Migrating Interface Implementations to Aspects

Paolo Tonella and Mariano Ceccato
ITC-irst
Centro per la Ricerca Scientifica e Tecnologica
38050 Povo (Trento), Italy
{tonella, ceccato}@itc.it

Abstract

Separation of concerns and modularization are the cornerstones of software engineering. However, when a system is decomposed into units, functionalities often emerge which cannot be assigned to a single element of the decomposition. The implementation of interfaces[1] *represents a typical instance of this problem. In fact, the code that defines the interface methods is often scattered across several classes in the system and tangled with the original code.*

Aspect Oriented Programming provides mechanisms for the dynamic and static composition of transversal functionalities, that can be used to factor out the implementation of interfaces. In this paper we describe a technique for the identification of those interface implementations that are most likely to represent crosscutting concerns. Moreover, the code transformation (refactoring) to migrate such interfaces to aspects is also presented. Experimental results validate the approach.

1 Introduction

Similarly to other engineering artifacts, the complexity of software systems is managed by applying the *"divide and conquer"* principle. The overall functionality is composed out of smaller units, with a well defined behavior and a low level of mutual coupling. Such a decomposition becomes the basic architecture of the system, an all successive changes are referred to it.

The weak point of this approach is that not all functionalities in a software system can be mapped to a single unit in the principal decomposition. Persistence is a typical example [14]. Code fragments to handle persistent storage and retrieval of data are often scattered across the modules in a system. Externally, such crosscutting functionalities are often viewed as interfaces implemented by classes.

[1]Not to be confused with graphical user interfaces.

Aspect Oriented Programming (AOP) [10] offers programming constructs to handle the crosscutting functionalities, which can be modularized in a similar way as the units in the principal decomposition. Instead of distributing the code that implements an interface across the classes in a system, it becomes possible to locate it in 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.

In this paper a technique is proposed for the aspectization of interface implementations. Identification of the interfaces that are most suitable for migration to aspects is conducted during an *aspect mining* phase, in which the source code is analyzed and candidate aspects are located. Then, in the *refactoring* phase, the code is transformed, so that interface implementations are realized by separate aspects instead of the original classes. We have implemented a toolkit to support the aspectization of interface implementation and we have applied it to the source code of some of the packages in the standard Java library. The aim of the experimental work was to assess the feasibility of the transformation and to evaluate the potential benefits.

Existing literature on AOP is mainly focused on language issues [8, 10, 17]. It is only more recently that examples of useful code aspectizations have been distilled and that the reorganization into aspects of existing systems has been considered [1, 14, 16]. Among the involved problems, aspect mining [7, 9, 12, 15] received a substantial attention. Some work also considered refactoring issues [3, 18]. No previous work (to the authors' knowledge) deals with the aspectization of interface implementation.

The paper is organized as follows: after a basic background on AOP, Section 3 motivates the migration of interface implementations to aspects. The proposed methods for aspect mining and refactoring are respectively described in Sections 4 and 5. The toolkit we developed to support the migration is presented in Section 6, while experimental results are provided in Section 7.

2 Aspect Oriented Programming

Programs are typically designed with a principal decomposition in mind. However, not all functionalities can be assigned to a single unit within such a decomposition. For example, persistence, concurrency and logging are functionalities whose code is often spread across several different units. Such kinds of functionalities are called *crosscutting concerns* and their main characteristic is that they are transversal with respect to the units in the principal decomposition, i.e., their implementation consists of a set of code fragments distributed over a number of units.

Crosscutting concerns are one of the main sources of problems during software maintenance. In fact, it is difficult to evolve a crosscutting concern, because its modification requires that:

- all code portions where such a functionality is implemented be located (problem: scattering);
- all ripple effects associated with the changes be determined (problem: tangling).

Aspect Oriented Programming (AOP) aims at solving the two main problems of crosscutting concerns, namely scattering and tangling, by providing a unique place where the related functionalities are implemented. A new modularization unit, called *aspect*, can be defined to factor out all code fragments related to a common functionality, otherwise spread all over the system. For example, an application can be developed according to its main logical decomposition, while the possibility to serialize and de-serialize some of its objects can be defined in a separate aspect.

Sometimes, the application being developed has a principal decomposition that is completely decoupled from its aspects. Such a property is called *obliviousness*. An application is oblivious of an aspect if it can be developed independently of it, and the aspect can be added (or removed) later, by compiling (*weaving*) the application with/without it. Obliviousness is not expected to hold for all aspects. For example, when persistence is implemented as a separate aspect, the application can be oblivious of this functionality, except for the deletion and retrieval of objects, which require two explicit invocations inside the modules in the principal decomposition [14] (partial obliviousness).

A related issue is *optionality*. An aspect may either be an optional or an integral part of an application. In the latter case, the application cannot be compiled without the aspect. However, it is not required that aspects be always optional features. In some cases, an application may require some of its aspects to work properly.

Among the programming languages and tools that have been developed to support AOP, AspectJ [11], an extension of Java with aspects, is one of the most popular and best supported. The two main new programming constructs provided by AspectJ for the definition of the behavior of an aspect are pointcuts and introductions.

