).

3.2.5. Fifth Task: Notify Observers. The fifth task was the notify observers task, where the participants had to insert a call to the notifyObservers method whenever a certain instance variable changes. This had to happen every time an instance variable was set, not just at the end of the method where the setting had

happened. Altogether, 19 code targets were subject of such editing.

For the object-oriented solution, a notifyObservers method call needs to be added to all places where a state change happens (see Figure 7). A corresponding aspect-oriented solution is visualized in Figure 8 where developers need to make use of the set-pointcut for detecting the state changes.

```
public void setValues(int value1, int value2) {
    x = value1;
    notifyObservers();
    y = value2;
    notifyObservers();
}
```

Figure 7. NotifyObservers call in Java

3.2.6. Sixth Task: Observer Null Check. In the sixth task developers were asked to check right before sending a notifyObservers message whether the observer field is null. In that case an exception should be thrown. In the object-oriented task it was necessary to change 15 code targets.

```
pointcut stateChange(GameObject g) :
    set(* GameObject+.*) &&
    this(g);
after(GameObject g) : stateChange (g) {
    g.notifyObservers();
}
```

Figure 8. NotifyObservers call in AspectJ

```
pointcut ObserversCheck(GameObject o) :
    call (void GameObject+.notifyObservers(..)) &&
    target(o);

before(GameObject o) : ObserversCheck (o) {
    if(o.observers == null)
        throw new InvalidGameStateException();
}
```

Figure 9. Observer Null task in AspectJ

For the aspect-oriented solution, a change in the pointcut and a slight change in the advice is necessary (see Figure 9).

```
pointcut refreshes() :
    execution(* LabyrinthFrame.refresh*(..));
before () : refreshes() {
    for (int i = 0; i < thisJoinPoint.getArgs().
        length; i++) {
        if (((Integer)thisJoinPoint.getArgs()[i]).
            intValue() < -1)
            throw new InvalidGameStateException();
    }
}
```

Figure 10. Refresh solution in AspectJ

3.2.7. Seventh Task: Refresh Task. The seventh task was to check for all methods of a certain class whose names start with the prefix refresh whether the delivered integer parameters are smaller than -1. In such a case, an exception needed to be thrown. Altogether, 8 code targets needed to be adapted.

While the object-oriented solution is quite straightforward, a typical aspect-oriented solution could look like as shown in Figure 10: the advice reflects on the arguments via thisJoinPoints and checks within a loop whether the condition holds.

3.2.8. Eighth Task: Label Value Check. The eighth task required developers to check right after setting the value of a label object whether this value is valid. Thereto, a method isValidLabelString was provided which needed to be invoked by the developers with a corresponding string as a parameter. The adaptation of six code targets was required. A possible AspectJ solution can be found in Figure 11.

```
pointcut afterSetText(JLabel label, Frame f):
  call(* JLabel.setText(..) && this(f) &&
  target(label);

after(JLabel j, Frame f) :
  afterSetText(j, f) {
    if(!f.isValidLabelString(j.getText()))
      throw new InvalidGameStateException();
  }
```

Figure 11. Label Value Check in AspectJ

3.2.9. Ninth Task: Current Level Check. The ninth task, again, was to check whenever a certain instance variable changes (a level-object) whether the new assigned object is valid. Thereto, a method checkLevelConstraints was provided which requires the newly assigned object and a level-object representing the current level as parameters. This task required the change of 7 code targets in the code. A possible AspectJ solution can be found in Figure 12.

```
pointcut setLevel(GameManager m, GameLevel l) :
  set(public GameLevel GameManager.currentlevel)
  && this(m);

before(GameManager m, GameLevel l) :
  setLevel(m, l){
    if(m.currentlevel != null)
      if(!m.checkLevelConstraints(
        l, m.currentlevel))
        throw new InvalidGameStateException();
  }
```

Figure 12. Current Level Check in AspectJ

3.3. Teaching

Each subject was taught a short introduction into AspectJ which took about 1.5 hours for which a corresponding tutorial was prepared. This tutorial (including exercises) was not meant to be an exhaustive training in AspectJ: only those language constructs that were required in the experiment were introduced. Constructs such as declare precedence statements (see further [12]), handler pointcuts or further advanced constructs in AspectJ were not trained.

