IS THE APPLICATION OF ASPECT-ORIENTED PROGRAMMING CONSTRUCTS BENEFICIAL?
First Experimental Results

Sebastian Kleinschmager and Stefan Hanenberg
Department for Computer Science and Business Information Systems, University of Duisburg-Essen, Germany

Keywords: Aspect-Oriented Software, Experiment, Empirical Research.

Abstract: Aspect-oriented software development is an approach which addresses the construction of software artefacts which 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 application of aspect-oriented constructs has a measurable, positive impact on the construction of software artefacts. This paper addresses this issue by an empirical study which compares the specification 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.

1 INTRODUCTION

A typical argument for aspect-oriented software development (AOSD, (Filman et. al, 2004) is that aspects permit a better modularization of so-called crosscutting concerns. Such arguments focus on the readability and maintainability of software constructed from the aspect-oriented approach (see further (K. De Volter and T. D'Hondt, 2004)). However, such arguments address pieces of software that are constructed in order to be maintained and extended for some amount of time – from a software manager's point of view the use of aspect-oriented techniques rather looks like an investment into the future. But such arguments do not address whether aspect-oriented techniques can be applied to reduce code in order to safe development time. However, it seems pursuable that aspect-orientation already safes time because of a reduced number of lines of code caused by the underlying modularization techniques.

One example for aspect-oriented techniques that is quite often cited in literature is logging (c.f. (J. Janssen and W. Laatz, 2003) among many others): invocations to the logger need to appear in a large number of modules. Furthermore, the pieces of code that are 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 specify 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 is seems clear that aspect-oriented programming techniques decrease significantly the development time.

From the other perspective, one can argue that aspect-oriented programming bring additional abstractions and therefore additional complexity into the development of software (cf. e.g. (F. Steimann, 2006)). 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 (W. Tichy, 1998) empirical methods (cf. (N. Juristo and A. Moreno, 2001; Shull et. al, 2008)) are an approach to address this problem. This would permit to strengthen (or weaken) arguments based on observed data.

This paper introduces a controlled experiment that studies the development costs in terms of

Kleinschmager S. and Hanenberg S. (2009).
IS THE APPLICATION OF ASPECT-ORIENTED PROGRAMMING CONSTRUCTS BENEFICIAL? - First Experimental Results .
In *Proceedings of the 11th International Conference on Enterprise Information Systems - Databases and Information Systems Integration*, pages 196-201
DOI: 10.5220/0002007301960201

development time caused by the specification of redundant code in the object-oriented programming language Java in comparison to the aspect-oriented programming language AspectJ. Section 2 describes the experimental design and argues why we only focus on static crosscutting code. Section 3 describes how the experiment was performed. Section 4 evaluates the results. After discussing the results in section 5, we present related work in section 6. Finally, we conclude the paper.

2 EXPERIMENT

Our study relies on static crosscutting code, i.e. code which does not need any dynamic condition fulfilled at runtime in order to determine whether it should be executed. In aspect-oriented terms spoken, we study only aspects with underlying static join points (cf. e.g. (S. Hanenberg, 2006)). 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. (K. Gybels and J. Brichau, 2003; L. Prechelt, 2001; Ostermann et. al, 2005)) we reduce our view on aspect-orientation to static elements because static crosscutting can be unambiguously determined, i.e. for the resulting redundant code fragments it can be clearly defined where and how they should appear.

Next, we are interested in the development time required to fulfill a programming task that consists of the specification of redundant code. Hence, we defined an experiment where developers are requested to fulfill a programming task in a pure object-oriented as well as in an aspect-oriented way. In this experiment, time is the free variable and the number of fulfilled programming tasks represents the dependent variable.

Within the experiment subjects were ask to specify a number of redundant lines of code into a target application. As a target application, we used a self-specified 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 two tasks needed to be done, each one in pure Java (version 1.6) and AspectJ (version 1.6.1) whereby the order of the programming language was randomly chosen.

