Empir Software Eng (2018) 23:2018–2050
https://doi.org/10.1007/s10664-017-9580-7

An empirical study on the impact of AspectJ on software evolvability

Adam Przybyłek [1]

Published online: 6 December 2017

Abstract Since its inception in 1996, aspect-oriented programming (AOP) has been believed to reduce the effort required to maintain software systems by replacing cross-cutting code with aspects. However, little convincing empirical evidence exists to support this claim, while several studies suggest that AOP brings new obstacles to maintainability. This paper discusses two controlled experiments conducted to evaluate the impact of AspectJ (the most mature and popular aspect-oriented programming language) versus Java on software evolvability. We consider evolvability as the ease with which a software system can be updated to fulfill new requirements. Since a minor language was compared to the mainstream, the experiments were designed so as to anticipate that the participants were much more experienced in one of the treatments. The first experiment was performed on 35 student subjects who were asked to comprehend either Java or AspectJ implementation of the same system, and perform the corresponding comprehension tasks. Participants of both groups achieved a high rate of correct answers without a statistically significant difference between the groups. Nevertheless, the Java group significantly outperformed the AspectJ group with respect to the average completion time. In the second experiment, 24 student subjects were asked to implement (in a non-invasive way) two extension scenarios to the system that they had already known. Each subject evolved either the Java version using Java or the AspectJ version using AspectJ. We found out that a typical AspectJ programmer needs significantly fewer atomic changes to implement the change scenarios than a typical Java programmer, but we did not observe a significant difference in completion time. The overall result indicates that AspectJ has a different effect on two sub-characteristics of the evolvability: understandability and changeability. While AspectJ decreases the former, it improves one aspect of the latter.

Communicated by: Sven Apel

✉ Adam Przybyłek
adam.przybylek@gmail.com

[1] Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology, Narutowicza 11/12, 80-233 Gdansk, Poland

Keywords Aspect-oriented programming · AOP · Maintainability · Understandability · Separation of concerns · Controlled experiment

1 Introduction

The evolution of programming languages has been driven by the need to achieve better separation of concerns (SoC). SoC is a fundamental principle that addresses the limitations of human cognition for dealing with complexity (Chavez et al. 2011). It advocates that in order to master complexity, one has to deal with one important issue at a time; it does not mean completely ignoring the other issues, but temporarily forgetting them to the extent that they are irrelevant for the current topic (Dijkstra 1976). Software engineering experts (Dijkstra 1976; Parnas 1972; Yourdon and Constantine 1979) have suggested that the best way to achieve SoC is through decomposition of the system into logically cohesive and loosely-coupled modules, which can be developed and maintained in relative isolation. The expected benefits are improved understandability and traceability throughout the development process, and increased potential for evolution and reuse (Arnaoudova et al. 2008; Brito and Moreira 2004; Ossher and Tarr 2001).

Kiczales et al. (1997) found that the abstractions offered by traditional programming paradigms are insufficient to express some issues of the problem as first-class entities in the adopted language. If an issue cuts across the system's basic functionality, its implementation necessarily spreads over the program, causing code tangling and code scattering (Mens et al. 2004). Such an issue is called a cross-cutting concern. Efforts to deal with cross-cutting concerns resulted in aspect-oriented programming (AOP). The first aspect-oriented (AO) language was AspectJ developed as an extension to Java by Kiczales and his team (2001). Nowadays, AspectJ is the most mature and most widely used representative of AOP. The distinguishing characteristic of AO languages is that they provide quantification and obliviousness (Filman 2001). Quantification refers to the ability of aspects to affect multiple non-local places in the program, whereas obliviousness states that the base code has no knowledge of which aspects affect it where or when. Note, that there has been a debate on what are fundamental characteristics of AOP, and some researchers have claimed that neither quantification nor obliviousness is an essential ingredient of AOP (Rashid and Moreira 2006).

The idea behind AOP is to implement cross-cutting concerns as separate modules, called aspects (Kiczales et al. 2001). In AspectJ, an aspect can declare new programming constructs like pointcuts, advices, and intertype declarations. A pointcut is used to intercept well-defined points in the execution of the program, which are referred to as join points. An advice is a method-like construct providing behavior to be inserted at all join points picked out by the associated pointcut. In turn, intertype declarations allow aspects to alter the static structure of the system, e.g. by introducing new attributes or methods to a given class.

For a long time, AOP has been presented as a paradigm that improves SoC (Hoffman and Eugster 2009; Kulesza et al. 2006; Munoz et al. 2008; Sant'Anna et al. 2003; Tsang et al. 2004). However, every new promising paradigm begins with hype – the claims about its capabilities are exaggerated (Bezdek 1993). We were the first (Przybylek 2011) who pointed out that advocates of AOP often wrongly identified the lexical SoC provided by AOP with the meaning attributed to SoC by Dijkstra. Although classes are syntactically oblivious to aspects, their behavior cannot be adequately understood in isolation, because they are not semantically oblivious to aspects (Dantas and Walker 2006; Kienzle and Guerraoui 2002; Przybylek 2011, 2013). Since semantic dependencies are implicit, one needs to consider all aspects in the system when studying a class, because each

aspect can potentially change the class's logic. Moreover, it has been indicated (Przybylek 2010; Steimann 2006) that the paradigm that was proposed to "modularize the un-modularizable" actually runs contrary to some fundamental modularity principles, such as low coupling, information hiding, and explicit interfaces. Although several proposals have been developed to restore modularity to AOP (a comprehensive review is presented by (Steimann et al. 2010)), none of them has received common acceptance among practitioners. Furthermore, the new programming constructs that AOP provides to support software evolution, gives rise to new evolution-related problems, such as the fragile pointcut problem and the aspect composition problem (Mens and Tourwé 2008; Marot 2011). The question is when (and indeed whether) the gains obtained from the lexical separation of concerns outweighs the associated costs. In this research, we address this question from the viewpoint of software evolution. Accordingly, we perform two controlled experiments to determine if AspectJ improves software evolvability as compared to Java. The first experiment was performed on 35 student subjects who were asked to comprehend either Java or AspectJ implementation of the same system, and perform the corresponding comprehension tasks. On average, the completion time was 29% longer for the AspectJ version. In the second experiment, 24 student subjects were asked to implement (in a non-invasive way) two extension scenarios to the system that they had already known. Each subject evolved either the Java version using Java or the AspectJ version using AspectJ. On average, the Java version required over twice as many atomic changes as the AspectJ version.

The rest of this paper is structured as follows. The next section motivates this work. Section 3 discusses considerations for experimental design and defines the GQM plan. Section 4 reviews related work on evaluation of maintainability of AO systems. Sections 5 and 6 describe our experimental designs. Section 7 presents the experimental results. Section 8 discusses threats to validity. Concluding remarks are given in Section 9.

2 Motivations

Changes to software are inevitable. Software systems change during their lifecycle to meet the growing needs and changing requirements of users. According to Lehman and Belady (1976), the functional content of a system must be continually increased or the system becomes progressively less useful. Moreover, as a system evolves its modularity decreases unless extra work is done to improve it. The modules become excessively loaded with responsibilities, while their interfaces become polluted. This breakage in modularization lowers adaptability to new requirements and increases the maintenance cost (Ponisio 2006).

The influence of AOP on software evolvability is unclear. On the one hand, replacing code that is scattered across many modules with a single aspect simplifies the localization of the relevant code during maintenance and potentially reduces the number of changes (Mortensen 2009; Tonella and Ceccato 2005). In addition, if a module does not tangle code from different concerns, it is also potentially easier to evolve.

On the other hand, constructs such as pointcuts and advices can make the ripple effects in AO systems far more difficult to control than in object-oriented (OO) systems. Mainstream AO languages (e.g. AspectJ, AspectC++) rely on referencing structural properties of the program such as naming conventions and package structure. These structural properties are used by pointcuts to define intended conceptual properties about the program. Thus, maintenance changes that conflict with the assumptions made by pointcuts introduce defects (Mortensen 2009; Tourwé et al. 2003). This phenomenon is called the pointcut fragility problem (Koppen

and Störzer 2004). It occurs when a pointcut unintentionally captures or misses a given join point as a consequence of seemingly safe modifications to the base code (Kellens et al. 2006; Koppen and Störzer 2004). For instance, adding a new class may seem to be safe, but it inevitably produces new join points that can be unintentionally intercepted. Kästner et al. (2007) reported such silent changes during AO refactoring. Kellens et al. (2006) addressed the fragile pointcut problem by declaring pointcuts in terms of a conceptual model of the base program, rather than defining them directly in terms of how the base program is structured. Hence, they transformed the fragile pointcut problem into the problem of keeping a conceptual model of the program synchronized with that program, when the program evolves. Nevertheless, their proposal has not got through to the mainstream, because it was implemented as an extension of CARMA, which is a barely known AO language for Smalltalk.

Moreover, the quantification and obliviousness properties of AO languages, which are the very advantage offered by AOP, turned out to be a source of difficulties for reasoning about the behavior of AO programs (Griswold et al. 2006; Munoz et al. 2008). Since aspects can modify data structures or control flow in non-local places, they undermine a programmer's ability to reason locally about the behavior of the module (Dantas and Walker 2006). To exacerbate this, since not all dependencies between the modules in AO systems are explicit, a maintainer has to invest more effort to get a mental model of the program (Storey et al. 1999). He has to "weave" all the modules that implement a given concern into a coherent unit in his mind. Creating a mental model is crucial to comprehend a concern before attempting to modify it (Mancoridis et al. 1998). Studies of software maintainers have shown that 30% to 50% of their time is spent on program comprehension (Fjeldstad and Hamlen 1983; Standish 1984).

Furthermore, we demonstrated (Przybylek 2011) that the new kinds of dependencies introduced by the AO constructs contribute to high coupling. Highly coupled systems have long been recognized as hard to comprehend (Lieberherr and Holland 1989; Page-Jones 1980) because their modules cannot be considered in separation from each other.

In addition, Hanenberg and Unland (2001) and Lopez-Herrejon et al. (2006) found that step-wise development is not satisfactorily supported by AspectJ. In particular, concrete aspects cannot be extended, while advices and concrete pointcuts cannot be overridden. Hanenberg and Unland (2001) proposed four rules of thumb which allow one to build reusable and incrementally modifiable aspects. However, enormous complexity is the price that has to be paid for it. Several other works showed that ensuring a sound combination of aspects is a challenging and difficult task. Kniesel et al. (2001) discussed weaknesses of AOP with regard to independent extensibility. They found that there are no satisfactory means to safely combine aspects that have been developed independently. In the general case, when aspects have not been specifically designed for joint use and do not have intimate knowledge of their respective implementation details, unwanted side effects may occur due to unexpected semantic interactions between the aspects. McEachen and Alexander (2005) pointed out the problems resulting from the unanticipated composition of aspects and base classes that arises when foreign aspects are rewoven. An aspect is foreign if it is woven into a class with the resultant bytecode being imported by another party which does not have access to the aspect source code. They concluded that proper handling of unanticipated composition requires language extensions that convey additional information to the aspect weaver. Furthermore, Arnaoudova et al. (2008) demonstrated that AspectJ semantics is not intuitive in several cases.

To summarize, the question about the impact of AOP on software evolvability remains open, and before AOP receives more acceptance in the industry, software engineers need further evidence of the real benefits as well as the costs.

3 Considerations for Experimental Design and our GQM Model

3.1 Considerations for Experimental Design

A controlled experiment can be conducted with between-subjects design or within-subjects design (Easterbrook et al. 2008). A within-subjects design is an experiment in which each subject is exposed to more than one level of an independent variable. In a between-subjects design, subjects are randomly assigned to groups and then each group is exposed to one and only one level of each independent variable.

Several issues must be considered before choosing the appropriate experimental design. On the one hand, there are two main advantages of a within-subjects design (Bordens and Abbott 2011): (1) most subject-related factors are literally identical across treatments, so they do not obscure the effect of the independent variable; and (2) a relatively smaller number of subjects is required in the experiment. On the other hand, a fundamental disadvantage of this research design is the problem of "carryover effects". Carryover effects occur when a previous treatment alters the behavior observed in a subsequent treatment (Bordens and Abbott 2011). Furthermore, within-subject designs are also more prone to "demand effects". A demand effect occurs when participants in experiments interpret the researcher's intentions and change their behavior (either consciously or not) to satisfy the researcher (Charnessa et al. 2011).