More precisely, this means that the pointcut language constructs call, execution, this, target, args, the combination of pointcuts, context exposure, thisJoinPoint and proceed (within around advice) were taught. Dynamic constructs such as cflow, cflowbelow, etc. (see [12]) were not taught.

Java was not explicitly taught, since all subjects passed fundamental Java programming courses.

3.4. Measurement

In order to measure the times required to fulfill a task the following measurement was performed: all actions performed on the code base were logged to a database, i.e. an underlying database represented all code changes performed by the developer. After the experiment, the time was measured between the moment when developers first performed a change on the code base for a given task and the moment when all test cases for that task were fulfilled. In case the developer performed a change afterwards, this was not considered in the measurement.

In order to perform this measurement, we extracted every 30 seconds snapshots from the logged data representing the code at the corresponding point in time. We used only those code snippets that could be compiled. Hence, if at a certain point in time no compilable code for a given class or aspect was available, we used the last version of the corresponding class or aspect that was compilable.

All subjects were required to complete all tasks.

3.5. Threats to validity

A slight problem with respect to the internal validity is that the experiment was performed in different sessions. Although the intention was to give the absolute same tutorial, there were practically slight differences in time (due to a different amount of questions asked by the subjects).

There was no difference in the ordering of the tasks. I.e. a possible learning curve of the participants also

potentially threatens the internal validity of the experiment.

The teaching issue is also a threat to the external validity. Although only basic language features of AspectJ were being taught, possibly a longer teaching time is required in order to become a good AspectJ developer. From the industry's point of view the external validity might still hold for projects that decide for the first time whether or not aspect-oriented techniques should be applied. However, it can be assumed that the development speed using aspect-oriented constructs increases the more experienced the developers are (although it is unknown how much time is required to become an AO expert). Hence, the experiment can give only statements about developers that are relatively new to the area of aspect-oriented software development.

Another issue is the characteristic of the programming tasks. Since no common aspect-oriented benchmark is available that could state "how complex real-world aspects might be" it is unclear whether the programming tasks are representative. Definitively, a problem in the aspect-oriented community, called the aspect-interference problem [14] is not part of the experiment. This problem describes the situation where different aspects influence each other because they are either applied to common places in the code or because they share some common state. However, similar to the comment above, it is unknown until now how often such situations appear.

3.6. Experiment Execution

20 subjects participated in the experiment. All subjects were students within their fifth semester or later and were selected using convenience sampling [24]. The experiment was performed in multiple sessions with different students (since it was not possible find a common date for all students). The subjects were divided into two groups, one group worked on the previous tasks in the object-oriented (OO) way and later on in the aspect-oriented (AO) way, the other group vice versa. Based on the results of a questionnaire, both groups had a similar number of subjects with high as well as with low development experience, whereby experience was measured based on each subject's personal estimation. The aspect-oriented groups did not get any hints how to solve the task using the aspect-oriented constructs. Only a sample piece of code showing the desired result in the object-oriented code was delivered. Furthermore, aspect declarations (without pointcuts and advices) were given to the students.

For each task, all subjects received a set of JUnit test cases that each subject could execute within its IDE. The set of test cases covered all subtasks that needed to be fulfilled. For example, for the first task there was an amount of test cases that covered all methods to be logged and checked whether the expected log-entries corresponded to the logs actually performed by the code. In order to finish a task and to switch to the next task, subjects were required to pass all test cases of the current task. The subjects were not required but allowed to use the test cases while they fulfilled their tasks. There were no automatic tests that checked whether the subjects used the right technique to solve the task. For example, we did not automatically check whether a subject tried to solve the aspect-oriented logging task by inserting log-statements in each method. This check was performed manually by the supervision of the experiment. Only if the successful test cases were shown to the supervision, subjects were permitted to switch to the next programming task.

4. Results and Analysis

First, section 4.1 describes the measured data from the experiments and performs some simple descriptive statistics. In 4.2 we first perform an analysis on the development times using an ANOVA in order to check the influence of language and tasks on the experiment results. Then, we analyze the overall impact of the differences in the development time. After comparing the development times task by task, we group the subjects and repeat the statistical analysis. Finally, we give a first summary of the results.

