Refactoring the Aspectizable Interfaces: An Empirical Assessment

Paolo Tonella, *Member*, *IEEE*, and Mariano Ceccato, *Student Member*, *IEEE*

Abstract—Aspect Oriented Programming aims at addressing the problem of the crosscutting concerns, i.e., those functionalities that are scattered among several modules in a given system. Aspects can be defined to modularize such concerns. In this work, we focus on a specific kind of crosscutting concerns, the scattered implementation of methods declared by interfaces that do not belong to the principal decomposition. We call such interfaces *aspectizable*. All the aspectizable interfaces identified within a large number of classes from the Java Standard Library and from three Java applications have been automatically migrated to aspects. To assess the effects of the migration on the internal and external quality attributes of these systems, we collected a set of metrics and we conducted an empirical study, in which some maintenance tasks were executed on the two alternative versions (with and without aspects) of the same system. In this paper, we report the results of such a comparison.

Index Terms—Aspect oriented programming, refactoring, program transformations, empirical studies.

1 Introduction

Aspect Oriented Programming (AOP) [1] offers programming constructs to modularize the crosscutting concerns implemented in a program. Instead of distributing the code for a transversal functionality (such as logging or persistence) across the classes in a system, AOP allows locating it into a so called *aspect*, the modularization unit for the crosscutting concerns. Understanding and changing such concerns is expected to be simplified once they are explicitly represented in the system's organization as a set of modules (aspects) devoted to them.

In Object Oriented Programming (OOP), the principal decomposition is mainly described by the class hierarchy (inheritance). Interclass relationships (aggregations, associations, etc.) and class properties (operations and attributes) complete the description. The interfaces and the related realization relationships naturally tend to crosscut the main view. In fact, it is necessary to use an interface each time classes that are taken from separate hierarchies must be dealt with uniformly. The common set of operations that they are required to provide is declared in the interface and is implemented by each class realizing the interface. For example, all classes whose instances can be serialized for storage or remote transmission will implement the *Serializable* interface, regardless of the hierarchy they belong in. This allows transparent management of objects from different classes implementing the same interface.

Thus, the methods defined to implement an interface are quite likely to represent good candidate aspects in that they are scattered code fragments that refer to a common, transversal functionality. For example, methods *readObject* and *writeObject* (required by *Serializable*) are often scattered across multiple, different hierarchies. However, not all the implementations of interfaces are associated with a crosscutting concern. For example, the interfaces *Collection* and *Map* implemented by some of the classes in the *java.util* package define (instead of crosscutting) the principal organization of the container classes. We call *aspectizable interfaces* those interfaces which crosscut the principal decomposition.

The main contribution of this paper is the assessment of the effects of migrating the implementation of the aspectizable interfaces to aspects, both in terms of internal and external quality attributes of the refactored applications. More than 1,300 classes (470,000 lines of code) have been analyzed and all their aspectizable interfaces have been migrated to aspects. In the refactored code, internal properties, such as size and modularity, are impacted. Externally, the expected benefits are an improved understandability and maintainability.

In order to compare the OOP and the AOP versions of the same applications, we collected a set of size and modularity metrics. The existence of a statistically significant difference between the values for the AOP code versus the values for the OOP code indicates that migration of the aspectizable interfaces has an impact on the internal code structure. However, such an effect alone does not necessarily correspond to an externally measurable benefit. In order to also assess the impact on external quality attributes, we conducted an empirical study with users, involving the execution of some maintenance tasks both on the OOP and on the AOP version of the same system. During the experimental sessions we collected some metrics that quantify the maintenance and understanding effort. User studies are inherently difficult to conduct and the results we obtained are in some respects limited. However, they allowed us to make some hypotheses on the relationship between the modularity improvement associated with

• *The authors are with the ITC-irst, Centro per la Ricerca Scientifica e Tecnologica, 38050 Povo (Trento), Italy.*
E-mail: {tonella, ceccato}@itc.it.

Manuscript received 23 Mar. 2005; revised 20 Sept. 2005; accepted 26 Sept. 2005; published online 3 Nov. 2005.
Recommended for acceptance by Harman, Korel, and Linos.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0075-0305.

the AOP code and its effects on the code maintainability/understandability.

The paper is organized as follows: Section 2 defines the notion of aspectizable interface and describes the technique we used to identify them in existing code and to migrate them to aspects. Section 3 gives a detailed description of the research questions that guided our study. The experimental design used to test the related hypotheses is also presented in this section, together with the metrics we collected and the statistical tests used to evaluate the significance of the differences. Section 4 presents the experimental results, followed by our conclusions and future work.

2 Aspectizable Interfaces

The notion of *aspectizable interface* was introduced for the first time in [2]. When a program is designed according to the OOP paradigm, the hierarchy of the classes reflects the (principal) decomposition of data structures and functions into smaller, composable units. In such a decomposition, the interfaces play a twofold role:

1. An interface may collect abstract properties of the principal decomposition, shared by the classes implementing it.
2. An interface may collect transversal properties, that crosscut the principal decomposition. Such properties recur across multiple unrelated classes, instead of being confined within a single, cohesive group of classes.

We call the latter an *aspectizable interface*.

If we consider the organization of the collection framework in the Java standard library, the interface *Collection* is a good example of the first kind, while *Serializable* belongs to the second case. In fact, the interface *Collection* is used to describe the container role that is played by classes from different subhierarchies, thus clearly contributing to the definition of the main organization of the library into smaller modularization units. On the other hand, the interface *Serializable* is not specific to the organization of the collection framework into subunits, being rather a transversal property of the classes in this package (and also of classes in other packages).

Another example of usage of interfaces is associated with the implementation of design patterns [3]. Roles are superimposed to the modules in a subsystem by making them realize given interfaces. For example, *Observable* and *Observer* interfaces can be introduced to specify the roles played by different classes organized according to the Model View Controller (MVC) pattern. Since role superimposition is typically orthogonal to the principal decomposition, this usage of the interfaces usually falls into the category of the aspectizable ones.

It should be noticed that the issues related to the implementation of the aspectizable interfaces generalize to all programming languages that support multiple object substitutability, i.e., the possibility for an object to play multiple roles in different contexts. In fact, one of such roles is typically dominant, the others being crosscutting with respect to the principal decomposition. For example, in the C++ language, which supports multiple inheritance, the principal decomposition is usually a slice of the class hierarchy, while some of the superclasses are better regarded as the declaration (possibly including a default implementation) of crosscutting properties. The relationship between aspects and multiple inheritance/roles is clarified in two papers by Hanenberg and Unland [4], [5]. Role-based refactoring of crosscutting concerns is described in a paper by Hannemann et al. [6].

2.1 Modularization of the Aspectizable Interfaces

Since the behavior captured by an aspectizable interface is shared by classes spread across different hierarchies, its OOP implementation consists of code fragments that are part of several classes. This phenomenon is known as code *scattering*. Moreover, the implementation requires intimate knowledge of the classes possessing such a behavior, giving raise to *tangled code*. When a behavior implemented in a software system is scattered (nonlocal implementation) and tangled (highly coupled) with the remaining code, the maintainability is expected to be affected negatively. Changes of such a behavior cannot be implemented locally (due to scattering) and ripple effects are hard to predict (due to tangling). A system property with these two features is called a *crosscutting concern* and Aspect Oriented Programming (AOP) [1] was developed with the explicit aim of addressing the crosscutting concerns that are often present in software systems. A new modularization unit, called an *aspect*, can be used to factor out all code fragments implementing a crosscutting concern.

Among the available proposals of programming languages inspired by the AOP principles, we will refer to AspectJ [7], an extension of Java with aspects. AspectJ offers two main programming constructs to modularize the crosscutting concerns, pointcuts and introductions. A *pointcut* is used to intercept an execution point (called a *join point*) in order to alter the original program behavior. When a crosscutting concern is factored out into an aspect, pointcuts are used to untangle it from the original code. *Introductions* are used to add properties (attributes, methods, interface implementations, etc.) to a given class. In this way, all properties related to a crosscutting concern can be modularized inside the related aspect, instead of being scattered across the classes sharing the concern.

When the principal decomposition is unaware of some aspect, in that it has no explicit reference to it, we say that it enjoys the *obliviousness* property. This does not mean that a working system can be built without the aspect. When this latter, stronger property occurs, we say the aspect is *optional*. Obliviousness and optionality are desirable properties when aspects are introduced, although it is not always possible to achieve both completely.