The most problematic carryover effects with regard to experiments on program comprehension are learning effects. The subjects learn about the program each time they comprehend it. Thus, the result of comprehending the second implementation might be better than the first, simply because the subjects know more about the program and not because the implementation is easier (Juristo and Moreno 2001; Kitchenham et al. 2002). In our comprehension experiment, it would be easy for subjects to remember the findings. Therefore, we chose a between-subjects design to avoid learning effect that otherwise would dominate the effect we want to measure.

In a true controlled experiments the researcher has to have control over all possible extraneous variables. In some cases it may be impossible or undesirable to apply experimental rigor so far as fully controlled settings, or random assignment. Providing the compromises and limitations are stated, understood, and taken into account in all conclusions and interpretations, such studies are referred to as quasi-experiments (Mauch and Park 2003).

Both between-subjects and within-subjects designs require that two or more groups of subjects are exposed to the various treatments. Then, the differences among the groups are tested statistically to determine whether these differences may be attributed to the effect of the independent variable (Bordens and Abbott 2011). However, to detect the effect, if any, it is required to have a sufficiently large number of subjects. Since it is very expensive and difficult to control a large number of subjects for a long period of time, the length of a controlled experiment is often limited to a few hours, which rules out realism in the experimental tasks (Harrison 2000; Fenton, 2001). To study more realistic situations, many researchers forego rigorous experimental designs and act as both experimenters and experimental subjects. This approach can be considered as a special case of the single-subject experiment (Bordens and Abbott 2011; Harrison 2000), where single subject refers to the participant or cluster of participants (e.g., a team) under investigation. Unfortunately, self-experimentation is often a weak example favoring the proposed technology over alternatives (Zelkowitz and Wallace 1998).

Many studies in software engineering rely on students as subjects. The reason for using students is that they are easily available and willing to participate in studies as part of courses they attend (Höst et al. 2000). If assessment is being performed on a new technology, experimenting with

students is often the only reasonable possibility. Nevertheless, students are not necessarily representative of professional programmers (Harrison 2000; Murphy et al. 1999). Professional programmers perform better than students if a technology used in the experiment is one that they use in their daily work (Hanenberg and Endrikat 2013). The advantage of working with students is that their prior knowledge is fairly homogeneous. Moreover, it would be extremely difficult to come by an adequate pool of professional programmers who are experienced in AspectJ and organize them into a set of experimental subjects.

3.2 Goal, Questions, and Metrics

We followed the Goal-Question-Metric (GQM) approach (Basili et al. 1994) to define a measurement system for our research. An overview of the goal, questions, and metrics as well as their interrelations is provided in Fig. 1. Our purpose is to compare Java and AspectJ (the most prominent AO language) with respect to software evolvability.

Comparing the effort required to evolve the software system using two different programming paradigms is a challenging task. If the programming experience of the subjects in one paradigm is much higher than in the other, the experience effect can dominate the main factor we want to observe. This is the typical situation when a new technique is compared to an existing one (Juristo and Moreno 2001). To mitigate the influence of the experience effect, we decomposed evolvability into understandability, which is not so prone to this effect, and changeability, and then we measured each sub-characteristics in two separate controlled experiments. The factor in each experiment was the programming language being used, which is a categorical variable with two values, Java and AspectJ. In the changeability experiment, subjects evolved a system that they had already known. In this way, we isolated the changeability effort from the understandability effort. Furthermore, the system incorporated two change scenarios implemented during the laboratory sessions. Thus, subjects had hints on how to use programming constructs to integrate new features with the existing code.

Following Bartsch and Harrison (2008), we consider a software system to be understandable if: (1) its output can be determined correctly by following the application's control flow; and (2) the identification of the output can be carried out in a short time. In turn, changeability refers to the ease with which a change can be applied to a software system. We consider a

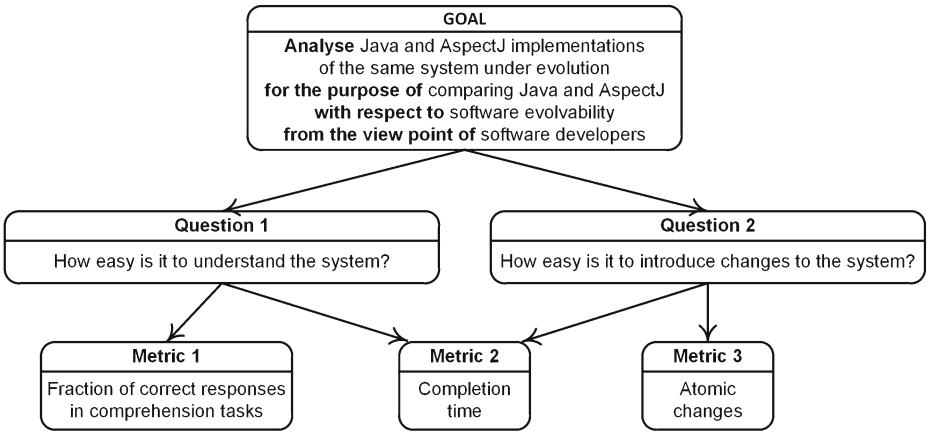

Fig. 1 GQM measurement plan

software system less changeable if: (1) it takes a long time to implement a new requirement; and (2) this implementation requires a high volume of changes made to the source code.

Our measurement model conforms to maintenance models that have been proposed so far. For instance, Boehm's model (1987) consists of three major phases: understanding, modifying, and revalidating the software. Generally, when a system is evolved, it follows a re-engineering process that encompasses reverse engineering and forward engineering (Chikofsky and Cross 1992). Reverse engineering is the process of analyzing system modules and their relationships in order to create a mental model of the source code (Margaret-Anne et al. 1999). Forward engineering is the traditional process of moving from high-level designs to the physical implementation of a system (Chikofsky and Cross 1992).

4 Related Work

4.1 Measuring Maintainability of AO Software

Although the term "software evolution" appeared in the software engineering literature in 1976 when Lehman and Belady (1976) started to formulate the laws of software evolution, it has not been given a widely accepted definition yet. We prefer the view that software evolution occurs when either reductive or enhancive maintenance (according to Chapin's classification (2001)) is carried out. These types of maintenance encompass activities that reduce, replace, add, or extend the customer-experienced functionality (Chapin et al. 2001) and can be equated to software enhancements in terms of the ISO/IEC 14764 standard (2006). Note that some authors of related work (Bartsch and Harrison 2008; Kulesza et al. 2006) use the term "software maintenance" even though they only perform enhancement maintenance, so, in fact, they evolve the system.

Analogically to how maintainability is perceived by the ISO/IEC 9126 standard (2001), evolvability can be considered as both internal and external quality attributes. Correspondingly, two groups of approaches for measuring it in the context of AO systems have been proposed. The first group uses internal product metrics to predict the level of effort required for evolving the software. Before AOP came into being, coupling and cohesion were key principles when comparing design alternatives. The dogma was that good design should exhibit high cohesion and low coupling (Coad and Yourdon 1991; Meyer 1989; Yourdon and Constantine 1979) and numerous empirical studies confirmed that improvements in coupling and cohesion are linked to improved maintainability (Briand et al. 1999, 2001; Hitz and Montazeri 1995). However, when we want to compare maintainability between OO and AO software, lexical SoC must be considered as a third dimension of modularity. Accordingly, Sant'Anna et al. (2003) defined three concern diffusion metrics to measure lexical SoC. They also adapted the Chidamber-Kemerer metrics suite (Chidamber and Kemerer 1994) to be applicable to AO software. Next, they developed a framework to provide support for assessment of reusability and maintainability of AO systems. Their framework includes measures of size, coupling, cohesion, and lexical SoC. Then, a number of studies (Benestad et al. 2006; Greenwood et al. 2007; Hoffman and Eugster 2009; Katić et al. 2013; Kulesza et al. 2006; Lobato et al. 2008; Mguni and Ayalew 2013; Shen et al. 2008) used those metrics as predictors of the maintenance effort. Unfortunately, using Sant'Anna's framework to compare evolvability between OO and AO software is problematic due to several reasons. First, the framework does not explain how to combine multiple metrics in order to provide a single view and allow the ranking of the examined implementations.

Second, there is no consensus on how to aggregate the measurements from the micro-level of individual classes to the macro-level of the entire software system (Chatzigeorgiou and Stiakakis 2013; Serebrenik and van den Brand 2010). Third, as demonstrated by Przybylek (2011, 2013), the coupling metric used in the framework does not cover all kinds of coupling dependencies in AO software and thus favors AOP.

The second group of approaches for measuring software evolvability uses external metrics that are focused on how much effort is required to evolve the system. Hanenberg et al. (2009), Tonella and Ceccato (2005), Walker et al. (1999), and Hanenberg and Endrikat (2013) measured this effort as the time required by subjects to implement a change scenario. The problem with this approach is that it does not take into account the quality of the solution. Well modularized code is usually more challenging, so it requires more time to develop. Moreover, subjects can often work faster and less carefully, reducing time, but also increasing the number of errors. Because of this tradeoff, some researchers (e.g. Bartsch and Harrison (2008)) measured both completion time and correctness of the solution. In turn, Hanenberg and Endrikat (2013), and Hanenberg et al. (2009) delivered test cases to subjects and accepted the corresponding task as done only after all tests passed successfully.

In turn, Sant'Anna et al. (2003), Kvale et al. (2005), Kulesza et al. (2006), Greenwood et al. (2007), Figueiredo et al. (2008), Lobato et al. (2008), Mortensen et al. (2012), and Mguni and Ayalew (2013) measured evolution effort by counting the number of source code modifications when going from one version to the next. The granularity of the change set differed among studies from the coarser file or module level to the finer operation or statement level. A fair balance among different granularity levels was obtained by Shen et al. (2008) who adapted the approach introduced by Ryder and Tip (2001). At the core of this approach is the ability to transform source code edits into a collection of atomic changes, which captures the semantic differences between two successive releases of a program. Examples of such changes are adding an empty module, adding an empty method/attribute, changing the body of a method, etc.

4.2 Experiments to Evaluate Maintainability of AO Software with External Product Metrics

Several studies have been conducted to evaluate the impact of AOP on software maintainability. In general, findings from these studies are quite diverse and thus inconclusive. Studies closely related to our work (these that use external product metrics) are summarized in Tables 1 and 2 and discussed further in this section (note that some of them additionally use internal product metrics). They can be classified into two broad groups according to the dependent variable under study. Studies in the first group measure the time spent to perform maintenance tasks, while studies in the second group investigate the volume of changes made to the source code. Note that all studies in the first group are controlled or quasi-controlled experiments, while all studies in the second group are single subject experiments, and so do not employ statistical inference. Although controlled experiments are considered more reliable because they reduce internal validity threats to drawing scientifically valid inferences from the results, their experimental objects often do not reflect the complexity observed in the industrial context. Indeed, except the experiment by Tonella and Ceccato (2005), who actually did not compare Java with AspectJ, but Java with a subset of AspectJ in a specific context, all programs under study in the controlled or quasi-controlled experiments are representatives of small systems (see Table 2). Similarly, we also used a small system, because we were constrained by the availability of the subjects, which was limited to a few hours. Surprisingly, among the single subject experiments only 2 out of 7 used notably larger (over 50 KLOC) systems.