First Task: Logging

The first was to add a logging-feature to the application, where each method (but no constructors) should be logged. Thereto, a corresponding logger-interface was provided that expects from each method its return type, the name of the classes where the method was declared in, an array type String with the formal parameter names and an array of its actual parameters.

```
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: Exemplary log-invocation in pure Java.

For the object-oriented solution, for a public method m in class C with parameter types int and A (and the corresponding parameter names i and a) and the return type R the corresponding invocation of the logger in pure Java that needed to be defined by the developer looks like shown in Figure 1.

The expected type names are simple names, i.e. no package names were expected by the logger. Altogether, such a line needed to be added to all 110 methods.

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 specify 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.

```
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: Exemplary log-invocation in AspectJ.

Second Task: Nullpointer-Checks

The second task was to add nullpointer-checks to all non-primitive parameters of all methods in the application (without constructors). In case one of the non-primitive parameters was null, an InvalidGameStateException should be thrown (which was part of the application). For the object-oriented solution, 36 methods needed to be adapted.

3 EXPERIMENT EXECUTION

20 subjects participated in the experiment. The subjects performed their tasks in identical environmental settings (machines, rooms, etc.). As an IDE, Eclipse has been used. All subjects were students within their fifth semester or later.

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

After dividing the subjects into two groups, one group worked on the previous tasks in the object-oriented way and later on in the aspect-oriented way, the other group vice versa. Based on the results of the questionnaire, both groups had a similar number of subjects with high as well as with low development experience. The aspect-oriented groups did not get any hints how to solve the task. The only thing delivered to them was the aspect declaration (without pointcuts and advices).

For each task, all subjects received a set of JUnit test cases that each subject could execute within his IDE. The set of test cases covered all subtasks that needed to be fulfilled. For example, there was an amount of test cases that covered for the first task all methods to be logged that 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. The actions performed by each subject were logged in a way that permitted later on to compute how many subtasks have been achieved at what points in time. Furthermore, a screen recorder ran throughout the whole experiment for each subject in order to permit later on to extract information about each subject.

4 RESULTS

Figure 3 shows the collected data from the experiment (all times are expressed in seconds). For both tasks, the complete time to fulfill the task was measured. In order to remove the time required by the participants to understand the task, the starting point was set to the moment when subjects started to write code. The end point was set to the moment when the subjects fulfilled all test cases. Furthermore, the descriptive statistics, i.e. the sum, arithmetic mean, minimum, maximum and standard derivation is shown in Figure 3.

	logging				null-pointer check			
subject	t_{oo} (sec.)	t_{ao} (sec.)	t_{oo}-t_{ao}	t_{ao}/t_{oo}	t_{oo} (sec.)	t_{ao} (sec.)	t_{oo}-t_{ao}	t_{ao}/t_{oo}
1	4556	4912	-356	107,81%	1110	437	673	39,37%
2	4922	847	4075	17,21%	683	1154	-471	168,96%
3	3015	2258	757	74,89%	768	355	413	46,22%
4	7947	4916	3031	61,86%	862	2722	-1860	315,78%
5	6318	5468	850	86,55%	2036	451	1585	22,15%
6	7271	5017	2254	69,00%	1065	516	549	48,45%
7	2840	661	2179	23,27%	694	271	423	39,05%
8	4715	3758	957	79,70%	1494	355	1139	23,76%
9	4826	9902	-5076	205,18%	957	897	60	93,73%
10	2614	2581	33	98,74%	733	513	220	69,99%
11	12240	8146	4094	66,55%	915	375	540	40,98%
12	4985	7177	-2192	143,97%	1777	1510	267	84,97%
13	3791	872	2919	23,00%	630	237	393	37,62%
14	3354	566	2788	16,88%	739	415	324	56,16%
15	2727	3770	-1043	138,25%	419	348	71	83,05%
16	3001	2004	997	66,78%	441	119	322	26,98%
17	3249	682	2567	20,99%	612	189	423	30,88%
18	4586	4515	71	98,45%	684	416	268	60,82%
19	2565	3569	-1004	139,14%	390	208	182	53,33%
20	6705	5074	1631	75,67%	1069	5164	-4095	483,07%

	logging		null-pointer check	
	t_{oo} (sec.)	t_{ao} (sec.)	t_{oo} (sec.)	t_{ao} (sec.)
sum	96227	76695	18078	16652
min	2565	566	390	119
max	12240	9902	2036	5164
mean	4811	3834,8	903,9	832,6
std. deriv.	2367	2624,3	434,21	1184

Figure 3: Results.