When a software system is designed, there is a high degree of subjectivity in the assignment of a functionality to the principal decomposition versus an aspect. Actually, such a choice is a typical design decision, where the experience of the designer plays a major role and the underlying motivations are hard to quantify. However, the preliminary aspect mining work that we conducted to identify aspectizable interfaces indicates that the separation

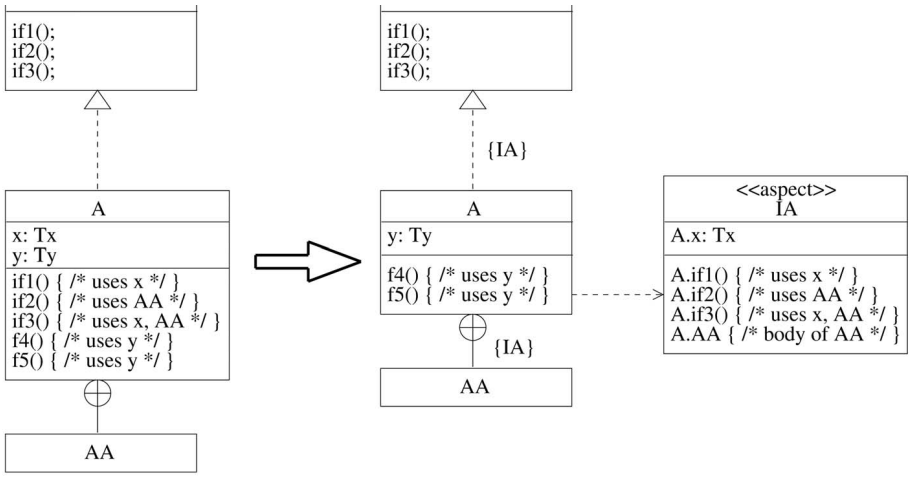

Fig. 1. Refactoring: Move interface implementation to aspect.

between interfaces belonging to the principal decomposition and aspectizable interfaces is very sharp and that very elementary aspect mining techniques (e.g., looking for simple patterns in interface names) work very well in practice. Thus, in the following, we make the assumption that the identification of which interfaces to migrate to aspects is a relatively easy task and we focus on the refactored code.

2.2 Refactoring

Migration of an aspectizable interface to an aspect is not different from the migration of a (general) crosscutting concern to an aspect. This involves two main, high-level code transformations (refactorings, [8]):

1. *Move properties to aspect*: properties (attributes, methods, inner classes) are modularized in the aspect, that introduces them into the affected classes.
2. *Remove references to properties*: execution points referencing aspectized properties are moved into the aspect code (called *advice* code) triggered by the pointcuts.

In the case of the aspectizable interfaces, the first transformation is the most important one since the methods in the interface implementations are seldom referenced by methods in the principal decomposition.

Fig. 1 shows the mechanics of the first refactoring. The overall transformation can be described in terms of three simpler refactoring steps, applied repeatedly:

- *Move method to aspect*.
- *Move field to aspect*.
- *Move inner class to aspect*.

These three (atomic) refactorings consist of removing a method (respectively, field or inner class) from a given class and adding it to an aspect, where it becomes an introduction.

In Fig. 1, class `A` implements the interface `I` by defining the body of methods `if1`, `if2`, `if3`, the class field `x` is used only inside `if1`, `if3`, and the inner class `AA` is used only inside `if2`, `if3`. Moving the interface implementation to a new aspect `IA` consists of applying the three steps above respectively to `if1`, `if2`, `if3`, to `x`, and to `AA`.

The result (see Fig. 1, right) is a thinner class `A`, with only one field (`y`) and two methods (`f4`, `f5`), which depends (dashed edge) on the aspect `IA` for the implementation of the interface `I` (see tag over the realization relationship). Inclusion of the inner class `AA` is also dependent on the new aspect `IA` (tag over nesting relationship).

Inheritance of interfaces is handled by computing the union of all superinterface methods (flattening). In fact, when a class implements an interface it must also implement all the superinterface methods. Thus, the methods declared in the superinterfaces are also migrated to the aspect being constructed.

When an interface migrated to an aspect is implemented by several classes in the system under analysis, additional advantages can be potentially obtained from the separation of the crosscutting concern represented by the interface. In fact, if the different implementations of the interface share some computations, it becomes possible to factor them out into a common superaspect.

The refactoring shown in Fig. 1 was implemented in the TXL [9] module *UNPLUG* (see Fig. 2). Given an input source (`A.java`) and an interface to be migrated to an aspect (`I.java`), the TXL module *UNPLUG* produces a new source file (`A'.java`), in which the interface implementation is absent, and an aspect (`IA.java`), which introduces the interface implementation into the original class. The aspect is also responsible for declaring that the class implements the interface, using the `declare parents` construct provided by AspectJ. Abstraction of common aspect code into a superaspect is performed manually (clone detection [10], [11], [12] techniques can be

Fig. 2. The refactoring module *UNPLUG*.

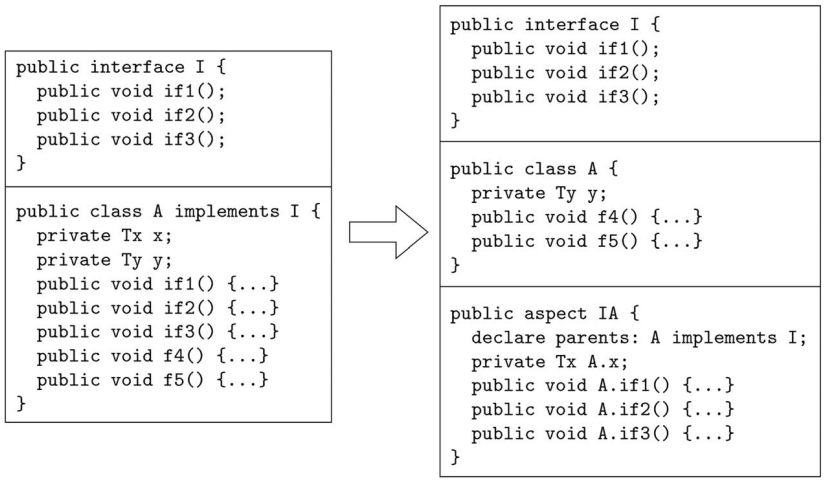

Fig. 3. Source code transformation operated by *UNPLUG*.

used to locate candidate code fragments), as a postprocessing aimed at refining the automatically produced aspects.

Fig. 3 shows the output generated by UNPLUG, when it is applied to the example depicted in Fig. 1 (inner class excluded). The responsibility of implementing the interface `I` is taken out of class `A` and is assigned to the aspect `IA`, so as to leave only principal decomposition methods inside `A` (assuming `I` is an aspectizable interface). Thus, one aspect is generated for each implementation of a given aspectizable interface.

Some limitations of the current version of AspectJ prevented us from implementing exactly the refactoring presented above. In particular, in AspectJ (version 1.2) inner classes cannot be introduced by an aspect. To cope with this problem, the adopted workaround consists of bringing the innerclass to the top level, with package-protected visibility, and inserting it into the same file containing the implementation of the aspect.

3 Assessment

Migration of OOP code to AOP is expected to be beneficial for external quality attributes such as the understandability and maintainability, as well as for internal quality attributes, such as the modularity. In fact, the possibility to encapsulate separate concerns should result in a localized comprehension and modification effort, and an improved code structure.

3.1 Aim of the Study

Following the Goal Question Metrics (GQM) approach [13], we stated the overall aim of this study:

Goal. Comparison between OOP and AOP implementations of aspectizable interfaces.

The object of the study is a set of OOP systems migrated to AOP by our tool *UNPLUG*. The purpose is the comparison of the two alternative implementations, with a focus on two external quality attributes: *maintainability* and *understandability*. The viewpoint is that of the code maintainers in a typical development setting.

3.2 Experimental Hypotheses

The experimental hypotheses that are tested by means of this study can be expressed through the following list of research questions:

- **RQ1**. Is the AOP code for the aspectizable interfaces easier to maintain than the OOP code?
- **RQ2**. Is the AOP code for the aspectizable interfaces easier to understand than the OOP code?
- **RQ3**. Does the migration of the aspectizable interface code produce a significant size reduction in the principal decomposition?
- **RQ4**. Does the migration of the aspectizable interface code produce an improved modularity in the principal decomposition?