Table 1 Related studies investigating software maintainability

Authors	Experimental method	Experimental subjects	Dependent variables	Succeeded paradigm
Walker et al. (1999)	Quasi-experiment, between	5 Students and 1 Professional	Completion time	OOP?
Tonella and Ceccato (2005)	CE, between	12 Students	Completion time (understanding)	AOP
			Completion time (total)	AOP?
Bartsch and Harrison (2008)	Quasi-experiment, between	11 Professionals	Completion time	OOP?
			Structural changes	OOP?
Hanenberg et al. (2009)	CE, within	20 Students	Completion time	OOP
Hanenberg and Endrikat (2013)	CE, within	15 Students	Completion time	OOP
Sant'Anna et al. (2003)	SSE	1 Team	Structural changes	AOP
Kvale et al. (2005)	SSE	1 Team	Structural changes	AOP
Kulesza et al. (2006) Greenwood et al. (2007)	SSE	1 Team	Structural changes	AOP
Figueiredo et al. (2008)	SSE	1 Team	Structural changes	–
Lobato et al. (2008)	SSE	1 Team	Structural changes	AOP
Mortensen et al. (2012)	SSE	1 Team	Structural changes	AOP
Mguni and Ayalew (2013)	SSE	1 Team	Structural changes	AOP

CE controlled experiment; *between* between-subject design; *within* within-subject design; *SSE* single-subject experiment;? results are not statistically significant

The other challenge of controlled experiments is to achieve realism regarding experimental tasks. Our change scenarios are realistic for small systems and relatively big when compared to maintenance tasks carried out in the prior controlled experiments (see Table 2). The average time to accomplish our comprehension task was 49 min for the Java version and 63 min for the AspectJ version, while the average time of implementing change scenarios using Java and AspectJ was 84 min and 93 min, respectively. In turn, maintenance tasks carried out by Figueiredo et al. (2008),

Table 2 Related studies – additional information

Authors	Experimental objects	Time*		Granularity level of the change set
		OO	AO	
Walker et al. (1999)	A distributed digital library, 1.5 kloc	95	136	N/A
Tonella and Ceccato (2005)	java.awt package (jdk 1.4), 36 kloc	54	39	N/A
Bartsch and Harrison (2008)	An online shopping system, 0.5 KLOC	49	60	LOC
Hanenberg et al. (2009)	A game, 9 classes	124	183	N/A
Hanenberg and Endrikat (2013)	A library system, 26 classes	88	167	N/A
Sant'Anna et al. (2003)	Portalware, 1 KLOC	N/A	N/A	LOC; operations; modules
Kvale et al. (2005)	Java Email Server, 1.8 KLOC	N/A	N/A	LOC; modules
Kulesza et al. (2006) Greenwood et al. (2007)	Health watcher, 5.5 KLOC	N/A	N/A	LOC; modules
Figueiredo et al. (2008)	MobileMedia, 3 KLOC	N/A	N/A	LOC; operations; modules
Lobato et al. (2008)	MobiGrid framework, 0.7 KLOC	N/A	N/A	operations; relationships; modules
Mortensen et al. (2012)	InstanceDrivers, 1.6 KLOC; PowerAnalyzer, 13.9 KLOC; ErcChecker, 51.6 KLOC	N/A	N/A	LOC; modules; files
Mguni and Ayalew (2013)	Jasperreports, 137 KLOC; OpenBravoPOS, 53 KLOC	N/A	N/A	LOC; operations; modules

* the average time of executing all experimental tasks (in minutes)

Kulesza et al. (2006), Kvale et al. (2005), and Mortensen et al. (2012) seem to be bigger than any maintenance tasks carried out in a controlled setting. What is interesting, change scenarios involved by Mguni and Ayalew (2013), who evolved the biggest systems, were smaller than our change scenarios regarding the number of lines of code required to implement the change.

Although we leveraged experimental practices elaborated by others (mainly by Bartsch and Harrison (2008), Hanenberg and Endrikat (2013)), our work differs from previous studies in several respects. First of all, to reduce the experience effect, we divided evolvability into two sub-characteristics, understandability and changeability, and then evaluated each separately adjusting experimental settings. Moreover, contrary to Hanenberg and Endrikat (2013), we think that within-subjects experiments in software maintenance are flawed due to learning effects, and so we employed a between-subjects design. In turn, to reduce the problem of error variance, we used many more subjects than the related between-subjects experiments. Indeed, Bartsch and Harrison (2008) did not obtain statistically significant results, while Walker et al. (1999) did not conduct hypothesis testing probably due to the limited number of subjects.

Furthermore, the studies in the second group used multiple metrics, each of which measures different kinds of change (i.e. element added, modified or deleted) at different levels of granularity (e.g. modules, operations, LOC). This approach makes it difficult or impossible to determine the overall difference in evolution effort between the AO versus the OO code. For instance, compared to its AspectJ counterpart, the Java implementation of MobileMedia (Figueiredo et al. 2008) required 34% fewer modules to be added, but 20% more modules to be modified as it evolved through all change scenarios. It also required fewer LOC to be added and modified, but more LOC to be deleted (see Table 3). Thus, we can not say which paradigm performed better. To deal with these issues, our metric combines different kinds of change into a single view. In addition, our metric differentiates the significance of a particular change (e.g. multiple changes inside a method body are counted the same as a single change to a method signature), so it more accurately reflects the true evolution effort.

Finally, in contrast to the previous work, we provide what is called a "reproducible package" (Torkar et al. 2017) to foster replication of our experiments. The package, which is available at http://przybylek.wzr.pl/ESE/, includes the experimental materials, the raw data, the processed dataset, and R scripts to plot data, test hypotheses, etc.

Walker et al. (1999) conducted one of the first experiments to understand how the separation of concerns provided by AOP affects a programmer's ability to accomplish system modification tasks. A multi-threaded, distributed system (a digital library) was used in the experiment. The system had two implementations; one was written in AspectJ and another in Emerald (an OO language). Six participants (5 students and 1 professional) were allocated to treatment groups based on their backgrounds (e.g. participants with previous knowledge of Emerald were assigned to the Emerald group). Then, the participants individually performed three change tasks using either AspectJ or Emerald. Among some other measurements, the time needed to complete the tasks was measured. The AspectJ participants were slower at the

Table 3 Measures of changeability in MobileMedia (derived from the original study)

Modules						Operations						LOC					
Added		Deleted		Changed		Added		Deleted		Changed		Added		Deleted		Changed	
OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO
45	68	9	9	66	55	284	360	105	100	109	145	3268	3513	1251	1111	176	382

change tasks, but statistical inference was not performed due to the lack of random assignment and the small size of the groups.

Tonella and Ceccato (2005) focused on a specific kind of cross-cutting concerns, the scattered implementation of methods declared by interfaces that do not belong to the principal decomposition. They called such interfaces aspectizable, and migrated them to aspects in a number of classes from the JDK 1.4 java.awt package (36 KLOC). Next, they designed a controlled experiment to measure the maintenance and comprehension effort required to complete two maintenance tasks. The independent variables were the source code being used (either Java or AspectJ) and the task being executed. In both settings, execution of Task 1 required the modification of one file (2 lines of code), while Task 2 required changing seven files (18 lines of code). The dependent variables were the maintenance time (total time measured during the execution of a maintenance task) and the understanding time (time measured while the programmer is comprehending the code structure or is determining the code fragments to be changed). It turned out that aspectization of the aspectizable interface implementations resulted in a significantly lower understanding time, while the differences in the overall maintenance time were not statistically significant at the 5% level.

Bartsch and Harrison (2008) conducted a quasi-experiment in which 11 subjects were asked to comprehend either Java or AspectJ version of an online shopping system (about 0.5 KLOC) and carry out a maintenance task. They identified the following response variables to measure maintainability: (1) the amount of time required to identify all modules in the system; (2) the percentage of correctly identified modules; (3) the amount of time required to identify the system output; (4) subjects' opinions of software understandability; (5) the number of LOC changed in order to implement a new requirement; and (6) the amount of time required to implement the requirement. Since the subjects stayed at their home while participating in the experiment, the researchers did not have control over extraneous variables. All experimental materials and a questionnaire, which consisted of questions reflecting the response variables, were sent via an email. The subjects were also asked to mail back their solutions. The result of the experiment was that for none of the measurements a significant difference between the two paradigms at the 10% level was found. Nevertheless, in the context of our research it is still worth looking at the mean values for their third response variable, because we measured the same variable using a similar approach. The average completion time for their AO group was 31% longer than for the OO group, while in our experiment, this difference amounts to 29%.

Hanenberg et al. (2009) found that the overall development time for cross-cutting concerns using Java was shorter than the development time for such concerns using AspectJ. Nevertheless, for two tasks (logging and null-pointer-checks) subjects using AspectJ performed better. Their study was performed on 20 students as a within-subject experiment. As a target application, they used a game consisting of 9 classes. Using this application, nine tasks needed to be completed, each one in Java as well as AspectJ whereby the order of the programming language was randomly chosen. The programming tasks represented cross-cutting concerns that were well suited to being implemented in AspectJ. Hanenberg and Endrikat (2013) extended the research in new settings. They developed a simple Java application which represented a library system (26 classes). Next, they defined 9 programming tasks to conduct a new within-subject experiment. The tasks were designed in a way that subjects needed to perform changes on aspect code that they previously wrote on their own. The AspectJ-based solutions required significant more time than the Java solutions for 5 tasks. For the other 4 tasks there was no statistically significant difference. The experiment was performed with 15 student subjects who attanded an AOSD master course.

Sant'Anna et al. (2003) conducted a quasi-controlled experiment to compare the use of AspectJ and Java to implement Portalware (about 60 modules and over 1 KLOC). Portalware is a multi-agent system (MAS) that supports the development and management of Internet portals. The experiment team (3 PhD candidates and 1 MSc student) developed two versions of the Portalware system: an AO version and an OO version. Next, the same team simulated seven maintenance/reuse scenarios that are recurrent in large-scale MAS. For each scenario, the difficulty of maintainability was defined in terms of structural changes to the artifacts in the AO and OO systems, such as number of modules added or changed, number of added or changed LOC, and so forth. The total lines of code, that were added or copied to perform the maintenance tasks, equaled 534 for the OO approach and 472 for the AO approach. The OO approach required 7 modules and 6 LOC to be modified, while the AO approach needed 3 modules and 10 LOC.

Kvale et al. (2005) compared AspectJ and Java in COTS-based development. They refactored Java Email Server (1.8 KLOC) to AspectJ and then performed three maintenance tasks on both versions. The number of LOC and modules that required to be changed (added, modified or deleted) in the OO implementations were 235 and 15 respectively. The corresponding values for the AO implementations were 221 and 3.

Kulesza et al. (2006) evolved AspectJ and Java implementations of a health complaint system, called Health Watcher (over 5.5 KLOC). The purpose of the system was to improve the quality of the services provided by health care institutions. Some of the cross-cutting concerns were refactored to aspect from the first release, while others were refactored as new versions were released. In the maintenance phase of their study, they introduced a set of 8 new use cases. It was found that more modules were needed to be modified in the AspectJ version. Surprisingly, Greenwood et al. (2007), who refer to the same study, report that 111 modules needed to be changed in the AspectJ implementation, and 143 modules in the Java implementation. In turn, both papers report that more line of codes needed to be added in the Java implementation.

Figueiredo et al. (2008) assessed the impact of AspectJ on a number of changes applied to MobileMedia (3 KLOC). MobileMedia is a software product line for applications that manipulate photo, music, and video on mobile devices. The original release was available in both AspectJ and Java (the Java version used conditional compilation as the variability mechanism). Then, a group of five post-graduate students was responsible for implementing 7 successive evolution scenarios. Each new release was created by modifying the previous release of the respective version. The scenarios comprised of different types of changes involving mandatory, optional, and alternative features, as well as non-functional concerns. Table 3 presents summarized results over all change scenarios. Besides, it was found that AspectJ usually did not cope with the introduction of mandatory features. Moreover, depending on the evolution scenario, AspectJ pointcuts were more fragile than conditional compilation.

Lobato et al. (2008) applied 4 heterogeneous evolutionary changes to the MobiGrid framework (0.7 KLOC). MobiGrid is a mobile agent system, which is employed to encapsulate and execute long processing tasks using the idle cycles of a network of personal workstations. MobiGrid was originally implemented in Java, then refactored to AspectJ and evolved in both versions by the researchers. Both paradigms gave the same results with respect to the number of added/changed/removed components. The main difference was the number of added operations that equaled to 10 for the AO implementation, and 17 for the OO counterpart.

Mortensen et al. (2012) adopted AOP in three proprietary VLSI CAD programs: InstanceDrivers (1.6 KLOC), PowerAnalyzer (13.9 KLOC), and ErcChecker (51.6 KLOC). They refactored 5 cross-cutting concerns as aspects in the first program, 8 in the second, and 7 in the third. Because the programs were originally implemented in C++, they used AspectC++.

Then, they followed the evolution of the applications across several revisions for both paradigms. They used 6 revisions in InstanceDrivers, 7 in PowerAnalyzer, and 3 in ErcChecker. The AO revisions required 5.8%, 4.9%, and 2.3% fewer line changes in InstanceDrivers, PowerAnalyzer, and ErcChecker, respectively. They also resulted in a 11.7%, 5.3%, and 1.6% reduction in the number of modules changed.