2.1 Pointcuts

Pointcuts identify *join points* in a program, i.e., points in the control flow where execution can be intercepted by an aspect to alter the original behavior of the code. For example, if a persistence aspect is defined to serialize all objects of class *Person* as soon as they are created, an appropriate pointcut can be used to intercept calls to any constructor of class *Person*. In AspectJ, this looks like:

```
aspect PersistentPerson {
   pointcut personCreation(): call(Person.new(..));
   after(): personCreation() {
     // save Person data to db
   }
}
```

2.2 Introductions

While pointcuts operate dynamically, introductions aim at altering the static structure of an Object Oriented program. Introductions can be used to change the inheritance hierarchy, by modifying the super-class or the super-interfaces of a given class. Moreover, they can be used to insert new members (attributes or methods) into a class.

For example, it is possible to declare that a class implements a given interface inside an aspect, while the original code does not do it. In such a case, the methods required by the interface must be introduced as well.

With reference to the persistence aspect, it is possible to declare that the interface *Serializable* is implemented by class *Person* and to introduce the related methods, by means of the following code fragment:

```
aspect PersistentPerson {
   declare parents: Person implements Serializable;
   private void
       Person.readObject(ObjectInputStream in) {...}
   private void
       Person.writeObject(ObjectOutputStream out) {...}
}
```

3 Interface implementation as a crosscutting concern

The principal decomposition of a program is typically apparent from the hierarchy of its classes. In fact, the inheritance relationship usually models the refinement of abstract into concrete entities, according to a model of the application domain.

In addition to extending the super-class, classes may implement interfaces. Interfaces provide alternative decompositions of the functionalities, according to a different, possibly orthogonal view. For this reason, interfaces often (though not always) do not belong to the principal decomposition of a system.

Let us consider persistence (interface *Serializable*). The code fragments implementing this interface are spread across several classes (scattering). Moreover, this code requires access to information about each entity to be stored persistently (tangling). On the other side, if we consider the persistence functionality from a logical point of view, it clearly does not belong to the principal decomposition of this application. Rather, it is a transversal computation that has to be superimposed. In other words, it is an *aspect* of this application.

The reasons for considering the *Serializable* interface as an aspect hold for many of the interfaces that are usually implemented by the classes in a Java application. Another example is the interface *Cloneable*, which allows duplicating existing objects. When necessary, this functionality is added. However, it typically does not take part in the main decomposition of the system.

Interface implementation seems a good starting point for the identification of candidate aspects in an existing system. It involves the introduction of methods in a class – those required by the interface. Moreover, sometimes a proper (e.g., efficient) behavior of the interface methods require that additional attributes (e.g., a cache) be inserted into the given class and that inner classes be defined.

When an interface implementation is recognized as a crosscutting concern, it can be migrated to an aspect. This entails the extraction of the interface methods from the given classes, as well as of all attributes and inner classes that are functional to the implementation of the interface.

4 Aspect mining

The purpose of aspect mining is the identification of those interfaces, implemented in a given class, which can be regarded as crosscutting concerns and can be subjected to refactoring in order to aspectize them.

While a fully automated aspect identification process is not feasible, because of the fuzzy notion of principal decomposition and of the level of subjectivity involved, it is possible to define a set of indicators that hint a high likelihood that a given interface implementation represents an aspect. The result of computing such indicators must then be interpreted by a human, making the final decision.

We have defined the following set of aspect mining indicators specifically for the migration of interface implementations to aspects. The implementation of an interface is marked as a candidate aspect when:

External package The interface implemented in a class belongs to a package different from that of the given class.

String matching The name of the interface implemented in a class matches the pattern `".*able"`.

Clustering When methods are clustered according to the call relationship, interface methods are not grouped together with other (non-interface) class methods.

Unpluggability The methods of the interface implemented in a class can be unplugged from the given class, since they are not invoked by other methods of the same class.

The first criterion assumes that interfaces in the principal decomposition are declared in the same package as the classes under analysis. The second criterion was derived from the aspect mining methods based on string matching [7]. The regular expression matched against each interface name consists of an arbitrary prefix (`".*"`) followed by the suffix `"able"`. Such a suffix typically indicates an additional property of the given class, orthogonal to its main properties (e.g., *Serializable, Cloneable*).

The rationale behind the last two aspect mining criteria is that methods inserted into a class to implement an interface which is not in the principal decomposition are loosely coupled with the other methods of the class, since they do not contribute to the main functionalities of the class. The *Unpluggability* criterion makes the (strong) hypothesis that no method calls any one of the interface methods, except for the interface methods themselves. A weaker hypothesis, made in the *Clustering* criterion, is that the call relationship identifies subgroups of cohesive methods and that interface methods are never in a group including also non-interface methods. In other words, calls from non-interface methods are admitted, as long as calling methods are not highly coupled with the called interface methods.

The clustering method used to determine groups (clusters) of highly connected methods has been defined along the lines given in [13]. A metric of modularization quality, accounting for the difference between cohesion and coupling, is maximized by means of a proper combinatorial optimization heuristics (we used hill climbing).

Once an interface is classified as a candidate aspect, not only its methods are migrated to the new aspect, but also all fields and inner classes functional to the interface implementation. In order to identify which fields and inner classes should be assigned to the aspect being migrated, the following, simple criterion was adopted:

Fields and inner classes are migrated to the aspect associated with a given interface if they are used inside, but not outside the interface methods.