The research questions RQ1 and RQ2 refer to external quality attributes (understandability and maintainability), while RQ3 and RQ4 are focused on internal quality attributes (size and modularity). All of them aim at testing the hypothesis that a significant difference does hold between the AOP and the OOP implementation of the aspectizable interfaces, in terms of some external/internal quality attribute. Thus, the null hypothesis is that there is no statistically significant difference in the quality attributes for the AOP versus OOP versions of the same applications.

RQ1 and RQ2 are the most interesting research questions since they address quality attributes that directly impact the code evolution process. A positive answer to these questions implies that an externally measurable benefit is expected to occur when migrating the aspectizable interfaces to AOP. On the other hand, these questions are related to quality attributes that are quite difficult to measure, in that their quantification requires the execution of a properly designed empirical study with users.

RQ3 and RQ4 are only indirectly associated with externally measurable benefits since they refer to internal features of the source code, whose impact on the code evolution process are hypothesized but not proved. On the other hand, it is quite easy to obtain metrics that quantify these attributes since this requires just the execution of some static code analysis.

3.3 Internal Quality Attributes

For the internal quality attributes, we customized some metrics available in the literature, in order to capture the size and modularity of the code in the principal decomposition.

3.3.1 Metrics

We focused on the two internal attributes, *size* and *modularity*, which are expected to be most affected by the aspectization of the crosscutting interfaces. In fact, migration to AOP of the aspectizable interfaces results in a decrease of the size of the classes that implement such interfaces. Moreover, the expected effects on the modularity are an increased cohesion of the classes, assuming that the aspectizable interface methods contribute negatively to the class cohesion, being loosely associated with the class main function. A decreased coupling is also expected since the dependency on the aspectizable interface is removed from a class when the interface implementation is aspectized. For the definition of the modularity (cohesion and coupling) metrics, we applied the framework described in [14], [15].

Size measures:

- **UCLOC**. Uncommented Lines Of Code.
- **OP**. Class Operations (methods, constructors, setters, and getters).

In order to normalize the size measure given by the lines of code, we produce a pretty-printed version of the code without comments, in which the same formatting convention is applied to every file. The number of lines resulting from such a processing gives our UCLOC metric.

Interface aspectization results in some class properties moved to an aspect. The metric OP, counting the number of operations in each class (inherited operations excluded), allows a direct measure of the size reduction produced by the refactoring of the aspectizable interfaces.

Modularity measures:

- **OCOH**. Operation Cohesion: pairs of distinct methods such that the first calls the second, normalized by the maximum number of such pairs.
- **ACOH**. Attribute Cohesion: pairs of distinct methods which reference at least one common attribute, normalized by the maximum number of such pairs.
- **ICOUPL**. Interface Coupling: number of interface realization relationships.

The definition of OCOH and ACOH is motivated by the following observation: The implementation of the aspectizable interfaces is associated with transversal functions, which are not specific to the class hosting the implemented methods, so we expect that such methods are seldom used by, or using, other methods. Moreover, being not associated with the class main responsibility, these methods might operate on a small subset of the class attributes. Thus, we measure the connections between methods due to invocation or attribute reference, in order to capture the increased connectivity (i.e., cohesion) that is expected to occur when the methods that implement the aspectizable interfaces are refactored.

In measuring OCOH, polymorphism is dealt with statically. In measuring ACOH, getter/setter methods and constructors are excluded because these special purpose methods must necessarily reference the class attributes, thus introducing a bias in the cohesion metric. In measuring both OCOH and ACOH, inherited properties (methods and attributes) are excluded. OCOH and ACOH satisfy the four properties proposed in [14] for the theoretical validation of any cohesion metric (nonnegativity and normalization, null value, monotonicity, merging of unconnected classes).

Since we aim at decoupling the implementation of the aspectizable interfaces from the principal decomposition, the chosen coupling metric counts the number of interfaces implemented by a class (completely inherited interface implementations are not counted). This corresponds to the intuitive notion of a class being coupled to an interface whenever the former implements the latter. ICOUPL satisfies the five properties proposed in [15] for the theoretical validation of any coupling measure (nonnegativity, null value, monotonicity, merging of classes, merging of unconnected classes).

3.3.2 Statistical Test

Statistical tests are used to determine whether the differences between the metrics collected either in the AOP or in the OOP setting are significant or not. For the internal (size and modularity) metrics, the *Pearson Chi-square test* can be used.

The Chi-square test is a nonparametric test of statistical significance for bivariate tabular analysis. It computes the degree of confidence in rejecting the null hypothesis, which states that the differences observed between two samples, taken form two populations, are due to random error. If the null hypothesis is rejected by the Chi-square test, the two samples are different enough, with respect to the considered characteristic, that we are allowed to generalize and conclude that the whole populations from which the samples are drawn are also different in the same characteristic. This means that the relationship between the samples and the characteristic is systematic in the populations and is not due to random error alone.

The Chi-square test requires to fix the *tolerance for error*, i.e., the probability of rejecting a true null hypothesis. This is commonly set to 0.05 (i.e., 5 percent). Data are grouped into categories that form a partition of the observed values. Percentiles are usually adopted in this phase. The frequencies of data in each category (percentile), divided by the independent variable (AOP versus OOP), are compared to the expected frequencies, obtained regardless of the independent variable. The output of the Chi-square test is a *Chi-square value*. When this value is above a threshold (computed from the tolerance for error), the observed differences can be assumed to generalize, i.e., to be due to the independent variable, even in the larger populations. In such a case, the strength of the association between the independent variable (AOP versus OOP) and the observed metrics (size or modularity) is given by the *shared variance*, i.e., the portion of the total variable distribution which is due to the relationship between independent and dependent variables.

3.4 External Quality Attributes

For the external quality attributes, we mapped the intuitive notions of maintainability and understandability into an effort metric.

3.4.1 Metrics

Quantification of the external quality attributes considered in this study requires the execution of an empirical study, involving some subjects executing some maintenance tasks either on the AOP or on the OOP code (see below). The times measured during the execution of the maintenance tasks are used to estimate the maintenance and understanding effort. The chosen metrics for *maintainability* and *understandability* are respectively:

- **MTIME**. Maintenance Time: total time measured during the execution of each maintenance task.
- **UTIME**. Understanding Time: time measured during the execution of each maintenance task, in the *Understanding* state.

In order to measure the understanding time separately from the total maintenance time, a *Time recorder* tool was developed. It is displayed on the screen as a small window, that shows the state the programmer is in, during the execution of the maintenance task. A simple two-state model was adopted to distinguish between the *Understanding* state, in which the programmer is comprehending the code structure or is determining the code fragments to be changed, and the *Modifying* state, in which the programmer is editing the source code to implement the requested change, or is compiling/executing the code to see the effects of the change. State modification is triggered by the programmer, who is requested to click on a button of the *Time recorder* upon each state change. The current state of the *Time recorder* is shown as the button label.

3.4.2 Experimental Design

An empirical study was designed in order to measure the maintenance and understanding effort required to complete a set of typical maintenance tasks, executed either on the AOP or on the OOP code. The experiment involves four groups of subjects and two maintenance tasks. The independent variables are the source code being used (either OOP or AOP) and the task being executed (either Task1 or Task2). The dependent variables are the maintenance time (MTIME) and the understanding time (UTIME).

In order to minimize the effect of the order in which the tasks are executed and the two code variants (OOP versus AOP) are modified, we counterbalanced both the task execution order and the order of code modification. The resulting experimental design is shown in Table 1. The OOP code is the first to be modified by the groups G1 and G4, while the AOP code is modified first by G2 and G3. Moreover, Task1 is executed as the first task by the groups G1 and G2, while Task2 is executed first by G3 and G4.

It should be noted that a factorial (completely balanced) design involving 2 (tasks) $\times$ 4 (groups) $\times$ 2 (treatments, AOP versus OOP) = 16 sessions is not feasible in our case since replication of the same task by the same group under different treatments makes no sense because reexecution of the same task would be simplified by the previously accumulated knowledge (memory effect). Thus, we had to opt for the partially balanced design depicted in Table 1.