Mguni and Ayalew (2013) studied the maintainability of two COTS-based systems originally implemented in Java. The first system was a reporting tool – Jasperreports (137 KLOC). The second was OpenBravoPOS (53 KLOC), which provides functionalities for retail business. Mguni & Ayalew refactored both systems using AspectJ. Session management, logging, and exceptional handling were moved into aspects in OpenBravoPOS, while synchronization, object retrieval, and exceptional handling were aspectized in Jasperreports. They also defined two maintenance tasks. Maintenance task I referred to the introduction of a new feature to evaluate the execution time of SQL statements. Maintenance task II referred to the replacement of the logging component with another component of the same functionality. Next, they carried out maintenance task I on both systems and maintenance task II on OpenBravoPOS. In the OO implementation of OpenBravoPOS, the change affected 12 operations and 9 modules, while in the AO implementation, the change affected 2 operations and 2 module. For the Jasperreports OO implementation, the change affected 10 operations in 2 modules, while in the AO implementation, the change affected only 1 operation in 1 module.

4.3 Other Studies

Coady and Kiczales (2003) compared the evolution of two versions (C and AspectC) of four crosscutting concerns in FreeBSD. First, they refactored the crosscutting concerns (page daemon activation, prefetching for mapped files, quotas for disk usage, and tracing blocked processes in device drivers) into aspects. These implementations were then rolled forward into their subsequent incarnations. In each case the AspectC implementation better facilitated independent development and localized change.

Kästner et al. (2007) evaluated the suitability of AspectJ to implement features in a software product line. They used AspectJ to refactor the embedded database system Berkeley DB into 38 features. The resulting feature code was hard to understand and maintain due to strong and implicit coupling and fragile pointcuts.

Munoz et al. (2008) showed that aspects offer efficient mechanisms to implement crosscutting concerns, but that aspects can also introduce complex errors in case of evolution. To illustrate these issues, they implemented and then evolved a chat application. They found that it is very hard to reason about the aspects impact on the final application.

Kouskouras et al. (2008) built an emulator of a telecommunications exchange, allowing the user to configure it with commands and to emulate simple calls between subscribers. They developed three different implementation alternatives. The first one followed a simplistic solution applying OOP. The second made use of the Registry pattern. The third applied AspectJ to implement the Registry pattern. Next, they investigated the behavior of the designs at a specific extension scenario. The extension scenario involved the addition of several new commands and parameters. Since they made the source code available for us, we could apply our metric. The differences in Atomic Changes between various versions were less than 3%. Each version turned out to be very extensible and reusable.

Katić et al. (2013) explored the educational benefits of introducing AOP into a programming course. They separated 75 students into two groups: AO group (experimental group) and OO group (control group). The AOP group was given a lecture and a tutorial demonstrating AOP's theoretical and technical topics. Both groups had the same laboratory assignment, with the only difference being that the first group had to solve it using C#, while the second group had to use PostSharp. On average, subjects in the AOP group required around 15% fewer lines of code to implement the assignment.

Pereira et al. (2017) proposed a framework (AOF4OOP) to support semi-transparent schema evolution for object-oriented databases. The framework uses AspectJ to modularize the persistence concern of affected application. As a result, the class structure of the application can be freely changed in its source code, being then incrementally reflected into the database. To evaluate AOF4OOP, they developed a proof of concept application in two versions. One version used AOF4OOP as a persistence layer, while the other used DB4O, which is an embeddable object database for Java. Then, they defined a change scenario that required updating the class structure of the application. It turned out that, the evolution of the AOF4OOP-based version required fewer lines of code and modules to be added.

5 Experimental Design for Software Understandability

5.1 Objects

A flagship AspectJ application called Telecom was chosen to be used in this experiment. Telecom is part of the AspectJ distribution (www.eclipse.org/aspectj/). It is a telephony system simulator in which customers make, accept, merge and hang-up calls. In addition to these basic operations, there is a timing concern which measures the total connection time per customer. There is also a billing concern that charges customers for the calls they make, according to the amount of time they used and the types of calls they made. Following maintenance tasks proposed by Burrows et al. (2011), we also introduced another concern, which is responsible for storing information about historical connections for each customer. We independently built a Java and AspectJ implementations for each extension concerns. The details are provided in the subsections below. According to the open-closed principle (Meyer 1989), which states that "software entities should be open for extension but closed for modification", at each stage we tried to reuse the existing code using composition mechanisms and avoid invasive modifications.

Telecom was chosen because (1) it was originally developed to demonstrate the power of AOP; (2) it includes extension scenarios; (3) it is not overly complex and it can be comprehended in less than 2 h; and (4) it has been used in several other experiments (e.g. Burrows et al. 2011; Przybylek 2013; Rinard et al. 2004; Santos et al. 2016). Rinard et al. (2004) implemented a tool that automatically classifies interactions between aspects and methods. Then, they used Telecom to evaluate this tool. Burrows et al. (2011) analyzed how faults are introduced during maintenance tasks in AspectJ programs. They planned 3 maintenance tasks in Telecom. We took inspiration from these tasks when we defined our change scenarios. Przybylek (2013) used Telecom to evaluate his approach that reduces coupling in AspectJ programs and restores modular reasoning in case when the aspects are spectators. Santos et al. (2016) used Telecom to analyze code pitfalls that are likely to lead programmers to make mistakes in AspectJ refactoring.

5.1.1 Initial Release

Note that there are no aspects in the initial release, so Fig. 2 presents the share design for both languages. Calls are initiated by a caller (the customer). The Connection class models the physical details of establishing a connection between caller and receiver. If the caller and receiver have the same area code then the call is established with a Local connection, otherwise a LongDistance connection is required. Initially, only the connection between the caller and receiver exists but additional connections may be added if calls are merged to form conference calls. Moreover, a customer may be involved in many calls at one time.

5.1.2 Initial Release + Timing

In the AspectJ release, the Timing aspect declares that each Connection object has a timer. A Timer object records the current time when it is started and stopped, and returns the difference when asked for the elapsed time. A mechanism to start the timer when a connection is completed and to stop the timer when the connection is dropped, is implemented in two after advices. The Timing aspect also declares an inter-type field totalConnectTime for Customer to store the total connection time. This field is updated by the second advice.

In the Java release, the total connection time is stored in a subclass of Connection. To measure how long a connection lasts, we could add startTime and stopTime attributes to the Connection class. However, to avoid invasive modifications, we have to duplicate the same attributes in two new subclasses that extend Local and LongDistance respectively. Moreover, we have to extend Call to adjust the constructor (see its implementation in Fig. 2) to the new types. What is worse, we cannot reuse the constructor from Call, but we have to re-write almost the same in the subclass.

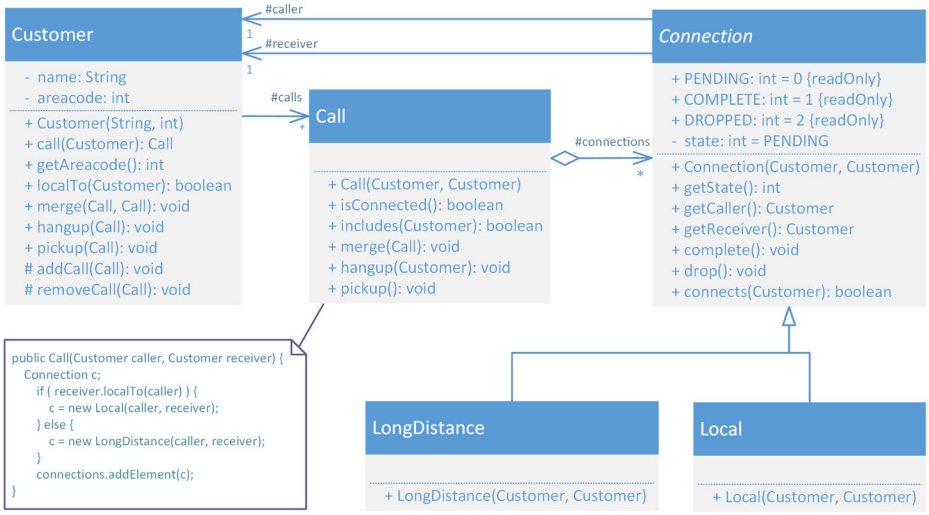

Fig. 2 Initial architecture of Telecom (class diagrams illustrating the successive releases are available at http://przybylek.wzr.pl/ESE/)

5.1.3 Initial Release + Timing + Billing

The Billing aspect uses the timer that is associated with a connection to calculate a charge per connection and to add that charge to the appropriate customer's bill. In detail, Billing declares the inter-type methods callRate() for Local and LongDistance so that different types of connection can be charged accordingly. The charge is calculated after the timer is stopped; this is handled by after advice on the Timing::endTiming pointcut and relevant precedence declaration. Finally, Billing declares inter-type methods and attributes for Customer to handle the amount of money that customers are charged.

In the Java implementation, the charge is kept in a subclass of Customer, while the call rates are kept in subclasses of Connection. Once again, we need to extend Call to adjust the constructor to the new types. Note that the previous subclass of Call is useless at this stage.

5.1.4 Initial Release + Timing + Billing + Listing

The Listing aspect declares that each Customer has inter-types to store and list its connection history. Connections are put into a linked list by the after advice on the Timing::endTiming pointcut.

Applying OO techniques, new properties to store and list historical connections are defined in a subclass of Customer, while the add(Connection) method to put connections into the underlying linked list is invoked from subclasses of Connection. Once again, we need to extend Call to substitute instances of Customer, Local and LongDistance by instances of the newly derived classes.

5.2 Subjects

Subjects in the comprehension experiment were 35 students from the OOSD course held in the 4th year of study in Computer Science. We managed to combine the learning objective of the course with the research objective of the study. Before taking the course, all subjects had wide experience with Java, since they had used Java as a primary language for their studies. The course devoted 4 lectures (4 h) and 4 labs to AOP. The students were also assigned homework (4 h estimated) that required them to use AspectJ. Although the participation in the experiment was voluntary, almost all students who were enrolled in the course decided to participate, because the successful completion of the experiment replaced the final exam.

5.3 Variables and Hypotheses

We identified the following response variables: (1) the time needed to comprehend the system; and (2) the comprehension accuracy. The former was measured by the time (in minutes) taken by subjects to complete comprehension tasks. The latter was the fraction of correct responses. Two groups of hypotheses, one for each dependent variable were derived as follows:

$H0_1$: The average comprehension time for the Java implementation is the same as the average comprehension time for the AspectJ implementation;
Ha_1: The average comprehension time for the Java implementation differs from the average comprehension time for the AspectJ implementation;
$H0_2$: Programmers who comprehend the Java implementation have the same distribution of comprehension accuracy as programmers who comprehend the AspectJ implementation;

Ha_2: The distribution of comprehension accuracy differs between the two groups of programmers.

We were not sure about the effect of AspectJ on the variables, so we used two-sided tests. Furthermore, since we did not expect the distribution of comprehension accuracy to be normal, we formulated corresponding non-parametric hypotheses.

5.4 Experiment Execution

First, the subjects were introduced to the overall format of the experiment. Then, they were randomly assigned to two groups (G_OO, G_AO). Group G_OO had 18 members, and group G_AO had 17 members. Next, each subject received either Java (9 pages) or AspectJ source code (6 pages) of Telecom printed on paper, according to his/her group. Following the recommendation by Hanenberg and Endrikat (2013) we decided to not provide any programming environment, because we wanted to have all external factors under control and equal for both languages.

Along with the source code, each participant received the output of the program. However, the order of the output lines was changed (Fig. 3). Moreover, results of six method calls were replaced in the output with blank spaces. For each output line, the subjects had to write its actual order according to the control-flow, while each gap had to be filled in. These tasks require deep comprehension of the system under study. Subjects can achieve a maximum of 9 points. One point is awarded for each correctly filled gap. In turn, the order of the output lines is regarded as one response. Thus, one point is awarded if all lines are correctly numbered. Figure 3 shows the correct answers in highlighted text. The most tricky part of the output is Line 17, because Crista's bill has not been charged even though she has been connected for a few seconds. Indeed, this Line accounts for most of the errors made by subjects regardless of the group. We made the materials available at http://przybylek.wzr.pl/ESE/ for other researchers who may be interested in replicating or further extending the experiment.

1	jim calls mik...
7	jim calls crista...
3	mik accepts...
9	crista accepts...
5	jim hangs up...
11	crista hangs up...
4	connection from Jim(650) to Mik(650) completed
6	connection from Jim(650) to Mik(650) dropped
10	connection from Jim(650) to Crista(415) completed
12	connection from Jim(650) to Crista(415) dropped
2	[new local connection from Jim(650) to Mik(650)]
8	[new long distance connection from Jim(650) to Crista(415)]
13	Jim(650) has been connected for ..6.. seconds and has a bill of ..32..
15	Crista(415) has been connected for ..2.. seconds and has a bill of ..0..
14	Detailed connection: Receiver time cost Mik(650) ..4.. ..12.. Crista(415) ..2.. ..20..
16	Detailed connection: Receiver time cost

Fig. 3 Experimental materials: the output of the program

The students were instructed that the shorter the completion time, the more bonus points are given, but to get any points the solution must be 100% correct. They were also given a limit of two hours, but they always completed the tasks much earlier.