5 Refactoring

Migration of an interface implementation to an aspect can be achieved by applying a code transformation (*refactoring* [5]) that changes the system's decomposition while leaving the overall behavior unaffected. Interface methods (plus associated inner classes and fields, if any) are removed from the class being refactored and are inserted into a new aspect. In this way, the remaining properties of the refactored class reflect the principal decomposition of the system, while the migrated properties reflect the crosscutting view associated with the interface.

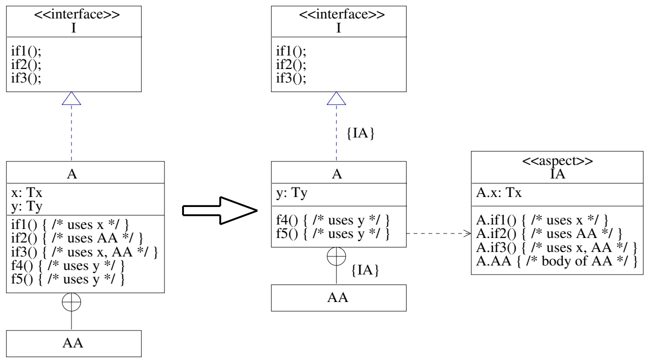


Figure 1. *Refactoring: Move interface implementation to aspect.*

Figure 1 shows the mechanics of this 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 (resp. field or inner class) from a given class and adding it to an aspect, where it becomes an introduction.

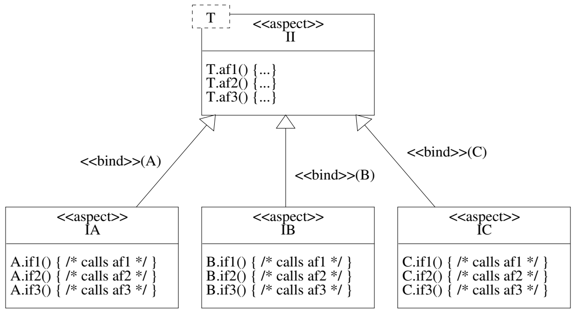
In Figure 1, class `A` implements the interface `I` by defining the body of methods `if1`, `if2`, `if3`, the class field `x` is used only inside `if1`, `if3`, and the inner class `AA` is used only inside `if2`, `if3`. Moving the interface implementation to a new aspect `IA` consists of applying the three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

The result (see Figure 1, right) is a thinner class `A`, with only 1 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).

In presence of interface hierarchies, the methods declared in the super-interfaces are also migrated to the aspect.

5.1 Abstracting aspects

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 super-aspect.



T
<<aspect>>
II
T.af1() {...}
T.af2() {...}
T.af3() {...}
<<bind>>(A)
<<bind>>(B)
<<bind>>(C)
<<aspect>>
IA
A.if1() { /* calls af1 */ }
A.if2() { /* calls af2 */ }
A.if3() { /* calls af3 */ }
<<aspect>>
IB
B.if1() { /* calls af1 */ }
B.if2() { /* calls af2 */ }
B.if3() { /* calls af3 */ }
<<aspect>>
IC
C.if1() { /* calls af1 */ }
C.if2() { /* calls af2 */ }
C.if3() { /* calls af3 */ }

Figure 2. *Abstraction of aspect computations into a super-aspect.*

Let us consider three classes `A`, `B`, `C`, implementing the interface `I` with interface methods `if1`, `if2`, `if3`. After applying the refactoring in Figure 1, three new aspects `IA`, `IB`, `IC` are created to separate the concern of interface implementation from the original classes. If the method introductions in the three aspects perform a same sub-computation, it is possible to separate it from the class-specific computations and move it to a new method (`af1` for the common sub-computation in `A.if1`, `B.if1`, `C.if1`, and similarly `af2` and `af3`). It should be noted that common sub-computations are quite likely to occur in practice in different implementations of the same interface methods, because (without the aspects) such implementations are spread across different classes, where factorization of the common sub-computations in a unique modular unit may be difficult to realize.

The common computations `af1`, `af2`, `af3` can be inserted into the classes `A`, `B`, `C` by a super-aspect `II`, extended by the three aspects `IA`, `IB`, `IC` (see Figure 2). In this super-aspect, a generic introduction is performed, using the type variable T. Sub-aspects bind the type variable

T to the classes they modify, so as to obtain the necessary introductions.

Identification and factorization of common computations into super-aspect introductions is a human-intensive activity, in that it involves lots of domain and application knowledge, and non trivial decision making. However, partial automation can be achieved by exploiting clone detection techniques [2]. The presence of a cloned code fragment inside interface methods introduced in different classes indicates the possibility of abstracting the related aspects and factoring out the cloned computation.

When common computations are identified in different aspects and abstracted into a super-aspect, the modularization power of AOP becomes even more evident. In fact, while initially a same interface is implemented in different classes by means of replicated code fragments, in the refactored system the commonalities among the different implementations are localized in a single module (no more scattering). In other words, the presence of a same crosscutting sub-computation is represented explicitly in the new modular unit being created, the super-aspect.