TABLE 1
Experimental Design of the Empirical Study to Measure Maintenance and Understanding Times

Group	OOP	AOP
G1	Task1(1^{st})	Task2(2^{nd})
G2	Task2(2^{nd})	Task1(1^{st})
G3	Task1(2^{nd})	Task2(1^{st})
G4	Task2(1^{st})	Task1(2^{nd})

3.4.3 Statistical Test

For the effort metrics collected during the execution of the empirical study, the appropriate statistical test is the *Anova* (Analysis of variance).

The Anova test aims at determining if the differences in means of the dependent variables are due to random error or to the independent variables (called *factors* in this context). This is obtained by partitioning the total variance into the component that is due to random error and the components that are due to differences between means. These latter variance components are then tested for statistical significance (via F-statistics) and, if significant, the null hypothesis of no differences between means is rejected.

The significance level *alpha* used in the F-statistics (probability of rejecting a true hypothesis) is usually set to 0.05. The output of the Anova test consists then of the p-value, i.e., the probability that the observed difference between means is due to pure chance. The commonly used significance threshold for the p-value is 0.05. Under this value, the null hypothesis is rejected and the difference between means is considered statistically significant.

4 Experimental Results

Refactoring of the aspectizable interfaces was applied to a large proportion of the standard Java library (all packages below `java` in the package hierarchy of JDK) and to three open source programs, JHotDraw, FreeTTS, and JGraph. Although necessarily limited in number, these examples range from component oriented classes to full applications, with intermediate cases, such as JHotDraw and JGraph, where a development framework is provided together with complete applications.

4.1 Data Set

Table 2 shows some dimensional data of the source code that we refactored. Classes in the standard JDK library distributed by Sun (http://java.sun.com/) were taken from all packages nested inside `java`, including all subpackages. JHotDraw (http://www.jhotdraw.org/) is a two-dimensional graphics framework for drawing editors, based on Erich Gamma's JHotDraw. It includes a standalone application to draw geometrical shapes of different kinds. FreeTTS (http://freetts.sourceforge.net/) is a speech synthesis system written in the Java programming language. FreeTTS

TABLE 2
Features of the Source Code under Analysis

Package	LOC	Classes	Unique Int.	Asp. Int.	Asp. Int. Impl.
java.applet	833	1	2	2	2
java.awt	140,274	259	56	5	86
java.beans	14,560	78	15	1	5
java.io	24,194	67	6	2	5
java.lang	34,330	100	6	2	3
java.math	5,794	5	2	1	2
java.net	20,713	52	4	2	3
java.nio	11,645	34	7	1	1
java.rmi	8,601	49	3	1	3
java.security	27,475	111	12	2	12
java.sql	13,741	12	1	0	0
java.text	25,480	43	6	2	23
java.util	54,893	103	12	2	39
Total	**382,533**	**914**	**132**	**23**	**184**
Application	**LOC**	**Classes**	**Unique Int.**	**Asp. Int**	**Asp. Int. Impl.**
JHotDraw	39,214	249	48	5	54
FreeTTS	31,099	113	22	1	5
JGraph	18,373	29	20	2	7
Total	**88,686**	**391**	**90**	**8**	**66**

The last three columns show the unique implemented interfaces, among them the aspectizable interfaces, and the number of implementations of the aspectizable interfaces.

TABLE 3
Aspectizable Interfaces Found in the Source Code under Analysis

Aspectizable interfaces	Package or application	Impl.
Accessible	java.applet, java.awt	21
Cloneable	java.awt, java.security, java.text, java.util, JHotDraw, JGraph	81
Comparable	java.io, java.net, java.nio	4
Dumpable	FreeTTS	5
Externalizable	java.awt	2
Runnable	java.awt, java.lang, JHotDraw	4
Serializable	java.applet, java.awt, java.beans, java.io, java.security, java.lang, java.math, java.net, java.rmi, java.text, java.util, JHotDraw, JGraph	96
Storable	JHotDraw	35
VersionRequester	JHotDraw	2

The number of implementations is given in the last column.

was developed by Sun Microsystems, based upon Flite, a small, fast, runtime speech synthesis engine, which, in turn, is based upon the University of Edinburgh's Festival Speech Synthesis System and Carnegie Mellon University's FestVox project. JGraph (http://www.jgraph.com/) is a graph visualization library, including several graphical and algorithmic functionalities. We used the source code of version 1.4.0 of JDK, version 5.4b of JHotDraw, version 1.2beta of FreeTTS, and version 3.1 of JGraph.

The Lines Of Code (LOC) in the second column of Table 2 were counted on the plain source files and include comments. They are the output given by the Unix command `wc`. The number of classes does not include inner classes, as well as interfaces. These two size measures are aligned with the typical values observed for Java applications, indicating that the examples are representative of the software development practice in Java.

The fourth column of Table 2 gives the number of actually implemented interfaces, excluding those interfaces that are declared to be implemented, but whose implementation is completely inherited from a superclass. This means that at least one of the interface methods must be defined or redefined by the class declaring the implementation of the interface. When more than one class implements the same interface in a JDK package or in one of the three applications under analysis, the implemented interface is counted once (i.e., *unique* implemented interfaces are counted).

The fifth column of Table 2 shows the number of aspectizable interfaces. Their identification was conducted automatically and refined manually. Table 3 enumerates the whole list of aspectizable interfaces for JDK and for the three applications under analysis. Among the aspect mining methods for the identification of the aspectizable interfaces that have been investigated in our previous work [2], we decided to use the one based on regular expressions. We looked for the interface names matching the pattern `"[A-Z][a-z]*ble"`, based on the observation that crosscutting functionalities are often expressed as "able"-ties (e.g., *Accessible*, *Serializable*). Such simple aspect mining technique allowed us to identify eight out of the nine aspectizable interfaces in the code under analysis (see Table 3), with a *recall* (fraction of aspects actually mined among those to be identified) equal to 89 percent. *Precision* (fraction of correct aspects among those retrieved) was 53 percent. Since the classification between interfaces in the principal decomposition and aspectizable interfaces was always very sharp with almost no controversial case, we are quite confident that the degree of subjectivity in the output of this activity is low.

Refactoring of each implementation of the aspectizable interfaces produced an aspect containing the introductions necessary to add the methods required by the interface. The number of refactored implementations is given in the last column of Table 2. This is also the number of generated aspects (one aspect is generated per implementation). In order to remove the invocations of the interface methods, it would be possible to exploit dynamic crosscutting (pointcuts and advices). However, this was considered out of the scope of the current work and was observed to be a minor issue for most of the code under analysis, where interface methods are seldom used in the base code itself (e.g., the methods *readObject* and *writeObject* required by the *Serializable* interface are offered to user classes, but are not used within the Java library classes themselves).

4.2 Internal Quality Attributes

4.2.1 Metrics

Table 4 shows the frequencies of the size and modularity metrics within each quartile (including 1/4 of the observed values, considered in increasing order) evaluated on the source code, both the original and the refactored versions.

TABLE 4
Internal Quality Metrics, Divided into Quartiles and Computed Either on the Original or on the Refactored Code

UCLOC	**12-85**	**85-220**	**223-388**	**392-3408**	**Total**
Original	47 (23.9%)	47 (23.9%)	54 (27.4%)	49 (24.9%)	197
Refactored	53 (26.9%)	51 (25.9%)	45 (22.8%)	48 (24.4%)	197
Total	**100**	**98**	**99**	**97**	**394**
OP	**1-10**	**11-20**	**21-35**	**36-252**	**Total**
Original	45 (22.8%)	51 (25.9%)	51 (25.9%)	50 (25.4%)	197
Refactored	55 (27.9%)	48 (24.4%)	49 (24.9%)	45 (24.9%)	197
Total	**100**	**99**	**100**	**95**	**394**
OCOH	**0.0-0.0102**	**0.0103-0.021**	**0.0219-0.0413**	**0.0414-1.0**	**Total**
Original	58 (29.4%)	51 (29.9%)	52 (26.4%)	36 (18.3%)	197
Refactored	42 (21.3%)	48 (24.4%)	46 (23.3%)	61 (31.0%)	197
Total	**100**	**99**	**98**	**97**	**394**
ACOH	**0.0-0.0396**	**0.04-0.1667**	**0.17-0.3889**	**0.3928-1.0**	**Total**
Original	54 (27.4%)	49 (24.9%)	51 (25.9%)	43 (21.8%)	197
Refactored	46 (23.4%)	50 (25.4%)	47 (23.9%)	54 (27.4%)	197
Total	**100**	**99**	**98**	**97**	**394**
ICOUPL	**0**	**1**	**2**	**3-7**	**Total**
Original	0 (0.0%)	79 (40.1%)	74 (37.6%)	44 (22.3%)	197
Refactored	103 (52.3%)	67 (34.0%)	21 (10.7%)	6 (3.0%)	197
Total	**103**	**146**	**95**	**50**	**394**

Only classes affected by the refactoring are considered (i.e., metrics are not computed for the generated aspects) since we are interested in evaluating the effect of migration on the principal decomposition. The crosscutting concerns, now modularized, are handled separately.