4.1. Descriptive Statistics

Table 1 shows the collected data from the experiment for all 20 subjects where all times are expressed in seconds. For each task we described the time measured for the object-oriented as well as for the aspect-oriented approach in separate, successive columns. For example, columns two and three describe the time (in seconds) measured for each subject for the logging task using the aspect-oriented (column two) and the object-oriented (column three) approach. Table 1 shows that for example subject 1 required 4585 seconds to solve the logging task in the object-oriented way and 4941 seconds to solve it in the aspect-oriented way.

Based on this we performed a simple descriptive statistics on the given data (see Table 2) where we computed for each task the sum of all times required to

Table 1. Descriptive Statistics

Subject	Task															
	Logging		Parameter Null		Synchronized		Player Check		Notify Observers		Observers Null		Refresh Constraint		Label Value Check	
	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO
1	4941	4585	466	1139	1138	521	334	29	631	162	429	207	185	138	1541	232
2	876	4951	1183	712	604	452	177	42	453	151	695	91	282	104	4078	125
3	2287	3044	384	797	389	424	543	32	893	128	152	28	496	146	725	80
4	4945	7976	2751	891	877	1443	642	70	1404	201	266	236	660	160	3688	371
5	5497	6347	480	2065	773	870	1600	109	5442	292	1003	185	574	138	2153	196
6	5046	7300	545	1094	1504	963	532	133	1271	365	1238	313	299	190	929	233
7	690	2869	300	723	223	481	195	41	244	220	181	153	162	79	465	111
8	3787	4744	384	1523	1667	382	221	102	806	466	632	149	265	367	1485	265
9	9956	4855	926	986	1344	405	526	79	3222	165	279	10	232	172	2822	299
10	2610	2643	542	762	558	423	215	78	1413	88	881	13	253	106	3662	305
11	8175	12269	404	943	1293	634	1651	87	717	228	294	316	1074	130	2761	275
12	7206	5014	1539	1806	1024	692	670	66	521	215	2196	198	481	137	2837	146
13	901	3820	266	659	382	274	238	137	870	167	683	167	499	220	416	224
14	595	3383	444	777	269	477	176	49	177	195	171	183	400	122	910	236
15	3799	2755	377	448	851	301	197	32	2250	97	119	131	322	62	333	97
16	2033	3030	148	470	248	502	383	22	189	127	273	178	310	85	734	120
17	711	3278	218	641	224	392	165	44	483	131	310	170	403	85	459	100
18	4544	4615	445	713	554	510	320	86	2051	231	551	174	818	113	1729	162
19	3598	3060	237	460	302	451	259	53	1627	107	774	82	276	105	934	294
20	5103	6733	5193	1098	1184	961	503	80	750	731	298	376	344	278	4359	200

Table 2. Results (in seconds)

Function	Task															
	Logging		Parameter Null		Synchronized		Player Check		Notify Observers		Observers Null		Refresh Constraint		Label Value Check	
	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO
Sum	77300	97271	17232	18707	15408	11558	9547	1371	25414	4467	11425	3360	8335	2937	37020	4071
max	9956	12269	5193	2065	1667	1443	1651	137	5442	731	2196	376	1074	367	4359	371
min	595	2643	148	448	223	274	165	22	177	88	119	10	162	62	333	80
arith. mean	3865	4864	862	935	770	578	477	69	1271	223	571	168	417	147	1851	204
med	3793	4600	445	787	689	479	327	68	838	181	370	172	333	134	1513	212
std. dev.	2627	2356	1184	432	464	284	426	34	1251	151	494	97	226	73	1344	83
Sum _{ao} - Sum _{oo}	-19971		-1475		3850		8176		20947		8065		5398		32949	
mean diff.	-999		-74		193		409		1047		403		270		1647	
med diff	-977		-328		122		270		596		246		243		1265	
Sum _{ao} / Sum _{oo}	79,47%		92,12%		133,31%		696,35%		568,93%		340,03%		283,79%		909,36%	
# (diff < 0)	15		17		8		0		1		4		1		0	

solve this task (row 1), the maximum and the minimum number of seconds required for each task (row two and three), the average time required for each task (row four), the median (row five) and the standard deviation (row six). In row seven we computed the difference between the sum of aspect-oriented and object-oriented times. Then, we computed the arithmetic average difference between such times (row eight) and the median of this difference (row nine). Then, we computed the ratio of aspect-oriented development times and object-oriented development times (row 10)

and we computed for how many subjects the aspect-oriented development time is less than the object-oriented development time. Figure 13 gives a graphical representation of the maximum, minimum and median development time for each task (in seconds).