5.5 Analysis Procedure

Figure 4 describes the procedure used for analyzing the data collected in the comprehension experiment. In all statistical tests, we considered a significance level of 0.05, i.e. we decided to accept a 5% probability of committing a type I error. Note that statistical hypothesis testing can only detect the presence of a significant difference, but it does not provide any information about the difference. Therefore, in case there is a significant difference, we also report a corresponding confidence interval describing that difference. We employed the R environment for statistical computing.

6 Experimental Design for Software Changeability

6.1 Objects and Subjects

Once again we used Telecom with the Timing and Billing concerns as the target application for this evaluation.

Subjects in the changeability experiment were 24 students with the same characteristics as those in the comprehension experiment, but enrolled in the next run of the OOSD course. However, we used different training examples. In particular, the participants were trained on Telecom. During the laboratory sessions, they had to implement the Timing and Billing

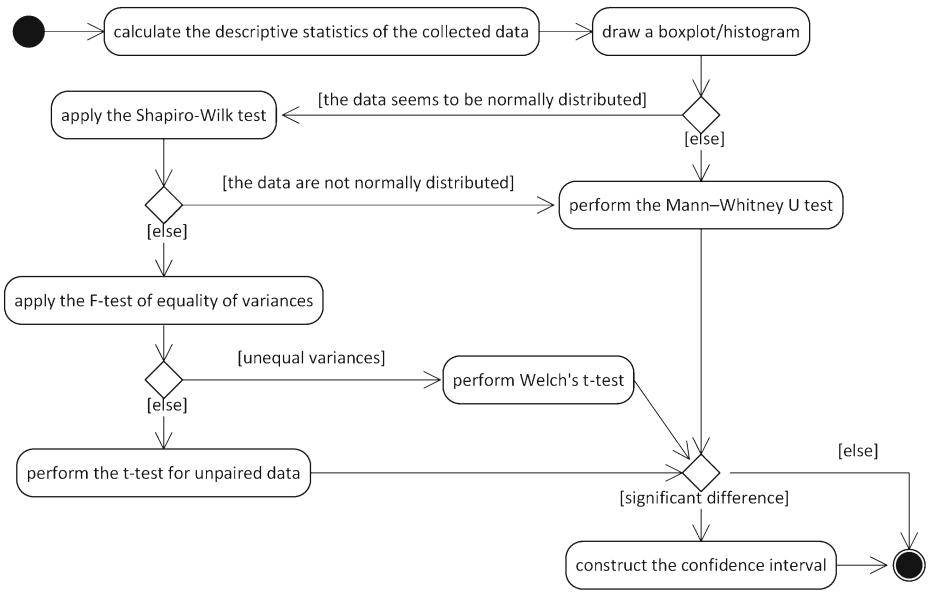

Fig. 4 Analysis procedure

concerns first using Java and then using AspectJ. We also discussed and shared with the participants our reference implementations.

6.2 Variables and Hypotheses

The response variables used in the changeability experiment were: (1) the time (in minutes) that the subjects spent to implement the change scenarios; and (2) the volume of changes made to the source code. The second variable was measured by the number of atomic changes that captured the semantic differences between two successive releases of a system. Our set of atomic changes was as follows: add an empty module, delete a module, add an empty method, change body of method, delete a method, add/delete an attribute, change an attribute initializer, add an empty advice, change an advice body, delete an advice, add an empty pointcut, change a pointcut body, delete a pointcut, introduce an empty method, change an introduced method body, delete an introduced method, introduce an attribute, delete an introduced attribute, change an introduced attribute initializer, add/delete a soften exception declaration, add/delete an aspect precedence, add/delete a hierarchy declaration, change an access level modifier. This set of atomic changes was adapted from Shen et al. (2008). Note that we took into account changes to access modifiers and we considered the deletion of a whole non-empty element as one atomic change.

Two sets of hypotheses analyzed in the experiment were as follows:

$H0_1$: The distribution of time required to implement the change scenarios does not differ between Java and AspectJ programmers;
Ha_1: The distribution of time required to implement the change scenarios differs between Java and AspectJ programmers;
$H0_2$: The distribution of atomic changes does not differ between Java and AspectJ programmers;
Ha_2: AspectJ programmers need fewer atomic changes to implement the change scenarios than Java programmers do need.

As for atomic changes, we used a one-sided hypothesis test, because before the experiment we implemented the change scenarios ourselves and we needed significantly more atomic changes for our Java implementations than for our AspectJ implementations. In turn, we did not have a hypothesis about the direction of the effect of AspectJ on the completion time, so we used a two-sided test.

6.3 Experiment Execution

The participants were informed about the experimental procedure and grading criteria (existing modules cannot be modified; for each correctly implemented scenario students obtain 7 points; in addition they may obtain up to 4 points for code quality, e.g. adherence to the information hiding principle, reusing business logic; and up to 2 points for the completion time). The aim of the grading criteria was to unify the coding style. The experiment time was limited to 130 min. The development environment used by the participants was Eclipse with the AJDT plugin. The participants were not allowed to use any resources, except the AspectJ 5 Quick Reference that was provided to them in a printed version.

Then, the participants were randomly assigned to two groups (G_OO, G_AO). Each group had 12 members. Next, the participants were asked to download and import an Eclipse project appropriate for their group. The project contained the Java or AspectJ implementation of Telecom (with Timing and Billing) - the same that we had shared with the students during the laboratory sessions. The project also included a couple of JUnit tests for both scenarios to be implemented, which participants could execute to detect implementation defects. Therefore participants were obliged to adhere to two method headers and one field declaration that we specified in the scenario definitions (detailed instructions are provided at our website).

As we mentioned earlier, before a programmer starts to modify/extend a system, he has to comprehend it. To isolate the effort associated with source code modifications from the effort associated with source code comprehension, the participants were asked to evolve the system that they had already known, i.e. Telecom. For the purpose of the experiment, on the top of Telecom we defined 2 new concerns, namely "Friend" and "Free Seconds" that subjects had to implement independently in two scenarios. The Friend concern gives each customer the possibility to set one other customer from the same area code as a friend. Then, a customer may call his friend for free. The following is an example AspectJ implementation of the Friend concern:

```
public aspect Friend {
        declare precedence: Friend, Billing;
        public static final long FRIEND_RATE = 0;
        private Customer Customer.friend;
        public void Customer.setFriend(Customer f) {
                if(localTo(f)) friend=f;
        }
        public boolean Customer.hasFriend(Customer customer) {
                return friend==customer;
        }
        public long Local.callRate() {
                if(this.caller.hasFriend(this.receiver))
                        return FRIEND_RATE;
                else
                        return Billing.LOCAL_RATE;
        }
}
```

The FreeSeconds concern makes that customers earn one free second for every two seconds of an inbound call from a different area code. Then, the collected seconds are used to make free outbound calls to non-friend customers from the same area code.

After all participants imported the project, the description of the first scenario was distributed in a printed version and the time was started. In order to complete a task, participants were required to pass all corresponding test cases. However, the test cases did not check whether the participants used the right programming paradigm or adhered to the open-closed principle. This check was performed manually after the experiment and the participants had been aware about it. When a participant finished the task the time was noted, the solution was copied into a pendrive and the participant was given the description of the second scenario. During the experiment, we were in the laboratory to prevent collaboration among participants, and to oversee compliance with the experimental procedure.

7 Experimental Results

7.1 The Comprehension Experiment

7.1.1 Measurements

Table 4 presents the collected data. Only one subject in the G_OO group (#18) and two in the G_AO group (#29, #32) failed to identify the correct order of the output lines. The remaining errors come from incorrectly filled blank spaces. The data reveals that the *i*-th best subject from the G_OO group always required less time to accomplish the experiment than the *i*-th best subject from the G_AO group. We can also observe that the five fastest subjects in each group did not commit any error.

7.1.2 Completion Time

We start by providing an overview of the results in Table 5 and Fig. 5. The descriptive statistics and the box plot suggest that generally subjects working on the Java implementation required less time to accomplish the comprehension task. However, to find out whether the difference is significant, it is necessary to apply an appropriate statistical test.

Since the distributions of completion time for both groups can be assumed (1) to be normal (the Shapiro–Wilk test gives a p-value of 0.72 for the G_OO group and a p-value of 0.84 for the G_AO group); and (2) to have the same variance (the F-test of equality of variances gives a p-value of 0.42), we perform an Independent Sample T-test to compare the means. Because the obtained p-value is less than the significance level (p-value = 0.0004 < 0.05 = α), we reject the null hypothesis and conclude that on average the G_AO group required significantly more time than did the G_OO group. The two-sided 95% confidence interval for the difference in average completion times between the G_AO and G_OO group is between 6.3 min and

Table 4 Results for individual subjects by group

G_OO			G_AO		
Subject	Time	Comprehension accuracy	Subject	Time	Comprehension accuracy
1	23	1.0	19	43	1.0
2	38.8	1.0	20	47	1.0
3	39	1.0	21	51	1.0
4	41.5	1.0	22	52.5	1.0
5	43.7	1.0	23	55.5	1.0
6	43.7	0.9	24	57.5	0.7
7	44.5	1.0	25	57.7	1.0
8	49.3	0.9	26	60.8	1.0
9	50	0.6	27	61	0.3
10	50.2	0.7	28	61.2	0.6
11	51	0.9	29	65.5	0.6
12	52.8	0.7	30	71.3	1.0
13	53	0.9	31	72.7	1.0
14	56	1.0	32	73	0.6
15	57.8	1.0	33	76.3	1.0
16	59.5	1.0	34	78	1.0
17	61.5	1.0	35	91.7	1.0
18	67.7	0.4			

Table 5 Descriptive statistics for completion time (in minutes)

Group	Min	Max	arith. Mean	Median	std. dev.
G_OO	23.0	67.7	49.1	50.1	10.3
G_AO	43.0	91.7	63.3	61.0	12.6

22.1 min. Thus, we are 95% confident that the AspectJ implementation requires, on average, between 6.3 and 22.1 min more to be effectively understood than the Java implementation.

7.1.3 Comprehension Accuracy

A summary of the descriptive statistics of comprehension accuracy is shown in Table 6. The results indicate that most of the subjects in both groups did not have major difficulties in understanding the system. The medians are the same, while the means are almost the same regardless of the paradigm. Figure 6 gives a more intuitive representation of the data. The box plot reveals that the distributions of scores for both groups are similar and heavily skewed to the left. Accordingly, we perform a non-parametric Mann-Whitney U test to verify whether the slight differences in scores between the G_OO and G_AO group could have been obtained by chance. The resulting *p*-value is 0.7, which indicates that equality of the groups cannot be rejected (we fail to reject the null hypothesis).

7.2 The Changeability Experiment

7.2.1 Measurements

Table 7 presents the collected data. Each row describes a single subject. For each change scenario, we show the amount of time (in minutes) the subjects spent implementing it (Columns 3 and 4) and the corresponding number of atomic changes (Columns 6 and 7). Notice, that we are not going to compare the times taken to implement the second scenario. Instead, we are going to analyze the total time spent in the execution of the experiment (Column 5). We expected all subjects to provide correct solutions. However, the first scenario was failed by one subject (#12) in the G_OO group and was implemented with a bug by two subjects (#20 and #22) in the G_OA group, while the second scenario was failed by 5 subjects in the G_AO group and 4 subjects in the G_OO group. Notice, that an artificial completion time of 130 min was used for subjects (#12 and #20) in order to get a fair comparison between the treatments. We used two-sided Fisher's exact test to investigate whether Java and AspectJ programmers are equally likely to complete the change scenario. For both change scenarios, the calculated *p*-values were equal to 1, thus, regarding this issue, the equality of the groups

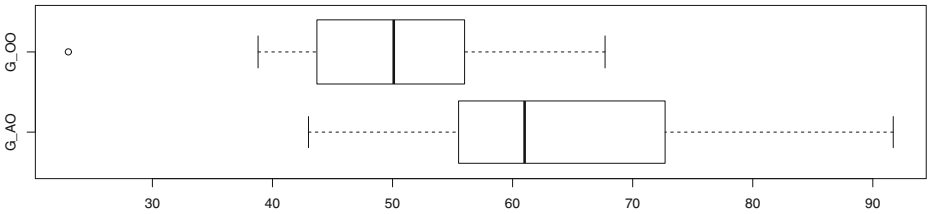

Fig. 5 Box plot of completion time by group

Table 6 Descriptive statistics for comprehension accuracy

Group	Min	Max	arith. Mean	Median	std. dev.
G_OO	0.43	1.00	0.88	1.00	0.17
G_AO	0.29	1.00	0.87	1.00	0.23

cannot be rejected. Furthermore, the data reveals an interesting fact – for both scenarios the number of atomic changes was always much smaller in the G_AO group than in the G_OO group. We believe that the G_OO group would have performed much better with respect to completion time as well as number of atomic changes, if the participants had been allowed to alter existing modules. However, in such a case, using AspectJ would be irrelevant, because the initial classes could be refactored to avoid code scattering. Note, that the last two rows represent our reference solutions that we implemented before conducting the experiment.