For each metric, quartiles are determined on the whole set of classes (both original and refactored classes), as apparent from row *Total*, where (approximately) the same number of classes populate each range. In the two rows above *Total* (labeled *Original* and *Refactored*), the classes in each range are distinguished based on the code provenance. The interesting case is when these two numbers differ for a given range of a metric. This indicates that original and refactored code differ with respect to the given metric in that they have a different number of classes with the metric in the given range.

Let us consider, for example, the metric OP. We can notice that 100 classes (approximately 25 percent) have a value of OP between 1 and 10 (first quartile), for 99 classes the value of OP is between 11 and 20 (second quartile), for 100 classes it is between 21 and 35, and, for 95 classes, it is between 36 and 252. Within the first quartile for OP, 45 classes (22.8 percent) belong to the original code, while 55 classes (27.9 percent) are from the code of the migrated applications. Similarly, in the second, third, and fourth quartiles there are, respectively, 51, 51, and 50 (25.9, 25.9, and 25.4 percent) classes from the original code and 48, 49, and 45 (24.4, 24.9, and 24.9 percent) classes from the refactored code. For ICOUPL the first two quartiles are replaced by the sets of classes, respectively, with this metric equal to 0 or 1 since the value of the median (1) does not allow splitting the data into two subgroups.

Whenever a deviation of the relative frequencies in the four quartiles can be observed with respect to the value 25 percent, we can hypothesize that the given metric differs in the original and in the refactored code. For example, less than 25 percent of the original classes have a value of OP in the first quartile, while more than 25 percent of the original classes have a higher value of OP (either in the second, third, or fourth quartiles). Correspondingly, the refactored code falls in the first quartile in more than 25 percent of the cases, while it is in the other quartiles with a relative frequency lower than 25 percent. This seems to indicate a trend: In the refactored code, the metric OP tends to be lower since the first quartile contains more refactored than original classes. A similar trend can be observed for UCLOC and ICOUPL, while the opposite trend (higher value in the refactored code) can be observed for OCOH and ACOH. This would indicate a lower size, a lower coupling, and a higher cohesion of the principal decomposition code after migrating the aspectizable interfaces. However, this conjecture must be subjected to the Chi-square test to check its statistical significance.

TABLE 5
Chi-Square Thresholds and Values

Metric	Threshold	Chi-square	Shared var.
UCLOC	7.82	1.35	-
OP	7.82	1.39	-
OCOH	7.82	**9.46**	2.4%
ACOH	7.82	2.06	-
ICOUPL	7.82	**162.43**	41.2%

For statistical significance, Chi-square $>$ Threshold is required.

4.2.2 Statistical Analysis

Table 5 shows the Chi-square figures compared with the respective thresholds for all the computed metrics. When the Chi-square computed on the sample data is greater than the threshold, the null hypothesis can be rejected. We achieve significance only for OCOH and ICOUPL, while for the other metrics the different distribution across the quartiles is not statistically significant.

If the null hypothesis can be rejected, we conclude that there exists a statistically significant relationship between the dependent variables (the metrics) and the independent (original versus refactored code) variable. The shared variance reported in Table 5 for the two metrics (OCOH and ICOUPL) that differ in a significant way measures the strength of such a relationship. Thus, 2.4 percent of the variance of OCOH and 41.2 percent of the variance of ICOUPL is due to the difference between original and refactored code. Migration of the aspectizable interfaces to aspects produced a significantly increased cohesion of the operations in each class (OCOH) and an even more significant decrease of the coupling with the interfaces (ICOUPL). On the other hand, no significant size reduction (metrics UCLOC and OP) and no significant increase of the cohesion around the attributes (ACOH) was observed. Thus, original code and refactored code have comparable size and comparable cohesion of the classes around the class attributes.

The effect of the decrease of ICOUPL on the understandability of the class diagram is apparent when Fig. 4a is compared to Fig. 4b. The hierarchy of the classes implementing the *Collection* interface as well as that related to the *Map* interface can be immediately identified in the new view. The diagram is no longer cluttered by crosscutting concerns, which previously obscured the core hierarchies and can now be represented in a separated view (aspect diagram), as shown in Fig. 5. In this diagram, the presence of three superaspects (*AbstractSerializable_Map*, *AbstractSerializable_Entry*, and *AbstractCloneable*) indicates that some implementations of the methods required by the aspectizable interfaces (*Serializable* and *Cloneable*) were the same. In the refactored version, such code is no longer duplicated, being inherited from the parent aspect.

The increased OCOH can be explained by the fact that the methods introduced to implement an aspectizable interface are often only loosely coupled with the principal functions operated in a given class. Correspondingly, they are seldom called by the other methods of the same class. For example, the methods *readObject* and *writeObject* required to implement the *Serializable* interface are called only implicitly when an object is serialized or deserialized to/from and object stream. No method of the class they belong to usually calls them directly. Separation of the methods which do not contribute to the main class functions improves the conceptual integrity of the class and potentially simplifies understanding.

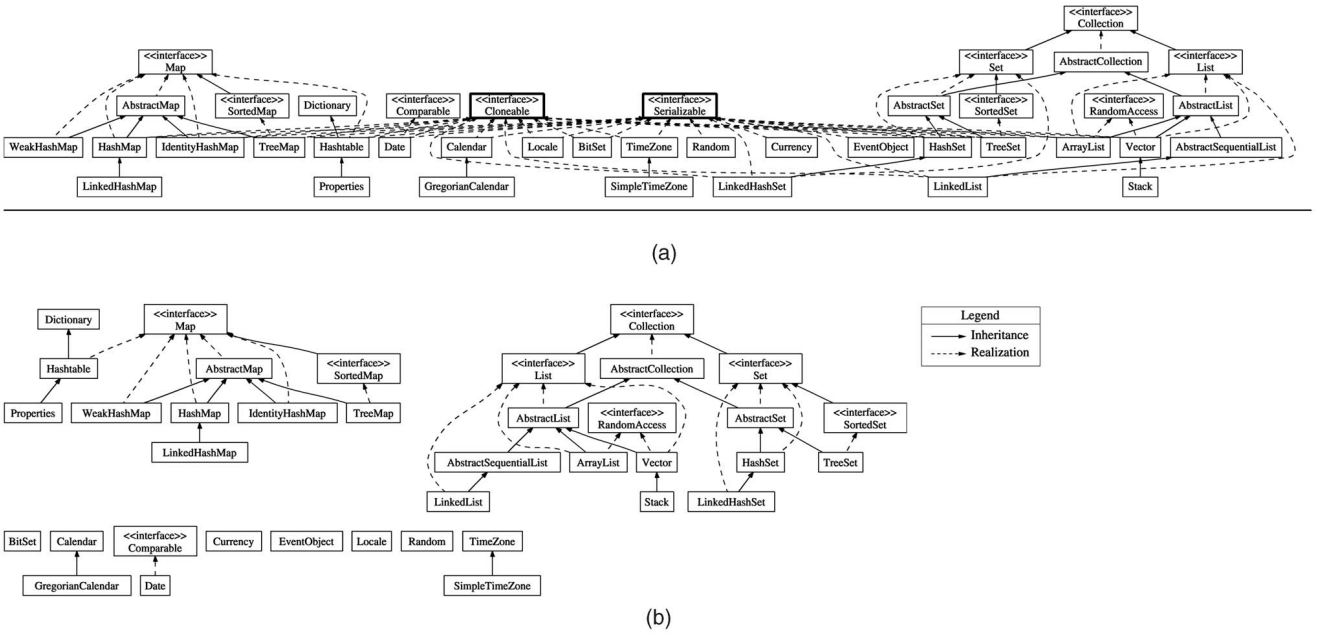

Fig. 4. Class diagram for the package *java.util*, with interfaces migrated to aspects in bold boxes (a). At the bottom (b), class diagram (principal decomposition) after migration of the aspectizable interfaces.