A first glimpse reveals that the results largely differ among the subjects for the object-oriented as well as for the aspect-oriented technique: in all tasks the minimum as well as the maximum values of times differ largely from the average time for this task. Hence, there is a huge gap between the minimum and

maximum time for each task. For example, the minimum time required to solve the object-oriented logging task (2643 s) is about 20% of the maximum time required for the corresponding task (12269 s). For the same task the minimum aspect-oriented time is only 6% of the maximum aspect-oriented development time. Hence, the standard deviation for all tasks is quite high.

For the first two tasks we see that the ratio of AO and OO times is less than 100%. Furthermore, we see that there is quite a large number of subjects where the AO-time is less than the OO-time (which could be an indicator that for these tasks the application of aspect-oriented techniques takes significant less time than the application of Java). For the synchronization the AO-OO-ratio is 133.31% but 8 participants were faster using the aspect-oriented approach. Here, it might be unclear whether the use of aspect-oriented constructs saves time. For the other tasks we see that the AO-OO ratio is quite high, for the level value check it is even more than 900%. Here, this could be a sign that the use of aspect-oriented constructs requires significant more time than the use of object-oriented constructs.

Figure 13. Maximum, minimum and median per task

4.2. Variance Analysis

First, we analyze whether the different times measured are caused by the different languages (AspectJ and Java) and the different tasks. Our experiment follows a complete 2x2 design, where we have the two factors task (with nine levels) and language (with two levels). Since our subjects fulfilled all tasks using both

languages, a possibility is to use a two-factor ANOVA with repeated measurement. Although the underlying data is not normal distributed (which we checked via a Shapiro-Wilk test as well as Kolmogorov-Smirnov test on all tasks where both tests rejected the normal distribution hypothesis), the ANOVA is stable against violations of its preconditions (because the number of subjects is more than 10, cf. [2]).

For the task factor, the Mauchly test sphericity is significant which means that a correction is required for the significance test. Using the Greenhouse-Geisser correction, we retrieve a significant influence of the task factor ($F(1.58)=55.02$; $p \sim 0,000$). In addition, a significant main effect for the factor language was observed. We receive a significant influence of the language on the results ($F(1)=13.75$; $p=0,001$). Hence, the factor tasks as well as the factor language is a significant influencing factor for the measured development times.

Importantly, there is a significant interaction effect between task and language (using the Greenhouse-Geisser correction with $F(2,81)=9,85$, $p \sim 0,000$). This effect can be interpreted in a way that for some tasks the use of aspect-oriented constructs takes less time while for others it takes more time. Hence, an effect caused by the chosen language can be abrogated by the task effect of a different task.

4.3. Comparing Aspect-Oriented and Object-Oriented Development Times (Task by Task)

Because of the previous variance analysis we can already say that there is a significant difference of development times using different languages for different tasks. However, it does not say for what tasks there is such a significant difference.

In order to check for what tasks significant differences exist between the development times we performed a Wilcoxon-test (a t-test, which would be more desirable cannot be applied here since the underlying data is not normally distributed). Hence, we checked the following hypothesis and the corresponding alternative hypothesis:

- H0: The medians of the aspect-oriented and object-oriented times for a task are equal.
- H1: The medians are not equal.

The results of the Wilcoxon-test can be seen in Table 3. It turns out that for all tasks except the synchronization task the hypothesis H0 is rejected on a significance base of 95%. However, the implications of the results differ. For the first two tasks (logging and

parameter null) there is a significant positive impact of using the aspect-oriented implementation technique (see column rank sum). For the synchronization task there is no significant difference. For the tasks 4-9 there is a significant negative impact of using the aspect-oriented technique.