7.2.2 Completion Time

Table 8 and the box-plots in Fig. 7 summarize the time spent on scenario I (*time1*) and the total time spent on both scenarios across groups. Concerning the mean and median values the subjects from the G_OO group needed less time to implement the changes, but the differences are not so much. Furthermore, the box-plots reveal that the distribution of *time1* for the G_OO group is right-skewed, while the distribution of *totalTime* for both groups are left-skewed. Since the data violates the normality assumption, we perform Mann-Whitney U tests to verify whether the distributions of *time1* and *totalTime* statistically differ between Java and AspectJ programmers. The obtained *p*-values are 0.49 for *time1* and 0.38 for *totalTime*. Thereby, we fail to reject the null hypothesis and we cannot claim that there is a significant difference between Java and AspectJ programmers when it comes to the time they require to implement the change scenarios.

7.2.3 Atomic Changes

Table 9 and the box-plots in Fig. 8 summarize the number of atomic changes required by each group to implement each change scenario. They confirm the observation that AspectJ programmers needed much fewer atomic changes than Java programmers. Thereby, calculating *p*-values is merely a formality. Since the data cannot be assumed normally distributed (as can be seen in Fig. 9), we use Mann-Whitney U test. According to our expectation, the resulting *p*-values (0.00002 for *AC1* and 0.0007 for *AC2*) are much smaller than the significance level (0.05), so we calculate corresponding confidence intervals. As for *AC1*, the one-sided 95% confidence interval is $(-\infty, -9]$ and we conclude that a typical Java programmer needs at least 9 more atomic

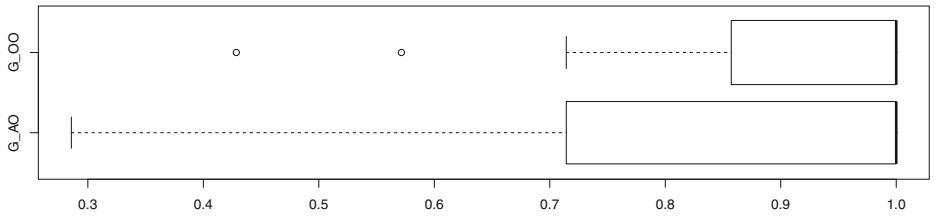

Fig. 6 Box plot of comprehension accuracy by group

Table 7 Results for individual subjects

Subject	Group	Time1	Time2	Total time	AC1	AC2
1	G_OO	20.7	25.3	45.9	22	26
2	G_OO	22.3	30.3	52.7	18	20
3	G_OO	47.0	30.7	77.7	18	20
4	G_OO	43.7	42.3	86.0	28	40
5	G_OO	34.8	52.5	87.3	20	28
6	G_OO	51.0	54.7	105.7	18	24
7	G_OO	40.3	65.7	106.0	20	26
8	G_OO	32.8	73.8	106.7	22	41
9	G_OO	79.0	–	–	22	–
10	G_OO	85.0	–	–	21	–
11	G_OO	107.5	–	–	27	–
12	G_OO	*130.0*	–	–	–	–
13	G_AO	13.8	33.3	47.2	9	10
14	G_AO	37.2	13.9	51.1	10	9
15	G_AO	34.2	27.9	62.1	10	6
16	G_AO	58.8	46.3	105.0	11	12
17	G_AO	40.3	69.7	110.0	11	8
18	G_AO	64.7	50.5	115.2	9	5
19	G_AO	84.0	41.2	125.2	9	6
20	G_AO	58.7	–	*130.0*	9	–
21	G_AO	59.2	–	–	10	–
22	G_AO	91.5	–	–	12	–
23	G_AO	118.0	–	–	10	–
24	G_AO	125.0	–	–	10	–
we	*G_OO*	*N/A*	*N/A*	*N/A*	*20*	*24*
we	*G_AO*	*N/A*	*N/A*	*N/A*	*11*	*10*

changes to implement the change scenario I than a typical AspectJ programmer. Concerning *AC2*, the one-sided 95% confidence interval is $(-\infty, -14]$ and the conclusion is analogous.

7.2.4 Qualitative Analysis

Generally, the quality of the solutions provided by the subjects was worse than the quality of our reference solutions. Regardless of the group, the fundamental problem was that most subjects re-implemented the business logic from previous versions instead of reusing it. Moreover, some subjects did not adhere to the information hiding principle. In particular, they directly accessed attributes that were defined in other modules. Paradoxically, this bad practice resulted in lower values of the AC metric (indeed, an implementation of a single accessor method requires two atomic changes). Nevertheless, no subject provided better solution than we did, while top subjects provided a solution similar to ours.

Furthermore, regardless of the programming paradigm being used, differences in coding style affected the number of atomic changes required to implement a given concern. For instance, some subjects defined FRIEND_RATE = 0 as a static final attribute, while others

Table 8 Descriptive statistics for completion times (in minutes)

	Time1					Total time				
Group	Min	Max	arith. Mean	Median	std. dev.	Min	Max	arith. Mean	Median	std. dev.
G_OO	20.7	130.0	57.8	45.4	34.8	45.9	106.7	83.5	86.7	23.8
G_AO	13.8	125.0	65.5	59.0	33.8	47.2	130	93.2	107.5	34.1

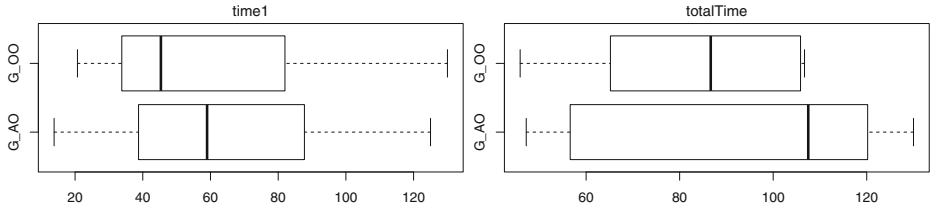

Fig. 7 Box plots of completion times (in minutes) by group

hardcoded the rate value within methods. Besides, some subjects preferred to create a large, complex method/advice rather than break it down into a few smaller methods.

Change Scenario I The main idea behind our reference AspectJ solution is to override the callRate() inter-type declaration for the Local class that was introduced by the Billing aspect. The new implementation of callRate() checks whether the callee is a friend of the caller and then returns an appropriate rate. Subject #17 proposed a solution that is the same as ours and requires the same number of atomic changes even though his implementations is slightly different. Subjects #14, #23, and #24 also proposed an elegant solution in which calls to the callRate() method are captured and if the callee is a friend of the caller, FRIEND_RATE is returned. Nevertheless, their solution seems to be a bit more complex than ours and requires one more atomic change. In turn, the implementation by subject #16 is ugly, but it is still based on the same idea as ours. Subjects #13, #15, #18, #19, and #21 decided to charge the caller twice but with the negative cost at the second time. The downside of their solution is that it was implemented by cloning the after advice from Billing and multiplying the calculated cost by minus 1, thus a part of the business logic is scattered through two aspects. Note that their solution usually was implemented with two atomic changes fewer than ours because it did not explicitly use FRIEND_RATE. Finally, subjects #20 and #22 provided incorrect implementations even though their implementations passed the test cases. Nevertheless, their implementations could be easily fixed without changing the number of atomic changes, thus we did not exclude them from our estimation sample.

As for our reference Java solution, we still followed the same strategy as in previous scenarios. Particularly, we defined a new subclass of BillingTimingLocal that overrides getCost() to make it “friend-aware”. Subject #3 implemented the scenario in a similar way and with the same number of atomic changes as we did. Subject #1 passed over BillingTimingLocal at all and defined a new subclass of TimingLocal. Then he overrode the drop() method to make it “friend-aware”. The problem with the new subclass is that it duplicates much of the business logic from BillingTimingLocal. Subjects #2 and #7 defined a new subclass of BillingTimingLocal and overrode drop() in this way that they charge the caller twice but with the negative cost at the second time. Subjects #6 and #10 introduced a

Table 9 Descriptive statistics for atomic changes

	AC1					AC2				
Group	Min	Max	arith. Mean	Median	std. dev.	Min	Max	arith. Mean	Median	std. dev.
G_OO	18	28	21.45	21	3.39	20	41	28.12	26	8.15
G_AO	9	12	10	10	0.95	5	12	8	8	2.52

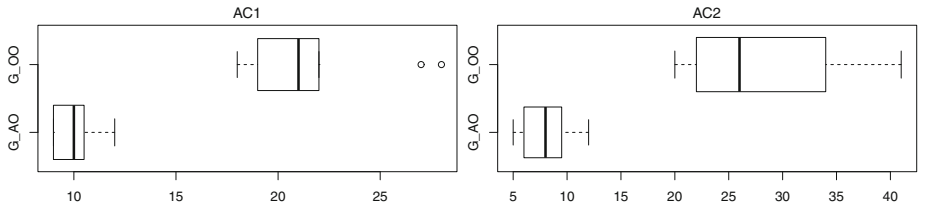

Fig. 8 Box plots of atomic changes by group

new subclass of TimingLocal with a very simple implementation, i.e. its overridden getCost() method just returns zero all the time. Thus, they use the new subclass just for addressing connections to friends and still need two other subclasses of TimingLocal for other connections. A similar solution but with a bug in its implementations was provided by subjects #8 and #9. They also overrode getCost() to return zero. However, their implementations charges the caller twice, because they unnecessarily overrode drop(). Although the addCharge() method is called twice, the final result is correct, because the argument of that method is zero. Finally, subject #4 provided a very complex implementation with five new classes.

Change Scenario II In our reference AspectJ solution, we defined a new aspect that introduces an inter-type field freeSeconds for Customer and two inter-type methods that updates this field. In addition, we defined an after advice on the Timing::endTiming() pointcut that, when appropriate, stores and consumes free seconds by means of the inter-type methods. The advice also calls addCharge() with a negative value to "cancel" the cost that was exchanged for free seconds. Finally, we declared a precedence to ensure that the new advice is executed after the advice defined in Timing. A similar solution was chosen by all subjects who accomplished this stage. However, the quality of their code is worse than ours, even though they usually needed less atomic changes. Particularly, subjects #18, #19, and #13 implemented a large complex advice that accesses and updates the freeSeconds field introduced to Customer. Besides, subjects #15 and #18 have a bug, even though their programs run correctly. They did not specify the order of advice execution between the new aspect and Timing. By default the order is undefined, while in our scenario it is essential that the Timing feature is executed first.

As for the reference Java solution, all inter-type declarations from our AO solution were implemented as members of a new subclass of Customer. However, the main part of our reference OO solution are two new subclasses for handling Local and LongDistance connections. The former overrides getCost() to consume free seconds from the caller when appropriate, while the latter overrides drop() to add free seconds to the callee. Subjects #2, #3, #4, #5, and #7 provided similar implementations as we did, but a bit more complex. Moreover, subjects #2 and #3 as well as #6 did not implement helper methods that updates the

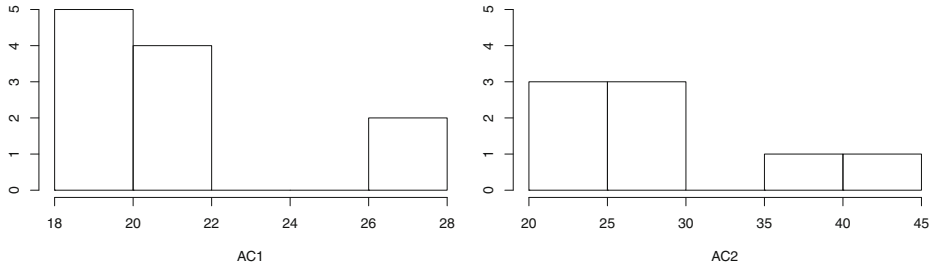

Fig. 9 Histograms for atomic changes for the G_OO group

freeSeconds field in Customer, but they accessed this field directly from other classes. In this way they saved a few atomic changes even though the quality of their source code is inferior. The worst implementations were provided by subjects #1 and #8, who re-implemented much of the business logic from previous scenarios.