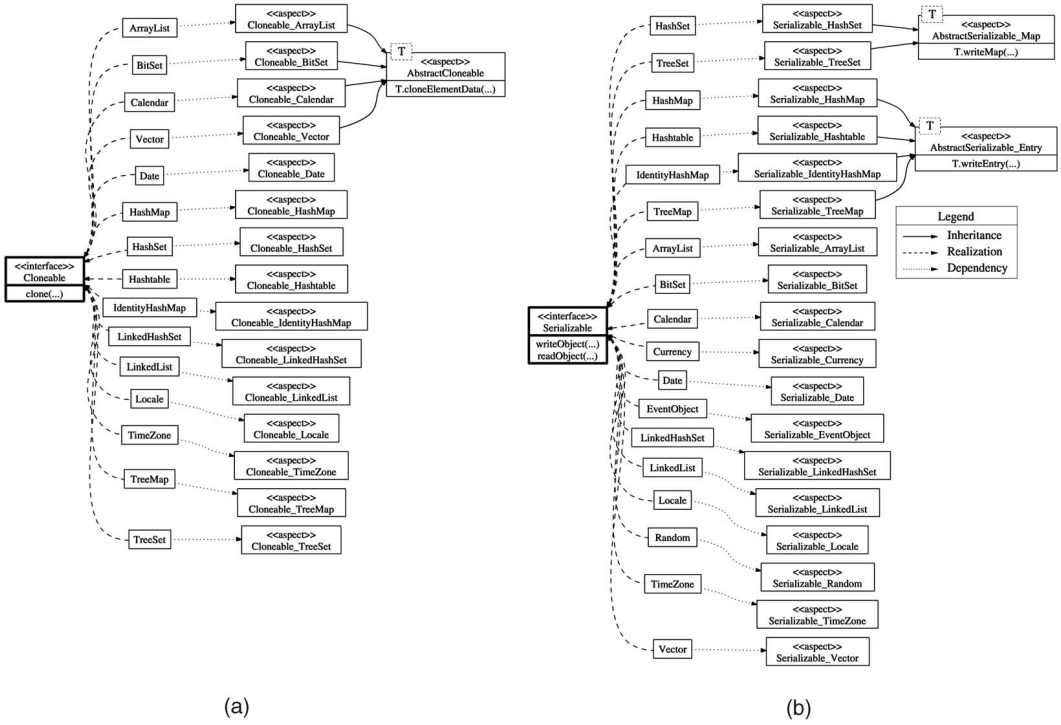

Fig. 5. Aspect diagram for the package *java.util*, representing the *Cloneable* (a) and *Serializable* (b) concern.

4.3 External Quality Attributes

4.3.1 Maintenance Tasks

The two tasks we defined for the empirical study are related to the source code of the `java.awt` package (and its subpackages), from JDK. Here is a summary description:

- **Task1**. In all the implementations of the `Cloneable` interface, a coding rule (not respected consistently in the `java.awt` package) on the scope of the variable declarations has to be enforced. It requires that variables be declared within the smallest scope where they are used.
- **Task2**. A specific error reporting protocol has to be enforced whenever the implementation of the `clone()` method, required by the `Cloneable` interface, invokes another implementation of `clone()` (either from the superclass or from the class of an attribute). All code portions not respecting the protocol must be updated to implement it.

The following criteria have been used to define them: tasks should be representative of typical interventions that can be required for library code such as that of JDK. They should have an intermediate complexity, making them non trivial to accomplish. At the same time, it should be possible to complete them within a reasonable time, since subjects are available for a limited time slot. We gave the subjects a limit of 2 hours and they were always able to complete them within such time.

4.3.2 Subjects

Twelve subjects were involved in the 24 experimental sessions conducted in our empirical study. Five subjects are researchers working at our Institute, four are programmers, three are students (two graduate and one undergraduate). All of them had previous experience with Java and OOP. All were already familiar with JDK. In order to evaluate their skills in Java and AspectJ programming, subjects' were administered an exercise on the first day of the experiment, after the tutorial on AspectJ. Successful completion of the exercise was considered an indicator of sufficient skill for the tasks at hand. Subjects were divided into four groups (G1-G4), each containing three subjects, and were randomly assigned Task1/Task2 in the OOP versus AOP setting according to the scheme shown in Table 1.

4.3.3 Setting

The maintenance tasks were executed in a typical programming environment, consisting of a code editor of choice (e.g., Emacs) and either the Java compiler javac or the AspectJ compiler ajc. The HTML documentation generated by means of javadoc was made available, in addition to the source code.

On the first day of the experiment, all subjects were given a short tutorial on AOP in general and AspectJ in particular. During such a presentation, they were also given some background information on the aspectizable interfaces and how they had been migrated to aspects. During the tutorial, the subjects practiced the usage of the time recorder.

On the second and third day, subjects executed the two maintenance tasks, either on AOP or on OOP code, according to the scheme prescribed by the experimental design. A complete environment, consisting of the source code, the HTML documentation and a compilation script, was installed on their machine.

After the completion of each experimental session, we collected the times measured by the time recorder, we made

TABLE 6
Understanding Time (UTIME) and Total Maintenance Time (MTIME) in Seconds, Measured in Each Experimental Session

	MTIME		UTIME	
	OOP	**AOP**	**OOP**	**AOP**
Task1	1028	1212	832	917
	828	561	772	453
	1072	555	664	523
	656	834	582	674
	1393	498	1176	361
	693	610	497	362
Avg	**945**	**712**	**754**	**548**
Var	76887	73707	57645	46250
Task2	1751	1455	1496	733
	2772	969	2664	840
	1538	1026	1165	803
	3418	1663	2327	1064
	1514	2289	1063	772
	2837	2356	2210	1391
Avg	**2305**	**1626**	**1821**	**934**
Var	652154	358945	445706	63678
Tot avg	**1625**	**1169**	**1287**	**741**
Vot var	835819	424828	539293	90498

Average (Avg) and variance (Var) are also shown.

a copy of the modified source code to check the successful completion of the tasks, and we gathered some free-format comments by the subjects, about the difficulties they encountered and the way they proceeded.

Task 1 applies to the classes belonging to *java.awt*, subpackages excluded, while Task 2 refers to *java.awt* and its subpackages. Correspondingly, subjects were given 106 files for Task 1 and 315 files for Task 2, in the OOP setting. They were given 157 files for Task 1 and 401 files for Task 2 in the AOP setting, due to presence of the files generated for the aspects. In both settings, execution of Task 1 required the modification of one file (two lines of code), while Task 2 required changing seven files (18 lines of code).

4.3.4 Metrics

Table 6 reports the amount of time (in seconds) spent in the execution of the maintenance tasks (both total time, MTIME, and understanding time, UTIME). Apparently, a reduction of the times in the AOP setting can be observed with respect to OOP, if we look at the averages. However, a high variability is also apparent from the values of the variances (in Table 6, *Var* is $\sigma^2 = \sum_{i=1}^{n} (x_i - \bar{x})^2/(n-1)$). Thus, execution of the Anova test is necessary to see if the difference between the averages is statistically meaningful or if it is due to the random error, which is responsible for the high variances.

4.3.5 Statistical Analysis

Table 7 shows the output of the Anova test, executed on the data obtained from the empirical study (both MTIME and UTIME). Row *Task* shows that there clearly is a statistically significant difference between the two tasks (p-value largely below 0.05 for both MTIME and UTIME). More interesting is row *AOP versus OOP*, indicating that the understanding time is significantly lower in the AOP setting than in the OOP setting, the difference being not due to random error (null hypothesis rejected). Rows *Interaction* and *Within group* account for the proportion of the total variance respectively due to the interaction between the two factors and to the variability internal to each group. As far as the maintenance time is concerned, although the p-value is not below the conventional threshold (0.05), it can be noticed that it is very close to such a value. We can say that the difference between maintaining the AOP versus the OOP code versions is close to significance, from a statistical point of view. In practice, a difference is observed and the probability that such a difference is due to chance is quite low (slightly above 5 percent). Thus, we can conclude that aspectization of the aspectizable interface implementations results in a significantly lower understanding time, during software maintenance, and that the overall maintenance time is also *quite* likely to decrease, thanks to the refactoring.