6 Tool

Automated support to *aspect mining* requires the execution of some code analyses on the input program. Information on the interface names and packages, and on the call relationships between methods has to be recovered, in order for the proposed aspect mining techniques to work. The static analyzer that performs this job was written in the source code transformation and analysis tool TXL [4]. A few small Java programs have been developed to implement the aspect mining checks described in Section 4, based upon the output produced by the TXL analyzer.

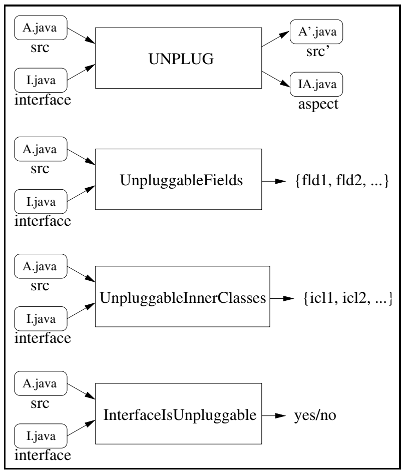


Figure 3. *Refactoring toolkit.*

The *refactoring* toolkit consists of the modules depicted in Figure 3, which are also written in TXL. The main module, shown at the top, implements the actual code transformation. Given an input source (e.g., `A.java`) and an interface to be migrated to an aspect (e.g., `I.java`), the TXL module *UNPLUG* produces a new source file (`A'.java`), in which the interface implementation is absent, and an aspect (file `IA.java`), which introduces the interface implementation into the original class. The aspect is also responsible for declaring that the class implements the interface (`declare parents` construct, see Section 2).

The other modules in Figure 3 are accessory modules that give specific information about interface aspectization. Module *UnpluggableFields* produces the list of fields that are accessed only from interface methods. Similarly, *UnpluggableInnerClasses* gives the inner classes not accessed outside the aspect being constructed. Finally, `InterfaceIsUnpluggable` gives information about the role of the aspect under construction. In case this module produces an output *yes*, the aspect will be regarded as an optional class feature, to be weaved with the class only upon request from the client. Otherwise, the aspect has to be considered an essential feature, because the methods it introduces are used also by other class methods.

In the development of the toolkit described above, some limitations of the current version of AspectJ have been identified. They prevented us from obtaining an implementation completely consistent with the approach presented in the previous sections. This is the list of the main problems encountered and of the workarounds we have adopted:

1. Private methods cannot be invoked (while private fields can be accessed) from methods introduced by an aspect.

2. Inner classes cannot be introduced by an aspect.

3. Genericity is currently not supported.

The workaround used for (1) was changing the visibility of methods where required. Problem (2) was resolved by bringing the inner class at top level with package-protected visibility. For (3), when possible, shared sub-computations introduced by super-aspects were introduced in a common super-class. Otherwise, aspect methods instead of introductions have been used.

7 Experimental results

In the following, we present the results obtained in the analysis and aspectization of some of the packages belonging to the Java standard library.

7.1 Data set

The possibility to identify and migrate interface implementations, that can be regarded as crosscutting concerns, has been tested on the java source code of three packages from the standard library of the Java 2 Runtime Environment, Standard Edition (build 1.4.0-b92). As shown in Table 1, the three analyzed packages contain 131 classes, for a total of 44 199 Line Of Code (LOC), comments excluded. The 52 interfaces implemented by these classes (179 implementations in total) have been subjected to aspect mining. Then, those representing actual crosscutting concerns have been migrated to aspects.

Package	**Classes**	**LOC**	**Int. impl.**	**Unique interfaces**
java.util	41	13,993	61	13
java.awt	76	24,425	97	35
java.awt.geom	14	5,781	21	4
Total	**131**	**44,199**	**179**	**52**

Table 1. *Data set under analysis.*

The first package (*java.util*) provides general purpose data structures, such as lists, sets, trees and hash tables. They offer a very popular programming framework, used by several software developers, who do not have to re-write their own implementation of these common data structures. They are used by classes in other library packages as well.

Because of the general purpose, all the classes in this package implement many functionalities that could be useful in software development (for example serialization, cloneability, etc.), although these may not be required in all uses. Moreover, they are typically not essential in defining the meaning of each class, while being useful to enrich it. Thus, we expect that several of them can be regarded as crosscutting concerns.

The *java.awt* package contains classes for creating graphic user interfaces and for painting graphics and images. The implementations of some layout managers and default event handlers are also included. Its sub-package *java.awt.geom* provides classes for defining and performing operations on two-dimensional objects. All these classes provide the general, basic behavior needed by advanced window-based environments (such as the *swing* environment), or by users who define their own one. Thus, similarly to the package *java.util*, these classes implement functionalities that are possibly required in a generic usage context.

Given the features of the analyzed packages, a better modularization of the code could be reached by separating the principal behavior of each class from the crosscutting functionalities added to increase generality and reuse. Such features are expected to be found in several other real-world Object Oriented systems, that support some degree of reuse. In this kind of systems, interface implementations are typically included to define a set of multiple access points for the external, user classes.

7.2 Aspect mining

In order to assess the results obtained with the proposed methods for aspect mining, a solution of trusted accuracy is necessary. An expert was involved in defining such a solution. He examined the class diagram, the source code and the Javadoc documentation in order to understand the role of each interface in each implementing class.