8 Threats to Validity

It is necessary for the researcher to figure out the appropriate balance of construct validity, internal validity, and external validity. Achieving high levels for all of these factors may not be possible for new technologies (Murphy et al. 1999). Furthermore, it is impossible to conduct controlled experiments with a reasonable budget at the scales that are seen in production environments. Accordingly, in our study we trade external validity for internal validity.

8.1 Construct Validity

Evolvability, like other quality factors, is difficult to measure. The difficulty is even greater if we have to compare the evolvability of two alternatives of a given system implemented in different programming languages. Although several GQM models supporting such a comparison have been elaborated, we proposed a new one that is suitable in the case when subjects are less experienced in one of the language. The rationale for our approach and the choice of metrics were explained. Nevertheless, we are aware that our metrics do not capture all dimensions of evolvability and other metrics could return different results.

8.2 Internal Validity

A major challenge to internal validity is the difference in experience in the treatments (Java vs. AspectJ). This problem concerns all experiments that compare a new technology with the existing, well-established one. To mitigate this threat we employed appropriate experimental designs and provided an extensive training program along with a motivational system encouraging students to learn AOP.

Another threat is the variability in participants' skill. We minimized this threat by both choosing participants who had a common educational background, and randomizing the assignment of participants to the treatments.

The other important threat is related to the extent to which the implementation alternatives used in the study constitute a "good" OO or AO design. We chose to use a system that was implemented to demonstrate the strengths of AspectJ. Thus, we believe that it represents a good usage of AOP. We also made our best effort to develop high quality corresponding Java code. However, other programmers could choose different strategies for implementing the system as well as change scenarios.

Another issue concerns the influence of the IDE. While IDEs play a relevant role for most programming tasks, their influence on different programming paradigms is unknown (Hanenberg and Endrikat 2013). This issue is especially relevant for program comprehension, since advanced search tools and outlines of the program structure can significantly speed up an understanding of the program behavior. Therefore, we eliminated the impact of tool support in our comprehension experiment by using the paper versions of the program. However, we used Eclipse together with

the AJDT plugin to implement the change scenarios, because working without an intelligent code completion would favor the programming language that is better known.

The last threat to the internal validity comes from the fact that our experiments involved multiple participants in the same room. In such a case, participants might "follow the crowd" and finish early simply because other participants are leaving.

8.3 External Validity

The most important threat to external validity concerns the representativeness of the experimental object. Since Telecom was originally implemented by professionals to demonstrate the power of the new paradigm, it is a representative for the usage of AspectJ. However, the size of the system, regarding number of modules and lines of code, does not reflect the complexity observed in the industrial software systems.

Another potential threat is related to the representativeness of the subjects. It is quite common for experiments in software engineering to use students as subjects, because professionals are not easily available. Similarly, in our experiments, resorting to students was the only pragmatic possibility. Although professional programmers perform better than students, both groups are similar in their approach to developing software (Hanenberg and Endrikat 2013). Höst et al. (2000) suggested that results achieved with students are transferable and can provide valuable insights into an analyzed problem domain. In turn, Scholtz and Wiedenbeck (1992) demonstrated that experienced programmers exhibit a drop in performance and their solution process is disrupted when using an unfamiliar programming language. Thus, in our experiments, using professionals would be biased, since they would probably be much more experienced with Java, but would not have prior knowledge of AspectJ.

9 Conclusions

In today's dynamic business environment, frequent changes to information systems are inevitable and impose a number of evolution tasks. Accordingly, software fulfilling business needs is not only difficult to develop, but it is even harder to evolve. Thereby, software engineers search for methods and techniques to support the evolution of software systems. During the 2000s decade, notable expectations were put on AOP. However, although AOP has gained substantial attention from the scientific community, it has not managed to become widely adopted by industry. We believe that one of the reasons for this is the accumulation of misleading catchwords over unbiased discussions of both the strengths and limitations of AOP.

In this paper, we report on an empirical evaluation of the impact of AspectJ on software evolvability. So far, there has been no consensus on this issue. Our first experiment was performed on 35 subjects who were asked to comprehend either Java or AspectJ implementation of the same system and perform the corresponding comprehension tasks. The average completion time was 29% longer for the AO group than for the OO group and the difference was statistically significant. However, quite surprisingly, we did not find evidence that AspectJ had an effect on comprehension accuracy. Most participants in both groups completed the tasks correctly. We can suppose that a majority of the AO group just concluded the weaving order from the output without a deep analysis of the source code. Thereby, it would be interesting to see how the participants would behave if we had made this order invalid. We plan to investigate this issue in future work. Note, that in the comprehension experiment, we

did not allow for using IDE because our intention was not to measure the impact of IDEs, but the impact of language features. Nevertheless, our approach can be considered artificial. In the production environment, developers use various IDEs to navigate the complex dependencies between program entities. As for AspectJ programs, an appropriate IDE support (e.g. AJDT with the Cross References view in Eclipse) can help a programmer identify aspects that augment the studied class and rebuild the global picture of how the aspects interact with the base code. Thereby, in future work, we intend to replicate this experiment in different settings, i.e. we want subjects to use Eclipse when comprehending a system.

Our second experiment was performed on 24 subjects who were asked to implement two change scenarios for the system that they had already known (in this way we isolated the changeability effort from the understandability effort). Each subject had to evolve in a non-invasive way either the AspectJ version using AspectJ or the Java version using Java. Although subjects who evolved the AspectJ version generally required more time, the difference was not statistically significant. Besides, we found out that a typical AspectJ programmer needs significantly fewer atomic changes to implement the change scenarios than a typical Java programmer. Although comparable findings have been obtained in numerous single-subject experiments (Greenwood et al. 2007; Kvale et al. 2005; Lobato et al. 2008; Mguni and Ayalew 2013; Mortensen et al. 2012; Sant'Anna et al. 2003), to the best of our knowledge, this is the first statistically significant result regarding structural changes published so far. One could think that the greater number of atomic changes needed to implement a new requirement is not an issue as long as the time is the same. However, non-invasive changes usually result in non-optimal design decisions, which in turn contribute to design erosion. Indeed, as for OOP, we observed a paradox of the open-closed principle. When deriving a class through inheritance, every place in the code where the base class is instantiated usually should be re-written to instantiate the subclass. Nonetheless, if we want to follow the open-closed principle, we cannot just modify the existing code, we have to extend relevant classes and override those methods that instantiated the base class. Accordingly, even if we want to add a single method, we often have to derive a few subclasses. However, a cascade of changes is exactly what the open-closed principle seeks to avoid. Moreover, successive extensions lead to deep inheritance hierarchies that somewhat degrade the system design. These findings raise new research questions, such as which paradigm is more robust on the accumulation of non-optimal design decisions, or how difficult is it to refactor Java and AspectJ systems that have undergone a lot of change scenarios.

A key novel insight of our research is that AspectJ differently affects on two sub-characteristics of evolvability: understandability and changeability. While AspectJ decreases the former, it improves one aspect of the latter. This has several implications for research and practice. First of all, this explains why previous experiments that measured the completion time usually showed the advantage of OOP over AOP, while most experiments focused on structural changes found positive impact of AOP. Furthermore, our results present a new perspective on the dispute between those who claim that AspectJ improves SoC and those who have the opposite viewpoint. Aspects allow us to separate the implementation of crosscutting concerns from the implementation of core concerns at the lexical level. Thereby, we can evolve existing concerns independently regarding the lexical level or easily plug in new concerns. As a consequence, using AspectJ we will usually need fewer structural changes to evolve our system. However, we must be aware that the lexical separation does not imply a semantic decoupling. A class augmented by aspects in general cannot be meaningfully studied in isolation. We need to undertake additional effort to "weave" the aspects into the class in our minds. Moreover, by looking at a class we even do not know what aspects augment it. To determine the applicability of aspects, we require global knowledge, i.e. all aspects need to be revised. Thus, we believe that the number of aspects that are manageable within a

program is limited and cannot grow together with the size of the program. Furthermore, when any name in a base program is changed, not only the aspects that have relied on this name must be changed, but also all aspects with unbound pointcuts must be reconsidered. Accordingly, we will usually need more time to comprehend an AO system than its OO counterpart. Finally, our study shows that switching from Java to AspectJ does not require vast expenditures of time and effort. This might be an encouraging factor for companies that would like to try to put AspectJ to work on real projects. Even though our study suggests that in general AspectJ does not decrease the overall time required to evolve a software system, there may be specific situations in which AspectJ performs better. For instance, Hohenstein and Jaeger (2011) reported a successful application of AspectJ in integrating 3rd party software, which saved development time and money. For those who want to replicate our experiments or conduct reanalysis of the collected data using other statistical methods, we provide the reproducible package, available at http://przybylek.wzr.pl/ESE/.

Acknowledgments The study is cofounded by the European Union from resources of the European Social Fund. Project PO KL "Information technologies: Research and their interdisciplinary applications", Agreement UDA-POKL.04.01.01-00-051/10-00.

References

Arnaoudova V, Eshkevari LM, Sharifabadi ES, Constantinides C (2008) Overcoming comprehension barriers in the AspectJ programming language. J Object Technol 7(6):121–142

Bartsch M, Harrison R (2008) An exploratory study of the effect of aspect-oriented programming on maintainability. Softw Qual J 16(1):23–44

Basili VR, Caldiera G, Rombach HD (1994) Goal Question Metric Approach. In: Encyclopedia of Software Engineering. Wiley, Chichester, pp 528–532

Benestad H, Anda B, Arisholm E (2006) Assessing Software Product Maintainability Based on Class-Level Structural Measures. In: International Conference on Product Focused Software Process Improvement, Amsterdam

Bezdek JC (1993) Fuzzy models - what are they, and why. IEEE Trans Fuzzy Syst 1(1):1–6

Boehm B (1987) Software engineering. IEEE Trans Comput 25(12):1226–1242

Bordens K, Abbott B (2011) Research design and methods: a process approach. McGraw-Hill, New York

Briand LC, Wüst J, Lounis H (1999) Using coupling measurement for impact analysis in object-oriented systems. In: IEEE Int'l Conf. On software maintenance (ICSM'99), Oxford

Briand LC, Wüst J, Lounis H (2001) Replicated case studies for investigating quality factors in object-oriented designs. Empir Softw Eng 6(1):11–58

Brito I, Moreira A (2004) Integrating the NFR framework in a RE model. In: 3rd workshop on early aspects at AOSD'04, Lancaster

Burrows R, Taïani F, Garcia A, Ferrari FC (2011) Reasoning about faults in aspect-oriented programs: a metrics-based evaluation. In: 19th international conference on program comprehension (ICPC'11), Kingston

Chapin N, Hale J, Khan K, Ramil J, Tan W-G (2001) Types of software evolution and software maintenance. J Softw Maint Evol Res Pract 13(1):3–30

Charnessa G, Gneezyb U, Kuhnc M (2011) Experimental methods: Between-subject and within-subject design. J Econ Behav Organ 81(1):1–8

Chatzigeorgiou A, Stiakakis E (2013) Combining metrics for software evolution assessment by means of data envelopment analysis. J Softw Evol and Proc 25(3):303–324

Chavez CH et al (2011) The AOSD research Community in Brazil and its crosscutting impact. In: 25th Brazilian symposium on software engineering, Sao Paulo