Table 3. Wilcoxon signed rank test on all tasks

Task		N	avg. rank	rank sum	Z	asymptotic significance
Logging	neg. ranks	5	9,6	48	-2,128	0,033
	pos. Ranks	15	10,8	162		
Parameter Null	neg. ranks	3	17,33	52	-1,979	0,048
	pos. Ranks	17	9,29	158		
Synchronization	neg. ranks	12	12	144	-1,456	0,145
	pos. Ranks	8	8,25	66		
Player Check	neg. ranks	20	10,5	210	-3,92	0
	pos. Ranks	0	0	0		
Notify Observers	neg. ranks	19	11	209	-3,883	0
	pos. Ranks	1	1	1		
Observers Null	neg. ranks	16	12,38	198	-3,472	0,001
	pos. Ranks	4	3	12		
Refresh Constraint	neg. ranks	19	10,79	205	-3,733	0
	pos. Ranks	1	5	5		
Label Value Check	neg. ranks	20	10,5	210	-3,92	0
	pos. Ranks	0	0	0		
Level Check	neg. ranks	18	11	198	-3,472	0,001
	pos. Ranks	2	6	12		

4.4. Overall Impact

Next, we check whether there is a significant impact of using the aspect-oriented language compared to using the object-oriented language in general, where we consider the sums of development times for all tasks for each single subject. I.e. for each subject we compute the sum of AO development times and the sum of OO development times. This approach assumes that the experiment addresses the fulfillment of all tasks instead of each single one.

Again, we perform the Wilcoxon test. Again, we receive a significant result (see Table 4). By additionally considering the rank sums this means that using AspectJ takes significant more time then than using Java (if the whole experiment is considered as one single task).

4.5. Grouped Subjects

The previous analysis does not consider any differences between the subjects. It is common practice to use a block design where experience of subjects is used as a blocking criterion (see for example [24]). Typically, in such cases experience is measured by a questionnaire filled in by each subject. However, we consider a blocking based on a questionnaire to be rather inappropriate because a questionnaire implies a

subject's subjective impression. Furthermore, it is unclear whether a subject having good experience in a certain language can learn and apply another language more easily.

Table 4. Wilcoxon test on development times

Task		N	avg. Rank	rank sum	Z	asymptotic significance
development times	neg. ranks	15	12,3	157	-2,128	0,003
	pos. Ranks	5	5	25		

We rather think that the development times measured in the experiment can be better used as a grouping criterion. According to [18] we decided to analyze the data on quartiles constructed from the underlying development times. We separately analyzed the data by constructing quartiles based on the object-oriented construction time as well as on the aspect-oriented construction time. Here, we assume that the sum of object-oriented development time is a subject's indicator for how experienced the subject is. Furthermore, we assume that the sum of aspect-oriented development times is a subject's indicator for how good the subject understood the aspect-oriented language constructs.

Figure 14. Quartiles based on object-oriented development times

Figure 14 illustrates the construction of the 2-quartiles based on the object-oriented construction times. The first quartile (which represents the good performing OO developers) contains all subjects whose development time is less than 6771 seconds. The quartile consists of the subjects 2, 3, 7, 10, 13-17 and 19.

Figure 15. Quartiles based on aspect-oriented development times

Figure 15 illustrates the construction of the 2-quartiles based on the aspect-oriented construction times. The first quartile (representing the good AO performers) contains all subjects whose development time is less than 10279 seconds. The quartile consists of the subjects 2, 3, 7, 8, 13-17 and 19.

One interesting observation is that there is a large overlap between the first quartile of the aspect-oriented and the first quartile of the object-oriented development times. Only subjects 10 and 8 are different (the first subject is in the first OO-quartile, the second subject is in the first AO-quartile). This is an indicator that the more experienced developers are, the better they use the aspect-oriented constructs (as shown in the variance analysis).

4.5.1. Analysis based on OO development times: For the first quartile, we see that there is a significant difference for task 1 and for tasks 4-9 (see Table 5). Again, for task 1 the application of aspect-oriented constructs decreased the development time, while for tasks 4-9 the development times increased. An interesting observation is that there is no significant result; neither for the second nor for the third task although in the overall analysis (section 4.3) there was a significant result for the second task. However, we see that with $p=0,074$ the second task is still “quite probable” although not under a significance level of 5%.