The output of the expert's analysis consists, for each package, of its principal decomposition, in which classes are grouped together with the implemented interfaces, when these are devoted to its main functionalities. Such interfaces are mandatory in defining the classes' predominant responsibilities. Interface implementations outside the principal decomposition are regarded as the expert's aspects.

Each of the four aspect mining methods has been applied to all the classes under analysis, giving a response about which interface implementations to change into an aspect. All the solutions have been compared with the (complement of the) expert's decomposition. They have been ranked in terms of *precision* and *recall*, where the former measures the proportion of correctly identified aspects over all the candidates produced by a given method, while the latter measures the proportion of correctly identified aspects over all those to be retrieved (i.e., those specified by the expert).

	External Package		**String Matching**	
	precision	recall	precision	recall
java.util	0.88	1.00	0.93	0.98
java.awt	0.91	0.91	0.89	0.87
java.awt.geom	0.50	1.00	0.82	0.93
Average	**0.86**	**0.95**	**0.86**	**0.94**
	Clustering		**Unpluggability**	
	precision	recall	precision	recall
java.util	0.73	0.95	0.82	0.90
java.awt	0.82	0.89	0.93	0.85
java.awt.geom	0.64	0.93	0.89	0.93
Average	**0.76**	**0.94**	**0.89**	**0.90**

Table 2. *Precision and recall of the four aspect mining methods.*

Table 2 shows the mean value of precision and recall computed for the classes contained in each package under analysis. The *average* values at the bottom are weighted by the number of classes in each package, since the considered

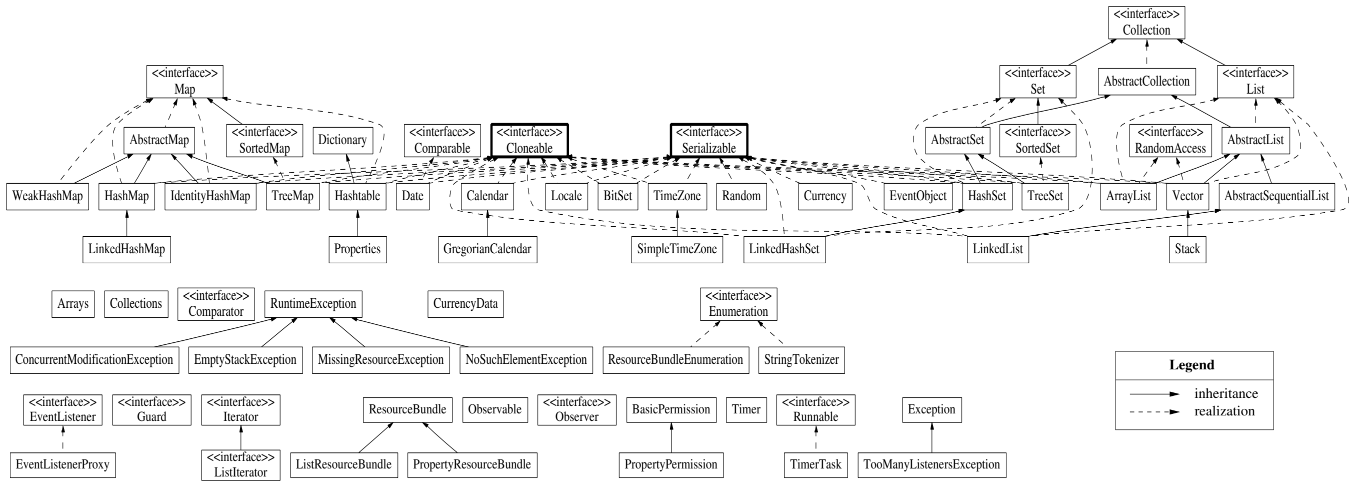


Figure 4. *Class diagram for the package* java.util. *Interfaces migrated to aspects are in bold boxes.*

packages belong to a same system, instead of being separate applications. In this way, the impact of small packages (such as *java.awt.geom*) on the final result is reduced.

The three methods *External Package*, *String Matching* and *Unpluggability* are fundamentally equivalent, having very similar values of precision and recall. The last one has the highest precision, but a slightly lower recall. The remaining method, *Clustering*, has a similar performance in terms of recall, but a much lower precision, making it substantially worse than the others.

A further investigation of unpluggability was obtained by running the modules *UnpluggableFields* and *UnpluggableInnerClasses*. Fields used only by interface methods have been detected by the module *UnpluggableFields* in 4 classes, for a total of 16 unpluggable fields. All of them are related to the implementation of the serialization concern (*Serializable* interface). Looking at the source code, we discovered that the purpose of these unpluggable data structures is to save specific state information, useful in a situation where complicated objects are restored from the serialized data.

The module *UnpluggableInnerClasses* detected 16 unpluggable inner classes. All of them are related to the accessibility concern (*Accessible* interface). In all these cases, we found a same pattern: an inner class specializes the abstract class *javax.accessibility.AccessibleContext*, used as return type by the accessibility interface method.

No field or inner class that the expert considered as a part of an interface implementation to be migrated was missed by our unpluggability modules (i.e., recall, as well as precision, is 100% for fields and inner classes to be aspectized).