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493

Chikofsky EJ, Cross JH (1992) II. Reverse Engineering and Design Recovery: A Taxonomy. In: Arnold RS (ed) Software Reengineering. IEEE Computer Society Press, Washington DC, pp 54–58
Coad P, Yourdon E (1991) Object-oriented analysis. Prentice Hall, Upper Saddle River
Coady Y, Kiczales G (2003) Back to the future: a retroactive study of aspect evolution in operating system code. In: 2nd inter. Conf. On aspect-oriented software development (AOSD'03), Boston
Dantas DS, Walker D (2006) Harmless advice. In: conference record of the 33rd ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM, pp. 383–396, New York
Dijkstra EW (1976) A discipline of programming. Prentice Hall, Englewood Cliffs
Easterbrook SM, Singer J, Storey MA, Damian D (2008) Selecting empirical methods for software engineering research. In: Shull F, Singer J, Sjøberg D (eds) Guide to advanced empirical software engineering, pp. 285–311. Springer, Berlin
Fenton N (2001) Conducting and presenting empirical software engineering. Empirical Software Engineering 6(3):195–200
Figueiredo E et al (2008) Evolving software product lines with aspects: An empirical study on design stability. In: 30th Intl. Conf. on Software Engineering, Leipzig
Filman RE (2001) What is Aspect-Oriented Programming, revisited. In: Workshop on Multi-Dimensional Separation of Concerns at ECOOP'01, Budapest
Fjeldstad R, Hamlen W (1983) Application program maintenance-report to to our respondents. In: Parikh G, Zvegintzov N (eds) Tutorial on Software Maintenance. IEEE Computer Soc. Press, Washington DC, pp 13–27
Greenwood P, Bartolomei TT, Figueiredo E, Dósea M, Garcia AF, Cacho N, Sant'Anna C, Soares S, Borba P, Kulesza U, Rashid A (2007) On the impact of aspectual decompositions on design stability: an empirical study. In: 21st European conference on object-oriented programming (ECOOP'07), Berlin
Griswold WG, Sullivan K, Song Y, Shonle M, Tewari N, Cai Y, Rajan H (2006) Modular software design with crosscutting interfaces. IEEE Softw 23(1):51–60
Hanenberg S, Endrikat S (2013) Aspect-orientation is a rewarding investment into future code changes - as long as the aspects hardly change. Inf Softw Technol 55(4):722–740
Hanenberg S, Unland R (2001) Using and Reusing Aspects in AspectJ. In: Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA'01, Tampa Bay
Hanenberg S, Kleinschmager S, Josupeit-Walter M (2009) Does aspect-oriented programming increase the development speed for crosscutting code? An empirical study. In: 3rd International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista
Harrison W (2000) N=1: An Alternative for Software Engineering Research? In: BBS Workshop at ICSE'00, Limerick
Hitz M, Montazeri B (1995) Measuring Coupling and Cohesion in Object-Oriented Systems. In: 3rd International Symposium on Applied Corporate Computing, Monterrey
Hoffman K, Eugster P (2009) Cooperative aspect-oriented programming. Sci Comput Program 74(5–6):333–354
Hohenstein U, Jaeger MC (2011) Tackling the challenges of integrating 3rd party software using AspectJ. In: Katz S, Mezini M, Schwanninger C, Joosen W (eds) Transactions on aspect-oriented software development VIII. Lecture Notes in Computer Science, vol 6580. Springer, Heidelberg
Höst M, Regnell B, Wohlin C (2000) Using students as subjects. A comparative study of students and professionals in lead-time impact assessment. Empir Softw Eng 5(3):201–214
ISO/IEC 14764 (2006) Software engineering – software life cycle processes – maintenance. IEEE Std 14764-2006, Geneva
ISO/IEC 9126-1 (2001) Software engineering. Product quality. Part 1: quality model
Juristo N, Moreno AM (2001) Basics of software engineering experimentation. Springer, Berlin
Kästner C, Apel S, Batory D (2007) A case study implementing features using AspectJ. In: 11th international conference of software product line conference, Kyoto
Katić M, Botički I, Fertalj K (2013) Impact of aspect-oriented programming on the quality of novices' programs: a comparative study. J Info Org Sci 37(1):45–61
Kellens A, Mens K, Brichau J, Gybels K (2006) Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts. In: 20th European Conference on Object-Oriented Programming (ECOOP'06), Nantes
Kiczales G, Lamping J, Mendhekar A, Maeda C, Cristina Lopes C, Loingtier J, Irwin J (1997) Aspect-oriented programming. In: LNCS, vol 1241. Springer, Heidelberg, pp 220–242
Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG (2001) An Overview of AspectJ. In: 15th European Conference on Object-Oriented Programming (ECOOP'01), Budapest
Kienzle J, Guerraoui R (2002) AOP: Does It Make Sense? The Case of Concurrency and Failures. In: 16th European Conference on Object-Oriented Programming (ECOOP'02), Málaga
Kitchenham BA, Pfleeger SL, Pickard LM, Jones PW, Hoaglin DC, El-Emam K, Rosenberg J (2002) Preliminary guidelines for empirical research in software engineering. IEEE Trans Softw Eng 28(8):721–734

Kniesel G, Costanza P, Austermann M (2001) Independent Extensibility for Aspect-Oriented Systems. In: Workshop on Advanced Separation of Concerns at ECOOP'01, Budapest
Koppen C, Störzer M (2004) PCDiff: attacking the fragile pointcut problem. In: European interactive workshop on aspects in software, Berlin
Kouskouras KG, Chatzigeorgiou A, Stephanides G (2008) Facilitating software extension with design patterns and aspect-oriented programming. J Syst Softw 81(10):1725–1737
Kulesza U, Sant'Anna C, Garcia A, Coelho R, von Staa A, Lucena C (2006) Quantifying the effects of aspect-oriented programming: A maintenance study. In: 22nd IEEE Intl. Conf. on Software Maintenance, Dublin
Kvale AA, Li J, Conradi R (2005) A case study on building COTS-based system using aspect-oriented programming. In: 20th ACM symposium on Applied computing (SAC'05), Santa Fe
Lehman MM, Belady LA (1976) A model of large program development. IBM Syst J 15(3):225–252
Lieberherr K, Holland I (1989) Assuring good style for object-oriented programs. IEEE Softw 6:38–48
Lobato C, Garcia A, Kulesza U, von Staa A, Lucena C (2008) Evolving and composing frameworks with aspects: the MobiGrid case. In: 7th international conference on composition-based software systems, Madrid
Lopez-Herrejon R, Batory D, Lengauer CH (2006) A disciplined approach to aspect composition. In: ACM SIGPLAN 2006 workshop on partial evaluation and program manipulation (PEPM'06). In: Charleston
Mancoridis S, Mitchell BS, Rorres C, Chen Y, Gansner ER (1998) Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In: 6th international Workshop on Program Comprehension, Ischia
Margaret-Anne D, Storey F, Fracchia D, Muller HA (1999) Cognitive design elements to support the construction of a mental model during software exploration. J Softw Syst 44:171–185
Marot A (2011) Preserving the separation of concerns while composing aspects with reflective AOP. Phd thesis. Universite Libre De Bruxelles, October, p 2011
Mauch JE, Park N (2003) Guide to the successful thesis and dissertation, 5th edn. Marcel Dekker, Inc., New York
McEachen N, Alexander R (2005) Distributing classes with woven concerns: an exploration of potential fault scenarios. In: 4th international conference on aspect-oriented software development (AOSD'05), Chicago
Mens K, Tourwé T (2008) Evolution issues in aspect-oriented programming. In: Mens T, Demeyer S (eds) Software evolution, pp. 203–232. Springer, Heidelberg
Mens T, Mens K, Tourwé T (2004) Software evolution and aspect-oriented software development, a cross-fertilisation. ERCIM special issue on Automated Software Engineering, Vienna
Meyer B (1989) Object-oriented software construction. Prentice Hall, Upper Saddle River
Mguni K, Ayalew Y (2013) An Assessment of Maintainability of an Aspect-Oriented System. In: ISRN Software Engineering, vol 2013 pp 11. https://doi.org/10.1155/2013/121692
Mortensen M (2009) Improving software maintainability through Aspectualization. PhD thesis, Department of Computer Science, Colorado State University, Co
Mortensen M, Ghosh S, Bieman J (2012) Aspect-oriented refactoring of legacy applications: an evaluation. IEEE Trans Softw Eng 38(1):118–140
Munoz F, Baudry B, Barais O (2008) Improving maintenance in AOP through an interaction specification framework. In: IEEE Intl. Conf. On software maintenance, Beijing
Murphy GC, Walker RJ, Banlassad ELA (1999) Evaluating emerging software development technologies: lessons learned from assessing aspect-oriented programming. IEEE Trans Softw Eng 25(4):438–455
Ossher H, Tarr P (2001) Hyper/J: multi-dimensional separation of concerns for java. In: 23rd international conference on software engineering (ICSE'01), Toronto
Page-Jones M (1980) The practical guide to structured systems design. Yourdon Press, New York
Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12):1053–1058 ACM Press, New York
Pereira RHR, García Perez-Schofield JB, Ortin F (2017) Modularizing application and database evolution – an aspect-oriented framework for orthogonal persistence. Softw Pract Exper 47(2):193–221
Ponisio ML (2006) Exploiting client usage to manage program modularity. University of Berne, PhD thesis
Przybylek A (2010) What is wrong with AOP?. In: 5th International Conference on Software and Data Technologies, Athens
Przybylek A (2011) Where the truth lies: AOP and its impact on software modularity. In: Giannakopoulou D, Orejas F (eds) ETAPS 2011. LNCS, vol. 6603. Springer, Heidelberg, pp 447–461
Przybylek A (2013) Quasi-controlled Experimentations on the Impact of AOP on Software Comprehensibility. In: 17th European Conference on Software Maintenance and Reengineering, Genova
Rashid A, Moreira A (2006) Domain models are not aspect free. In: 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06), Genova
Rinard M, Salcianu A, Bugrara S (2004) A classification system and analysis for Aspect-Oriented programs. In: 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Newport Beach
Ryder BG, Tip F (2001) Change impact analysis for object-oriented programs. In: 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Snowbird

Sant'Anna C, Garcia A, Chavez C, Lucena C, von Staa A (2003) On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In: 17th Brazilian Symposium on Software Engineering, Manaus
Santos A, Alves P, Figueiredo E, Ferrari F (2016) Avoiding code pitfalls in aspect-oriented programming. Sci Comput Program 119:31–50
Scholtz J, Wiedenbeck S (1992) The use of unfamiliar programming languages by experienced programmers. In: 7th Conference of the British Computer Society Human Computer Interaction Specialist Group - People and Computers VII, York
Serebrenik A, van den Brand M (2010) Theil index for aggregation of software metrics values. In: 26th IEEE International Conference on Software Maintenance (ICSM'2010). Timisoara
Shen H, Zhang S, Zhao J (2008) An Empirical Study of Maintainability in Aspect-Oriented System Evolution Using Coupling Metrics. In: 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering, Nanjing
Standish T (1984) An essay on software reuse. IEEE trans. On. Softw Eng 10(5):494–497
Steimann F (2006) The paradoxical success of aspect-oriented programming. SIGPLAN Not 41(10):481–497
Steimann F, Pawlitzki T, Apel S, Kästner CH (2010) Types and modularity for implicit invocation with implicit announcement. ACM Trans Softw Eng Methodol 20(1):43
Storey MD, Fracchia FD, Müller HA (1999) Cognitive design elements to support the construction of a mental model during software exploration. J Syst Softw 44(3):171–185
Tonella P, Ceccato M (2005) Refactoring the aspectizable interfaces: an empirical assessment. IEEE Trans Softw Eng 31(10):819–832
Torkar R, Feldt R, Oliveira Neto FG, Gren L (2017) Statistical and practical significance of empirical software engineering research: A maturity model. In: arXiv:1706.00933v3 [cs.SE]
Tourwé T, Brichau J, Gybels K (2003) On the existence of the AOSD-evolution paradox. In: AOSD 2003 workshop on software-engineering properties of languages for aspect technologies, Boston
Tsang SL, Clarke S, Baniassad EL (2004) An evaluation of aspect-oriented programming for java-based real-time systems development. In: 7th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'04), Vienna
Walker R, Baniassad E, Murphy G (1999) An initial assessment of aspect-oriented programming. In: 21st international conference on software engineering (ICSE), Los Angeles
Yourdon E, Constantine LL (1979) Structured design: fundamentals of a discipline of computer program and system design. Prentice-Hall, New York
Zelkowitz MV, Wallace DR (1998) Experimental models for validating technology. Computer 31(5):23–31

Adam Przybyłek is an assistant professor at Gdansk University of Technology, Poland, where he has been working since October 2012. Between 2002 and 2011 he was a network consultant and instructor at Cisco Networking Academy. He obtained his Ph.D. degree in Software Engineering in 2011. He also holds a master's degree in Management Information Systems. His main research interests are in empirical software engineering with focus on software modularity, program comprehension, software evolution, post object-oriented paradigms, agile methods, and software process improvement. Adam is the founder and programme chair of the International Conference on Lean and Agile Software Development (https://fedcsis.org/lasd/). He has also served on the program committees for ENASE and ACM SAC since 2015..