Table 5. Wilcoxon test for OO developers (1st quartile)

Task		N	avg. rank	rank sum	Z	asymptotic significance
Logging	neg. ranks	2	3,5	7	-2,09	0,037
	pos. Ranks	8	6	48		
Parameter Null	neg. ranks	1	10	10	-1,785	0,074
	pos. Ranks	9	5	45		
Synchronization	neg. ranks	4	5	20	-0,764	0,445
	pos. Ranks	6	5,83	35		
Player Check	neg. ranks	10	5,5	55	-2,803	0,005
	pos. Ranks	0	0	0		
Notify Observers	neg. ranks	9	6	54	-2,701	0,007
	pos. Ranks	1	1	1		
Observers Null	neg. ranks	8	6,5	52	-2,499	0,012
	pos. Ranks	2	1,5	3		
Refresh Constraint	neg. ranks	10	5,5	55	-2,803	0,005
	pos. Ranks	0	0	0		
Label Value Check	neg. ranks	10	5,5	55	-2,803	0,005
	pos. Ranks	0	0	0		
Level Check	neg. ranks	9	5,67	51	-2,395	0,017
	pos. Ranks	1	4	4		

For the second quartile, we do not get any significant result for the first and the second task. For all other tasks we get a significant negative impact of using the aspect-oriented language (see Table 6).

Table 6. Wilcoxon test for OO developers (2nd quartile)

Task		N	avg. rank	rank sum	Z	asymptotic significance
Logging	neg. ranks	3	6	18	-0,968	0,333
	pos. Ranks	7	5,29	37		
Parameter Null	neg. ranks	2	9,5	19	-0,866	0,386
	pos. Ranks	8	4,5	36		
Synchronization	neg. ranks	8	5,88	47	-1,988	0,047
	pos. Ranks	2	4	8		
Player Check	neg. ranks	10	5,5	55	-2,395	0,017
	pos. Ranks	0	0	0		
Notify Observers	neg. ranks	10	5,5	55	-2,803	0,005
	pos. Ranks	1	0	0		
Observers Null	neg. ranks	8	6,38	51	-2,395	0,017
	pos. Ranks	2	2	4		
Refresh Constraint	neg. ranks	8	5,67	51	-2,395	0,017
	pos. Ranks	2	4	4		
Label Value Check	neg. ranks	10	5,5	55	-2,803	0,005
	pos. Ranks	0	0	0		
Level Check	neg. ranks	9	6	54	-2,701	0,007
	pos. Ranks	1	1	1		

4.5.2. Analysis based on AO development times: Repeating the same analysis on the first quartile of the ao development times, we receive for the first task a significant positive impact of using the aspect-oriented language, no significance for task 2, 3 and 9 and a significant negative impact for tasks 4-8. Here, the missing significance for task 2 is again interesting. Furthermore, the missing significance of task 9 (which was in all previous analysis significant) must be mentioned.

Table 7. Wilcoxon test for AO developers (1st quartile)

Task		N	avg. rank	rank sum	Z	asymptotic significance
Logging	neg. ranks	2	3	6	-2,19	0,028
	pos. Ranks	8	6,12	49		
Parameter Null	neg. ranks	1	9	9	-1,89	0,059
	pos. Ranks	9	5,11	46		
Synchronization	neg. ranks	4	6,25	25	-0,26	0,799
	pos. Ranks	6	5	30		
Player Check	neg. ranks	10	5,5	55	-2,80	0,005
	pos. Ranks	0	0	0		
Notify Observers	neg. ranks	9	6	54	-2,70	0,007
	pos. Ranks	1	1	1		
Observers Null	neg. ranks	8	6,5	52	-2,50	0,012
	pos. Ranks	2	1,5	3		
Refresh Constraint	neg. ranks	9	5,89	53	-2,60	0,009
	pos. Ranks	1	2	2		
Label Value Check	neg. ranks	10	5,5	55	-2,80	0,005
	pos. Ranks	0	0	0		
Level Check	neg. ranks	8	5,38	43	-1,58	0,114
	pos. Ranks	2	6	12		

Repeating the same analysis on the 2nd quartile, we do not get any significant difference for the first and the second task while for all other tasks there was a significant negative impact (we neglect to give the corresponding results of the calculation here).