7.3 Refactoring

Analyzing the whole data set, the expert tagged 106 interface implementations in 61 classes as best expressed through aspects. This means that about half of the classes can be subjected to aspectization of a part of the code.

Package	**Classes with aspects**	**Int. implem. aspectized**
java.util	16/41	28/61
java.awt	37/76	68/97
java.awt.geom	8/14	10/21
Total (percentage)	**61/131 (47%)**	**106/179 (60%)**

Table 3. *Refactored classes and interface implementations.*

Table 3 shows also the number of interface implementations that have been aspectized by means of our refactoring toolkit (in total 106/179, that is 60%). Classes that just inherit (without redefining) an interface implementation have not been counted.

Table 4 shows data related just to classes for which at least one crosscutting concern was detected. It shows the impact of the refactoring in terms of LOC reduction produced by the aspectization. Since interface implementations are moved from classes to aspects, in the principal decomposition, where aspects are excluded, a reduction of the code size can be observed. The mean value of such a reduction is around 10%. Moreover, data seem to indicate that larger reductions are obtained at increasing package size. A possible explanation is that crosscutting concerns play an

Package	Lines Of Code Original	Refactored	Δ
java.util	8,693	6,416	-6,9%
java.awt	16,744	14,773	-11,8%
java.awt.geom	4,098	3,902	-4,8%
Total	**27,735**	**25,091**	**-9,5%**

Table 4. *LOC change in the principal decomposition.*

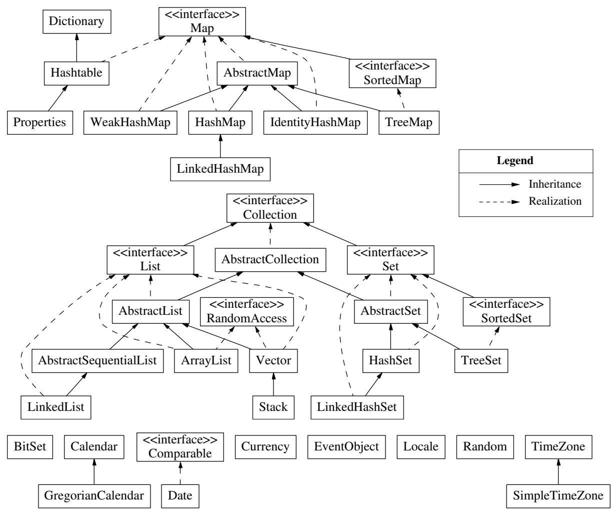


Figure 5. *Class diagram for the package* java.util, *restricted to the principal decomposition.*

important role in the increase of the size of a package. Packages are often enlarged with the addition of functionalities that crosscut the existing ones.

In the following, the effect of the refactoring on the understandabilty of the class diagrams is examined. Figures 4, 5 and 6 show how the class diagram changes due to our refactoring. Figure 4 represents the class diagram for the *java.util* package. It was built using a reverse engineering tool written in TXL. The layout was computed by the tool DOT [6]. Understanding the organization of the package from this diagram is a challenging task, mainly due to the high number of overlapping edges in the top region.

Figure 5 shows the principal decomposition for the package *java.util* (limited to the classes at the top in Figure 4), obtained by removing all interface implementations that represent a crosscutting concern and have been migrated to an aspect (see interfaces in bold boxes in Figure 4). The structure of this graph is much more understandable, thanks to a reduced number of edges. Its simplified layout reflects the way classes are organized, according to the main decomposition. 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.