First, it turns out that there is a large variety in the development speed among the subjects. For example, for the object-oriented logging task the slowest subject (subject no. eleven) is approximately five times slower than the fastest subject (subject

19). The same is true for the object-oriented null-pointer check task: while here the fastest subject is still subject 19, the slowest one is subject five. For the aspect-oriented solutions, the time differences between slowest and fastest are even more extreme: in the logging task, the slowest subject is more than 17 times slower than the fastest one; in the null-pointer check task it is more than 43 times slower.

Considering the comparison of time required for the object-oriented versus the time required for the aspect-oriented solution, it turns out that for the logging task 5 subjects were slower using the aspect-oriented approach. In the second task, three subjects were slower using the aspect-oriented approach.

Although we do not consider the comparison of absolute times between the subjects to be meaningful (due to the different development speeds), it turns out that the sums of times for the aspect-oriented solutions are in both cases less than the sums of times for the object-oriented solutions.

If we concentrate on the maximum increase of development speed for a single subject, we can see that for the logging task the use of aspect-orientation is only 17% of the time required for the object-oriented solution (subject 14). For the null-pointer check task we see that the time required by the aspect-orientated solution is only 22% of the time required by the object-oriented solution (subject 5). If we concentrate on the maximum decrease of development speed, we see that for the logging task subject 9 spent more than twice the time for solving it in an aspect-oriented way. For the null-pointer check we see that it took subject 20 five times more time to solve it in an aspect-oriented way.

In order to check whether there is a significant difference in the object-oriented and the aspect-oriented approach, we first check whether a pair-samples t-test can be applied. A pair-samples t-test requires (according to e.g. (J. Bortz, 1999)) that the underlying sample comes from a normally distributed population. Therefore, we applied the Shapiro–Wilk test (S. S. Shapiro and M. B. Wilk, 1965) which checks these characteristics. However, the result of the test is that the hypothesis, that the sample comes from a normally distributed population, needs to be rejected.

Hence, we need to apply a less restrictive statistical method: the Wilcoxon signed-rank test (see e.g. (J. Bortz, 1999)) which compares two related samples with respect to their central tendency.

According to this, we formulate the null hypothesis :

H0: The median of the object-oriented and the aspect-oriented solution are equal.

The alternative hypothesis is

H1: The median of the object-oriented and the aspect-oriented solution differs.

Performing the Wilcoxon-test on the logging task, we find with under significance level of 5% that here is a difference in the medians of the object-oriented and the aspect-oriented solution (hence, the null hypothesis is rejected). A comparison of positive and negative ranks reveals that the median of the aspect-oriented solution is significantly lower than the median of the object-oriented solution.

Repeating the same test on the null-pointer check reveals the same results.

Hence, in both cases we can determine that aspect-oriented construction reveals better results that the object-oriented way. In both cases this difference is significant.

5 DISCUSSION

Our intention was to study the different development times for static crosscutting code using Java and AspectJ. Thereto, two tasks were given to subjects (logging and null-pointer check). It turned out, that we were able to detect a significant difference in the development time in the logging example as well as in the null-pointer check examples. From that perspective is looks obvious that the application of aspect-oriented constructs for the specification of redundant code is more appropriate then the use of object-oriented constructs. However, we should also be aware of the kind of redundant code specifications: in both cases the number of redundant code lines is relatively high (110 and 36 lines of code). We think that we can conclude here that aspect-oriented constructs turn out to be beneficial in situations where the number of redundant code lines is relatively high. Whether the application of aspect-oriented constructs turns out to be beneficial in situations with rather a low number of redundant code can be doubted – however, the experiment is not able to make any statement about this. Furthermore, it should be noted that the impact of further potential influencing factors (for example the training of aspect-oriented constructs in the beginning of the experiment) has not been studied.