5. Related Work

The work that is directly related to our experiment is the one conducted by Walker et al [23]. Here, a number of subjects performed a number of tasks on an object-oriented system using the aspect-oriented languages AspectJ. Among many others, the time to complete the task was measured. The experiment also has some “mixed results” similar to our analysis of the average programmers and novices: that there are situations where aspect-orientation has a positive impact and others where it has not. The main difference to our work is that the analysis of the experiment was done in a qualitative way, i.e. no significance tests have been performed on the data. Furthermore, the number of subjects in the experiment was quite low which does not permit to make any significant statement about the data. Furthermore, the tasks given to the subjects differ widely with respect to what has to be done.

Further related approaches are for example studies on the maintainability of aspect-oriented software (cf. [1]). The result of the study was that no difference between aspect-oriented and object-oriented software could be measured with respect to maintainability.

Another empirical investigation of aspect-oriented programming constructs can be found in [13]. Although the experiment is just performed on a small number of subjects, it has some interesting insights, where the most important one is that no impact of aspect-oriented programming on software design modularity and size could be measured.

Also, studies about the design stability (see [5]) or language specific features such as the study performed in [3] are in that way related that the impact of aspect-oriented language constructs on some piece of software is being tested. The main difference between those approaches and the here described experiment is, that we try to focus only on the development time and neglect currently all other desirable attributes of software.

Further related works are those ones that study the impact of programming features such as (see for example [16, 17]): there special characteristics of programming languages are being evaluated within corresponding experiments.

6. Summary and Discussion

The intention in this paper was to study the different development times for static crosscutting code using Java and AspectJ in order to test whether there are significant differences.

In order to study this question, an experiment was performed where nine tasks were given to 20 subjects, whereby the tasks differ with respect to the time required to solve them: while some tasks (such as logging) required the editing of 110 code targets, others (such as the refresh task) required only to edit 8 code targets. During the experiment the development times were measured.

A first interesting result was that for all tasks with less than 36 code targets a significant negative impact of using the aspect-oriented language could be measured. The only exception from this is that for the level check task (with only seven code targets). Here, the 2nd quartile of the aspect-oriented developers, which represents underperforming aspect-oriented developers within the experiment, gives no significant result.

Furthermore, we see that an improvement of development time can be measured for the first and the second task if we ignore the different characteristics of the subjects. If we consider good performing subjects we can only measure a positive impact for the first task.

Next, it must be noted that the third task (which has 55 code targets) does not show any significant differences using the object-oriented or aspect-oriented language. It is interesting, because this task has more code targets than task two (for which under certain circumstances a positive impact was measured). A characteristic of the third task was (from the object-oriented perspective) that absolutely the same code (the synchronize block) could be inserted into the code targets (which is not the case in task one and two). This could be an indicator that using an aspect-oriented language does not only depend on the number of code targets, but also on the similarity of the code snippets to be inserted.

It should be emphasized that a critical point in the experiment is teaching: within the experiment aspect-orientation was taught within 1.5 hours. Maybe a more intensive training in aspect-orientation would significantly change the result. However, how much training is required in order to apply aspect-orientation is unknown until now. In the same context, it would be desirable to see the results of professional software developers with AspectJ experience in the experiment. However, it is until now even unclear whether such AspectJ experts exist in industry.

This experiment focuses on development time (i.e. development costs) and does not consider the role of maintenance. Hence, it would be interesting to compare the development costs with the maintenance costs in order to get a more complete picture of the potential role of aspect-orientation in the area of software

engineering. Furthermore, the experiment does only consider the static part of AspectJ's pointcut language. It is up to future work to study how the dynamic language features influence the development time.

The approximation that experienced developers require at least 36 code targets in order to decrease the development time is already a first good result. However, a more fine-grained analysis is desirable which permits to quantify how large the expected effect is. Hence, a repetition of the experiment with more subjects is desired in order to apply a statistical method (such as the t-test) that permits to quantify the expected development times.