4.3.6 Threats to Validity

We did our best to minimize the threats to the internal validity of the study. However, the specific background of each of the involved subjects might have played a role. For example, the level of familiarity with Java and JDK in particular and the previous experience in executing comparable maintenance tasks, might have had an impact

TABLE 7
Two-Factor Anova for MTIME and UTIME: SS = Sum of Squares, df = Degrees of Freedom, MS = Mean Sum of Squares

	MTIME				
Source of Var.	**SS**	**df**	**MS**	**F (F-crit.)**	**P-val.**
Task	7761163	1	7761163	26.7 (4.35)	0.0000466
AOP vs. OOP	1247616	1	1247616	4.30 (4.35)	**0.0513**
Interaction	297483	1	297483	1.02 (4.35)	0.324
Within group	5808465	20	290423		
	UTIME				
Source of Var.	**SS**	**df**	**MS**	**F (F-crit.)**	**P-val.**
Task	3164634	1	3164634	20.6 (4.35)	0,000198
AOP vs. OOP	1790334	1	1790334	11.7 (4.35)	**0.00273**
Interaction	696663	1	696663	4.54 (4.35)	0.0456
Within group	3066398	20	153320		

For statistical significance, F > F-crit is required.

on the difficulties encountered. We mitigated this threat to validity by replicating the study as many times as possible (24 sessions were conducted in total). The tasks chosen for the experiment may also have affected the results, although we tried to simulate *typical* maintenance interventions. Replication of the experiment with different tasks would be necessary to address this threat. The time recorder, which is potentially a source of disturbance, was not judged so by the subjects, according to the comments collected after each session.

Possible threats to the external validity are the inclusion of non professional programmers in the sample and the difficulty of extrapolating to aspects different from the aspectizable interfaces considered in this study.

4.4 Discussion

With regards to the internal quality attributes, aspectization of the crosscutting interfaces increases significantly the class cohesion associated with the method invocations since the extracted methods are loosely connected with the other class operations. An even more significant decrease of the coupling with the implemented interfaces can be observed. This is an expected result since we are migrating those interfaces. The simplification of the class diagram associated with the reduced coupling with the interfaces, combined with the increased class cohesion, might explain the improved understandability that was observed in the execution of the maintenance tasks.

The main result of the user study is that the migration of the aspectizable interfaces produces code that is easier to understand, during the execution of a maintenance task. The overall effect on the maintainability is somehow less apparent. A possible explanation for this, which would require further empirical validation, is that refactoring of the aspectizable interfaces does not affect significantly the principal decomposition size, as resulting from the related metric, so that the modification phase is only marginally improved.

Let us reconsider the initial research questions (see Section 3). The answer to RQ1 is partially positive (the AOP code is *quite* likely to be easier to maintain). The answer to RQ2 (on the code understandability) is definitively positive. No significant size reduction was observed, associated with the migration of the aspectizable interfaces (RQ3), while the modularity metrics for the class cohesion due to method invocations and for the class coupling with the interfaces were significantly improved (respectively, increased and decreased), so that RQ4 can be given a positive answer.

Overall, this study indicates that migration of the aspectizable interfaces has a limited impact on the principal decomposition size, but, at the same time, it produces an improvement of the code modularity. From the point of view of the external quality attributes, modularization of the implementation of the crosscutting interfaces clearly simplifies the comprehension of the source code. We hypothesize that further benefits in the overall maintainability would be achieved if a larger fraction of the code were affected by the migration to AOP. However, further experiments are necessary to validate this hypothesis.

5 Related Work

Existing literature on AOP is mainly focused on language issues [1], [16], [17]. It is only more recently that examples of useful code aspectizations have been distilled and that the reorganization of existing systems into aspects has been considered [18], [19], [20]. Among the involved problems, aspect mining [21], [22], [23], [24] received a substantial attention. Some works considered also refactoring issues [25], [26].

Some of the various aspect mining approaches rely upon the user definition of likely aspects, usually at the lexical level, through regular expressions, and support the user in the code browsing and navigation activities conducted to locate them [21], [22], [24], [27]. Other approaches try to improve their identification by adding more automation. They exploit either execution traces [28], [29] or identifiers [30], often in conjunction with formal concept analysis [29], [30]. Clone detection [31], [32] and fan-in analysis [33] represent other alternatives in this category. As reported in our conference paper [2], mining of candidate aspects among the interfaces implemented by a class is a relatively easy task, that can take advantage of the simplest (e.g., lexical) techniques.

Manual refactoring of existing code toward AOP was conducted by Marius Marin [34]. More automation is exploited in the works by Hanenberg et al. [35], Gybels and Kellens [36], Tourwe et al. [37], and Binkley et al. [38]. The work of Hanenberg deals with the redefinition of popular OOP refactorings taken from Fowler's work [8] in order to make them aspect-aware, so that each potentially affected aspect is properly updated when the base code is refactored. Moreover, this work considers refactorings to migrate from OOP to AOP and refactorings that apply to AOP code. Gybels and Kellens, and Tourwe's work uses inductive logic programming to generalize an extensional definition of the pointcuts (that just enumerates all the join points), into an intensional one (antiunification). Binkley et al. considered a set of automated refactorings from OOP to AOP aimed at intercepting the original execution at the points where aspect behavior must be added. The refactoring introduced in the present work is somewhat simpler than those investigated in the literature since it deals just with the static crosscutting, that is necessary to migrate the implementation of the aspectizable interfaces. Correspondingly, the problem of defining proper pointcuts to intercept the execution whenever appropriate and the problem of generalizing the definition of such pointcuts are absent in our setting.

The novel contribution of the present work with respect to our conference paper [2] consists mainly of the empirical assessment. An empirical study was designed and executed to compare the OOP and AOP versions of the same systems, from the point of view of some internal and external quality attributes. Thus, the effects of interface aspectization that were only hypothesized in the conference paper [2] have been investigated thoroughly in the present work.

6 Conclusions

A large number of classes was subjected to our technique for the migration of the aspectizable interfaces to aspects. Then, we assessed the resulting code in terms of its internal and external quality attributes. The results indicate that among the internal attributes, only those referred to the modularity had a significant change. The size was not significantly affected by the migration. This is due to the fact that the amount of code devoted to the implementation of the aspectizable interface methods is typically a small fraction of the overall size.

Externally, the refactored code, which resorts to aspects for the implementation of the aspectizable interfaces, is definitively easier to understand than the code that mixes such implementation with the primary class responsibilities. To some extent, it is also easier to maintain, although the small size of the affected code portion might, in our opinion, have reduced the overall benefits on maintainability.

Clearly, it is difficult to extrapolate from a single study, focused on a specific kind of aspects—the crosscutting interface implementation. However, the strong indication of an improved understandability represents an important result for the AOP research at large. This is the first empirical study with users, in which an external quality attribute of the source code is shown to be improved by the migration to aspects.

In our future work, we intend to replicate this study in different, alternative settings. We will consider aspects often reported in the literature (e.g., persistence, logging, and tracing) and see if their aspectization produces effects similar to those observed for the aspectizable interfaces. In particular, we are interested in the relationship between the size of the aspectized code and the externally visible effects on the code maintainability. It would be interesting to refactor an increasing number of aspects, possibly of different kinds, up to the extreme situation where the principal decomposition remains empty, falling into the Multidimensional Separation of Concerns paradigm [17]. Assessment of the effects on the internal and external quality attributes of the resulting code would provide useful indications about the trade-off between modules in the principal decomposition and modules for the crosscutting functionalities, thus contributing to a clarification of the notion of aspect itself.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect Oriented Programming," *Proc. 11th European Conf. Object Oriented Programming (ECOOP),* 1997.

[2] P. Tonella and M. Ceccato, "Migrating Interface Implementation to Aspect Oriented Programming," *Proc. Int'l Conf. Software Maintenance (ICSM),* pp. 220-229, Sept. 2004.