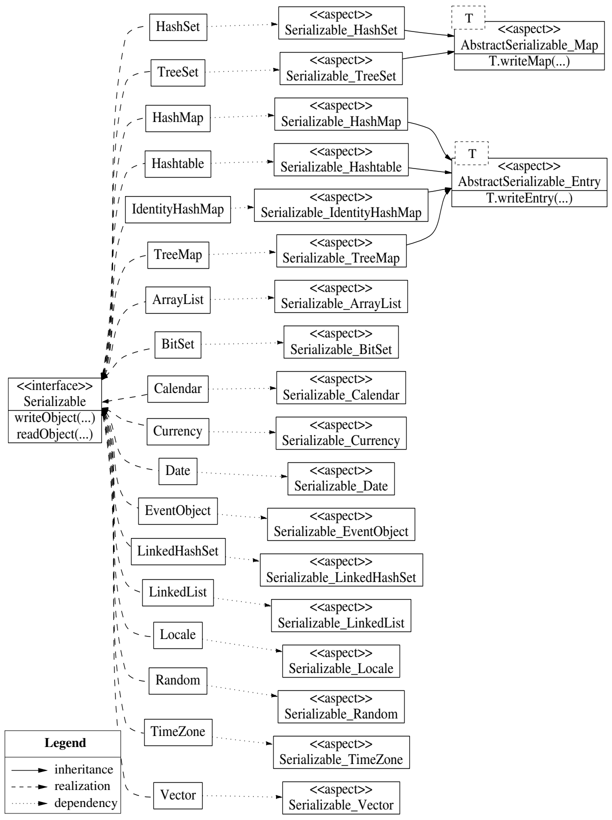

HashSet
<<aspect>> Serializable_HashSet
T
<<aspect>> AbstractSerializable_Map
T.writeMap(...)
TreeSet
<<aspect>> Serializable_TreeSet
HashMap
<<aspect>> Serializable_HashMap
T
<<aspect>> AbstractSerializable_Entry
T.writeEntry(...)
Hashtable
<<aspect>> Serializable_Hashtable
IdentityHashMap
<<aspect>> Serializable_IdentityHashMap
TreeMap
<<aspect>> Serializable_TreeMap
ArrayList
<<aspect>> Serializable_ArrayList
BitSet
<<aspect>> Serializable_BitSet
<<interface>> Serializable
writeObject(...)
readObject(...)
Calendar
<<aspect>> Serializable_Calendar
Currency
<<aspect>> Serializable_Currency
Date
<<aspect>> Serializable_Date
EventObject
<<aspect>> Serializable_EventObject
LinkedHashSet
<<aspect>> Serializable_LinkedHashSet
LinkedList
<<aspect>> Serializable_LinkedList
Locale
<<aspect>> Serializable_Locale
Random
<<aspect>> Serializable_Random
TimeZone
<<aspect>> Serializable_TimeZone
Legend
inheritance
realization
dependency
Vector
<<aspect>> Serializable_Vector

Figure 6. *Aspect diagram for the package* java.util, *representing the serialization concern.*

Figure 6 shows one of the crosscutting concerns (serialization) in the package *java.util*, as implemented by the aspects introduced by the proposed refactoring. The realization relationships that connect each implementing class to the interface *Serializable* are dependent on the corresponding aspect that introduces the interface methods into the class (dependency relationship in Figure 6). This means that interface implementation is conditional to the inclusion of the respective aspect in the compiled code.

In the new code organization, the serialization concern is handled separately, thanks to AOP. In the design reverse engineered from the code, serialization, as well as all the other crosscutting concerns, are represented in a distinct view, dedicated to a single concern only. This results in an increased modularization of the design and simplifies the comprehension of each crosscutting concern. For example, if we consider the serialization concern, Figure 6 gives a clear representation of the affected classes and of the in-

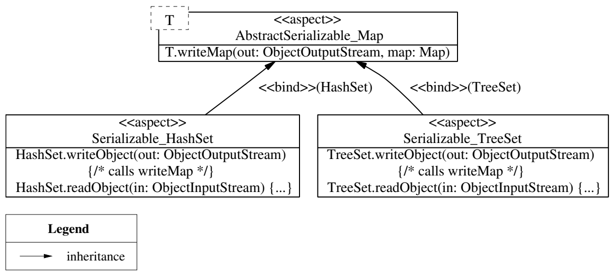

T
<<aspect>>
AbstractSerializable_Map
T.writeMap(out: ObjectOutputStream, map: Map)
<<bind>>(HashSet)
<<bind>>(TreeSet)
<<aspect>>
Serializable_HashSet
HashSet.writeObject(out: ObjectOutputStream)
{/* calls writeMap */}
HashSet.readObject(in: ObjectInputStream) {...}
<<aspect>>
Serializable_TreeSet
TreeSet.writeObject(out: ObjectOutputStream)
{/* calls writeMap */}
TreeSet.readObject(in: ObjectInputStream) {...}
Legend
inheritance

Figure 7. *Example of abstract, generic aspect introduced for the serialization concern.*

volved aspects.

A further advantage of the separation of the crosscutting concerns is that common code can be factored out into super-aspects. As shown in Figure 6, for the serialization concern this happened in two cases. The two aspects *Serializable_HashSet* and *Serializable_TreeSet* share a method, *writeMap*, implemented exactly in the same way in the two introductions. Consequently, this method can be moved to a common super-aspect, named *AbstractSerializable_Map*. A similar situation occurs with the aspects *Serializable_HashMap*, *Serializable_Hashtable*, *Serializable_IdentityHashMap* and *Serializable_TreeMap*, containing a same introduction, *writeEntry*, moved to the super-aspect *AbstractSerializable_Entry*.

Since the target class of the introduction performed in the super-aspect depends on the specific derived aspect, a generic introduction with type parameter T is performed (see also Figure 7 for *AbstractSerializable_Map*). Each sub-aspect binds this parameter to the specific class where method *writeMap* has to be introduced. Thus, *Serializable_HashSet* binds T to *HashSet*, while *Serializable_TreeSet* binds T to *TreeSet*. A similar binding occurs for the aspects descending from *AbstractSerializable_Entry*.

As indicated in Figure 7, the introduction *writeMap* inherited from *AbstractSerializable_Map* is used inside one of the two methods required by the interface *Serializable*, namley *writeObject*, which performs an invocation to such common method.

Modularization of common aspect code into abstract, generic aspects is an advantage offered by AOP that can be hardly achieved in standard Object-Oriented programming, when the shared code belong to a crosscutting concern. Actually, in the original code the methods abstracted into a super-aspect (such as *writeMap* and *writeEntry*) were duplicated, with all the well-known problems that the practice of code cloning poses.

In total, in our experimental data-set we found 5 abstract aspects for 26 aspects containing code clones that are easily factored out once aspects are introduced.

A possible measure of the effects that migration to AOP has on the class diagram is given by the reduction in the number of edges. As described above with reference to Figures 4, 5, a reduction in the number of edges is often associated with a more understandable diagram, where the main hierarchies in the packages, previously obscured by the interface implementations, come to light.