7. References

- [1] Bartsch, M.; Harrison, R.: An exploratory study of the effect of aspect-oriented programming on maintainability, *Software Quality*, Vol. 16, No 1, 2007, pp. 23-44.
- [2] Bortz, J.: *Statistik für Sozialwissenschaftler*, 5th ed., Springer, 1999.
- [3] Coelho, R.; Rashid, A.; Garcia, A.; Ferrari, F.; Cacho, N.; Kulesza, U.; von Staa, A.; Pereira de Lucena, C.: Assessing the Impact of Aspects on Exception Flows: An Exploratory Study *Proceedings of the European Conference on Object-Oriented Programming*, 2008, pp. 207-234.
- [4] Filman, R.; Elrad, T.; Clarke S.; Aksit, M. (eds.): *Aspect-Oriented Software Development*, Addison-Wesley Longman, Amsterdam, 2004.
- [5] Greenwood, P.; Bartolomei, T.; Figueiredo, E.; Dosea, M.; Garcia, A.; Cacho, N.; Sant'Anna, C.; Soares, S.; Borba, B.; Kulesza, U.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study, *Proceedings of ECOOP 2007*, pp. 176-200.
- [6] Gybels, K.; Brichau, J.: Arranging language features for more robust pattern-based crosscuts. *Proceedings of AOSD*, 2003, pp- 60-69.
- [7] Hanenberg, S.: *Design Dimensions of Aspect-Oriented Systems*, PhD thesis, University of Duisburg-Essen, Institute for Computer Science and Business Information Systems, 2006.
- [8] Juristo, N.; Moreno, A.: *Basics of Software Engineering Experimentation*, Kluwer Academic Publishers, 2001.
- [9] Kellens, A.; Mens, K.; Brichau, J., Gybels, K.: Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts, *Proceedings of the European Conference on Object-Oriented Programming*, 2006, 501-525.
- [10] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwin, J.: *Aspect-Oriented Programming*. *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 1997, p.220-242.
- [11] Kim, M.; Sazawal, V.; Notkin, D.; Murphy, G.: An empirical study of code clone genealogies, *Proceedings of ESEC/SIGSOFT FSE 2005*, pp. 187-196.
- [12] Laddad, R.: *AspectJ in Action: Practical Aspect-oriented Programming*, Manning, 2003
- [13] Madeyski, L., Szala, L.: Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study, *Software, IET*, Vol. 1, No. 5. (2007), pp. 180-187.
- [14] Nagy, I.; Bergmans, L. Aksit, M.: Composing aspects at shared join points, *Proceedings of International Conference NetObjectDays (NODE)*, *Lecture Notes in Informatics P-69*, Gesellschaft für Informatik (GI), Erfurt, Germany, 2005, pp. 19-38
- [15] Ostermann, K.; Mezini, M.; Bockisch, C.: Expressive Pointcuts for Increased Modularity. *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 2005, pp. 214-240.
- [16] Prechelt, L.; Unger, B.; Philippsen, M.; Tichy, W.: A Controlled Experiment on Inheritance Depth as a Cost Factor for Maintenance. *Journal of Systems and Software*, Elsevier, 65(2):115-126, February 2003.
- [17] Prechelt, L.: An empirical comparison of seven programming languages. *IEEE Computer* 33(10):23-29, October 2000.
- [18] Prechelt, L.: *Kontrollierte Experimente in der Softwaretechnik*, Springer, 2001.
- [19] Shull, F., Singer, J., Sjøberg, D. (eds.), *Guide to Advanced Empirical Software Engineering*, Springer, 2008.
- [20] Steimann, F.: The paradoxical success of aspect-oriented programming, *ACM SIGPLAN Notices*, Volume 41 , Issue 10 (October 2006), pp. 481 - 497
- [21] Tichy, W.: Should Computer Scientists Experiment More? *IEEE Computer* 31(5), 1998, pp. 32-40.
- [22] De Volder, K.; D'Hondt, T.: *Aspect-Oriented Logic Metaprogramming*, in [4], 2004.
- [23] Walker, R.; Baniassad, E.; Murphy, G.: An Initial Assessment of Aspect-oriented Programming, *Proceedings of the 21st International Conference on Software Engineering (16-22 May 1999, Los Angeles, CA, USA)*.
- [24] Wohlin, C., Runeson, P., Höst, M.: *Experimentation in Software Engineering: An Introduction*, Springer, 1999