[3] B. Bruegge and A.H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns and Java,* second ed. Prentice-Hall, 2003.

[4] S. Hanenberg and R. Unland, "Concerning AOP and Inheritance," *Proc. Workshop Aspect-Orientation,* 2001.

[5] S. Hanenberg and R. Unland, "Roles and Aspects: Similarities, Differences, And Synergetic Potential," *Proc. Eighth Int'l Conf. Object-Oriented Information Systems,* pp. 507-520, Sept. 2002.

[6] J. Hannemann, G.C. Murphy, and G. Kiczales, "Role-Based Refactoring of Crosscutting Concerns," *Proc. Fourth Int'l Conf. Aspect-Oriented Software Development (AOSD '05),* pp. 135-146, 2005.

[7] I. Kiselev, *Aspect-Oriented Programming with AspectJ.* Indianapolis, In.: Sams Publishing, 2002.

[8] M. Fowler, *Refactoring: Improving the Design Of Existing Code.* Addison-Wesley, 1999.

[9] J. Cordy, T. Dean, A. Malton, and K. Schneider, "Source Transformation in Software Engineering Using the TXL Transformation System," *Information and Software Technology,* vol. 44, no. 13, pp. 827-837, 2002.

[10] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proc. Int'l Conf. Software Maintenance,* pp. 368-377, Nov. 1998.

[11] A. Lakhotia, J. Li, A. Walenstein, and Y. Yang, "Towards a Clone Detection Benchmark Suite and Results Archive," *Proc. Int'l Workshop Program Comprehension (IWPC),* pp. 285-287, May 2003.

[12] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," *Proc. Int'l Conf. Software Maintenance,* pp. 244-253, Nov. 1996.

[13] V. Basili, G. Caldiera, and D.H. Rombach, *The Goal Question Metric Paradigm, Encyclopedia of Software Engineering.* John Wiley and Sons, 1994.

[14] L. Briand, J. Daly, and J. Wuest, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Eng.,* vol. 3, no. 1, pp. 65-117, 1998.

[15] L. Briand, J. Daly, and J. Wuest, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Trans. Software Eng.,* vol. 25, no. 1, pp. 91-121, Jan./Feb. 1999.

[16] J. Hannemann and G. Kiczales, "Design Pattern Implementation in Java and AspectJ," *Proc. 17th Ann. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA),* pp. 161-173, Nov. 2002.

[17] P.L. Tarr, H. Ossher, W.H. Harrison, and S.M. Sutton Jr., "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proc. Int'l Conf. Software Eng. (ICSE),* pp. 107-119, May 1999.

[18] E.L.A. Baniassad, G.C. Murphy, C. Schwanninger, and M. Kircher, "Managing Crosscutting Concerns during Software Evolution Tasks: An Inquisitive Study," *Proc. First Int'l Conf. Aspect-Oriented Software Development (AOSD),* pp. 120-126, Apr. 2002.

[19] A. Rashid and R. Chitchyan, "Persistence as an Aspect," *Proc. Second Int'l Conf. Aspect-Oriented Software Development,* pp. 120-129, 2003.

[20] S. Soares, E. Laureano, and P. Borba, "Implementing Distribution and Persistence Aspects with AspectJ," *Proc. 17th Ann. ACM Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA),* pp. 174-190, Nov. 2002.

[21] J. Hannemann and G. Kiczales, "Overcoming the Prevalent Decomposition of Legacy Code," *Proc. Workshop Advanced Separation of Concerns at the Int'l Conf. Software Eng. (ICSE),* 2001.

[22] D. Janzen and K.D. Volder, "Navigating and Querying Code without Getting Lost," *Proc. Second Int'l Conf. Aspect-Oriented Software Development (AOSD),* pp. 178-187, Mar. 2003.

[23] N. Loughran and A. Rashid, "Mining Aspects," *Proc. Workshop Early Aspects: Aspect-Oriented Requirements Eng. and Architecture Design (with AOSD),* Apr. 2002.

[24] M.P. Robillard and G.C. Murphy, "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies," *Proc. 24th Int'l Conf. Software Eng. (ICSE),* pp. 406-416, May 2002.

[25] P. Borba and S. Soares, "Refactoring and Code Generation Tools for AspectJ," *Proc. Workshop Tools for Aspect-Oriented Software Development (with OOPSLA),* Nov. 2002.

[26] A. van Deursen, M. Marin, and L. Moonen, "Aspect Mining and Refactoring," *Proc. First Int'l Workshop Refactoring: Achievements, Challenges, Effects (REFACE),* Nov. 2003.

[27] W.G. Griswold, J.J. Yuan, and Y. Kato, "Exploiting the Map Metaphor in a Tool for Software Evolution," *Proc. 2001 Int'l Conf. Software Eng. (ICSE),* pp. 265-274, Mar. 2001.

[28] S. Breu and J. Krinke, "Aspect Mining Using Event Traces," *Proc. Conf. Automated Software Eng. (ASE 2004),* pp. 310-315, Sept. 2004.

[29] P. Tonella and M. Ceccato, "Aspect Mining through the Formal Concept Analysis of Execution Traces," *Proc. 11th Working Conf. Reverse Eng. (WCRE),* pp. 112-121, Nov. 2004.

[30] T. Tourwe and K. Mens, "Mining Aspectual Views Using Formal Concept Analysis," *Proc. Fourth IEEE Int'l Workshop Source Code Analysis and Manipulation (SCAM 2004),* pp. 97-106, Sept. 2004.

[31] M. Bruntink, A. van Deursen, T. Tourwé, and R. van Engelen, "An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns," *Proc. Int'l Conf. Software Maintenance (ICSM),* pp. 200-209, Sept. 2004.

[32] D. Shepherd, E. Gibson, and L. Pollock, "Design and Evaluation of an Automated Aspect Mining Tool," *Proc. Mid-Atlantic Student Workshop Programming Languages and Systems,* Apr. 2004.
[33] M. Marin, A. van Deursen, and L. Moonen, "Identifying Aspects Using Fan-In Analysis," *Proc. 11th IEEE Working Conf. Reverse Eng. (WCRE 2004),* Nov. 2004.
[34] M. Marin, "Refactoring Jhotdraw's Undo Concern to Aspectj," *Proc. First Workshop Aspect Reverse Eng. (WARE2004),* Nov. 2004.
[35] S. Hanenberg, C. Oberschulte, and R. Unland, "Refactoring of Aspect-Oriented Software," *Proc. Fourth Ann. Int'l Conf. Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net. ObjectDays),* pp. 19-35, Sept. 2003.
[36] K. Gybels and A. Kellens, "An Experiment in Using Inductive Logic Programming to Uncover Pointcuts," *Proc. First European Interactive Workshop Aspects in Software,* 2004.
[37] T. Tourwé, A. Kellens, W. Vanderperren, and F. Vannieuwenhuyse, "Inductively Generated Pointcuts to Support Refactoring to Aspects," *Proc. Software Eng. Properties of Languages for Aspect Technology (SPLAT) Workshop at AOSD '04,* 2004.
[38] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella, "Automated Refactoring of Object Oriented Code into Aspects," *Proc. Int'l Conf. Software Maintenance (ICSM),* pp. 27-36, Sept. 2005.

Paolo Tonella received the laurea degree cum laude in electronic engineering from the University of Padua, Italy, in 1992, and the PhD degree in software engineering from the same university, in 1999, with the thesis "Code Analysis in Support to Software Maintenance." Since 1994, he has been a full-time researcher of the Software Engineering Group at ITC-irst, Trento, Italy, where he is now a senior researcher. He participated in several industrial and European Community projects on software analysis and testing. He is the author of *Reverse Engineering of Object Oriented Code* (Springer, 2005). His current research interests include reverse engineering, aspect oriented programming, empirical studies, Web applications, and testing. He is a member of the IEEE.

Mariano Ceccato received the degree in software engineering (specialization on computer systems and applications) from the University of Padua in 2003. He is a PhD student at the University of Trento, Italy. His master's thesis concerned the re-engineering of an existing data warehouse application. The project was developed in the Information Technology department at Alcoa Servizi. In 2003, he joined the STAR group at ITC-irst, Trento, as a PhD student. His research interests are in source code analysis and manipulation, more specifically for the migration of object-oriented code to aspect-oriented programming. He is a student member of the IEEE.