6 RELATED WORK

The work that is directly related to our experiment is the one conducted by Walker et al (Walker, et. al, 1999). Here, a number of subjects performed a number of tasks on an object-oriented system using the aspect-oriented language AspectJ. The main difference to our experiment is, that there developers had much more freedom about how to use the language in order to achieve a certain goal. Our experiment was in that way much more restricted. Further related approaches are for example studies on the maintainability of aspect-oriented software (cf. (M. Bartsch and R. Harrison, 2007)). Also, studies about the design stability (see (Greenwood et. al, 2007)) or language specific features such as the study performed in (Coelho et. al, 2008) 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 describes experiment is, that we try to focus only on the development time and neglects currently all other desirable attributes of software.

7 CONCLUSIONS AND FUTURE WORK

In this paper we presented an experiment that compares the use of aspect-oriented constructs for the purpose of specifying static crosscutting code with the corresponding specification using ordinary language constructs. In the experiment, 20 subjects performed two static crosscutting tasks on an object-oriented program using an object-oriented language as well as an aspect-oriented. The experiment showed that for tasks with a relative high number of redundant code lines the application of aspect-oriented techniques turns out to be useful.

Altogether, it should be mentioned that empirical knowledge, especially in the area of aspect-orientation, hardly exists and controlled experiments are rather rare. Hence, the here presented experiment cannot be considered as a final answer to the question of how beneficial aspect-orientation is. Instead, we rather consider this as a first and necessary step in order to explore quite a large field.

REFERENCES

Bartsch, M.; Harrison, R.: An exploratory study of the effect of aspect-oriented programming on maintainability, Software Quality, 2007.

Bortz, J.: Statistik für Sozialwissenschaftler, 5te Auflage, Springer, 1999

Box, G.; Jenkins, G. M.; Reinsel, G.: Time Series Analysis, Forecasting and Control, Prentice Hall, 1994.

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. ECOOP 2008: 207-234

Curtis, B.: Substantiating program variability, Proceedings of the IEEE, 69(7), July 1981.

De Volder, K.; D'Hondt, T.: Aspect-Oriented Logic Metaprogramming, in (Filman et. al, 2004), 2004.

Filman, R.; Elrad, T.; Clarke S.; Aksit, M. (eds.): Aspect-Oriented Software Development, Addison-Wesley Longman, Amsterdam, 2004.

Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dosea, Alessandro Garcia, Nelio Cacho, Cláudio Sant'Anna1, Sergio Soares, Paulo Borba, Uirá Kulesza, On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study, Proceedings of ECOOP 2007, pp. 176-200

Gybels, K.; Brichau, J.: Arranging language features for more robust pattern-based crosscuts. Proceedings of AOSD, 2003, pp- 60-69.

Hanenberg, S.: Design Dimensions of Aspect-Oriented Systems, PhD thesis, University of Duisburg-Essen, Institute for Computer Science and Business Information Systems, 2006.

Janssen, J.; Laatz, W.: Statistische Datenanalyse mit SPSS, 4th edition, Springer, 2003.

Juristo, N.; Moreno, A.: Basics of Software Engineering Experimentation, Kluwer Academic Publishers, 2001.

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.

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.

Ostermann, K.; Mezini, M.; Bockisch, C.: Expressive Pointcuts for Increased Modularity. Proceedings of European Conference on Object-Oriented Programming (ECOOP), 2005, pp. 214-240

Prechelt, L.: Kontrollierte Experimente in der Softwaretechnik, Springer, 2001.

Shapiro, S. S. and Wilk, M. B. (1965). "An analysis of variance test for normality (complete samples)", Biometrika, 52, 3 and 4, pages 591–611

Shull, F., Singer, J., Sjøberg, D. (eds.), Guide to Advanced Empirical Software Engineering, Springer, 2008.

Steimann, F.: The paradoxical success of aspect-oriented programming, ACM SIGPLAN Notices, Volume 41 , Issue 10 (October 2006), pp. 481 - 497

Tichy, W.: Should Computer Scientists Experiment More? IEEE Computer 31(5), 1998, pp. 32-40.

J.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).