	Number of edges		
Package	**Before**	**After**	**Δ**
java.util	81	55	-31%
java.awt	126	79	-37%
java.awt.geom	21	11	-48%
Total	**228**	**145**	**-36%**

Table 5. *Impact of refactoring on the complexity of the class diagram for the principal decomposition.*

Table 5 gives such a measure for the packages under analysis. On average, the new principal decomposition contains 36% edges less. The related visual cluttering is correspondingly reduced.

	Lines Of Code		
Interface	**Before**	**After**	**Δ**
LayoutManager2	67	46	-31,3%
LayoutManager	198	199	+0,5%
Shape	152	148	-2,6%
Cloneable	77	80	+3,9%
Serializable	532	501	-5,9%
Total	**1,026**	**974**	**-5,1%**

Table 6. *Impact of the introduction of super-aspects on the size of the aspect code.*

Table 6 gives the LOC of the aspects containing replicated computations, before and after introduction of super-aspects. For the concerns *LayoutManager* and *Cloneable*, the amount of code removed is smaller than the overhead of the abstract aspects due to the headers, so size actually increases instead of shrinking. In the general case, factorization has a positive effect on the code size, with an average reduction of around 5%. The overall effects on understandability and maintainability of the clone-free code are expected to be even more relevant.

8 Conclusions and future work

Experimental results indicate that the problem of identifying which interface implementations are suited for migration to aspects can be approached with simple methods (such as string matching), which are able to produce a very good starting point. Successive manual refinement is however still necessary.

The introduction of aspects for interface implementations impacted a large number of classes and interfaces. It produced a moderate reduction in the LOC (around 10%), which, however, resulted in a major simplification of the class diagrams (around 36% edges less). Visual inspection of such diagrams confirmed a reduction of their complexity. In the refactored code, interface implementations are no longer scattered and a better modularization is reached.

Code duplications that are otherwise difficult to factor out have been identified and moved to super-aspects. This has potentially a tremendous impact on the maintainability of the code (changes are located easily, just one code entity is modified, code updates are not missed due to a spread implementation of a same functionality) and on its testability (coverage testing is simplified, test cases can be modularized according to principal decomposition and crosscutting concerns, code portions to re-test are easily located).

The AOP language AspectJ has been chosen as the target of our refactoring. While working with AspectJ, we identified some improvement areas that we suggest for consideration in the development of the next versions of the language. Genericity should be supported in a similar way as in the new version (1.5) of Java. Introduction of inner classes should be allowed and access to private methods should be granted to introductions.

In our future work, we will consider crosscutting concerns that are not necessarily interface implementations. Clone detection techniques could be employed for their identification. We are also interested in conducting studies focused on specific interfaces that are known in advance to be good candidates for aspectization. Another interesting future direction consists of an empirical assessment of the impact of aspectization on maintainability.

References

[1] E. L. A. Baniassad, G. C. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: an inquisitive study. In *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, pages 120–126, Enschede, The Netherlands, April 2002. ACM press.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the International Conference on Software Maintenance*, pages 368–377, Bethesda, Maryland, USA, November 1998.

[3] P. Borba and S. Soares. Refactoring and code generation tools for AspectJ. In *Proc. of the Workshop on Tools for Aspect-Oriented Software Development (with OOPSLA)*, Seattle, Washington, USA, November 2002.

[4] J. Cordy, T. Dean, A. Malton, and K. Schneider. Source transformation in software engineering using the TXL transformation system. *Information and Software Technology*, 44(13):827–837, 2002.

[5] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Publishing Company, Reading, MA, 1999.

[6] E. Gasner, E. Koutsofios, S. North, and K.-P. Vo. A technique for drawing directed graphs. In *IEEE Transactions on Software Engineering*, March 1993.

[7] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Proc. of Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering (ICSE)*, Toronto, Canada, 2001.

[8] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proc. of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173, Seattle, Washington, USA, November 2002. ACM press.

[9] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *Proc. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 178–187, Boston, Massachusetts, USA, March 2003. ACM press.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *Proc. of the 11th European Conference on Object Oriented Programming (ECOOP), vol. 1241 of LNCS*, pages 220–242. Springer-Verlag, 1997.

[11] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, Indiana, USA, 2002.

[12] N. Loughran and A. Rashid. Mining aspects. In *Proc. of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (with AOSD)*, Enschede, The Netherlands, April 2002.

[13] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. of the International Workshop on Program Comprehension*, pages 45–52, Ischia, Italy, 1998.

[14] A. Rashid and R. Chitchyan. Persistence as an aspect. In *Proc. of the 2nd International Conference on Aspect-Oriented Software Development*, pages 120–129. ACM Publisher, 2003.

[15] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proc. of the 24th International Conference on Software Engineering (ICSE)*, pages 406–416, Orlando, FL, USA, May 2002. ACM press.

[16] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proc. of the 17th Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 174–190, Seattle, Washington, USA, November 2002. ACM press.

[17] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton Jr. N degrees of separation: Multi-dimensional separation

of concerns. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 107–119, Los Angeles, CA, USA, May 1999. ACM press.

[18] A. van Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE), with WCRE*, Waterloo, Canada, November 2003.