2008 20th IEEE International Conference on Tools with Artificial Intelligence

Mining Functional Aspects From Legacy Code

Amal Elkharraz, Hafedh Mili, Petko Valtchev
LATECE Laboratory, Université du Québec á Montréal
Montréal, Canada
ekharraz@iro.umontreal.ca, {hafedh.mili, petko.valtchev}@uqam.ca

Abstract

Aspect-oriented software development builds upon object-oriented (OO) development by offering additional abstraction boundaries that help us separate different types of requirements into so-called aspects. Given a legacy OO application, it pays to identify existing aspects to help understand the structure of the application and potentially to extract those aspects into separate software modules that can be maintained and reused individually. We are interested in the extraction and repackaging of functional aspects. We first characterize what we mean by functional aspect, and then explore the properties that such aspects are likely to exhibit in a legacy application that integrate such aspects. Next, we present algorithms for extracting such aspects, and the preliminary results we obtained. We conclude by discussing directions for future research.

1 Introduction

Software applications typically embody a complex web of requirements, both functional ones related to the implemented ⟨input, output⟩ relation, and non-functional ones, related to how that output is produced. A good software abstraction and packaging technique is one that enables us to implement different requirements in distinct software artifacts that we can develop, maintain, and compose at will [13]. Aspect-oriented software development (AOSD) builds upon the abstraction and packaging techniques inherent in OO software by proposing artifacts that enable us to untangle requirement types that OO abstractions could not. Given a legacy OO application, it pays to identify and to try to isolate code fragments that implement a particular requirement. Minimally, this can help in understanding the application and maintaining it by delineating those parts that are affected by a change in requirements. In some cases, it may even be possible to repackage those parts using OO refactoring [6] or AOSD-like refactoring [14] so they can be reused and composed with other programs. Our work deals with aspect mining in legacy Java code.

It is customary in the AOSD community to make a broad distinction between two families of aspects: one derived from subjects [16] and one derived from AspectJ [5]. Although both families can implement any kind of requirement, the latter aspects are often used to implement design level requirements such as persistence, distribution, security, etc., whereas the former tend to implement pluggable features that depend on core functionality [10]. We are interested in the identification and packaging of subject-like aspects implementing functional requirements. Such aspects embody often undocumented domain knowledge that is buried in legacy application code.

Our intuition is that the elements of an OO application contributing to a functional aspect (classes and their features) will exhibit stronger "cohesion" than the elements of the application as a whole whereas the sets of elements that implement different functional aspects will exhibit lower "coupling" than randomly selected subsets of classes and their features, however coupling and cohesion are defined in this case. In one method, we used code slicing to extract the classes, data members, methods– and statements in such methods– that contributed to the return value of a function within that aspect, and the results are promising [3].

In section 2, we provide our view on functional aspects, and explore the properties that the elements thereof (classes, methods, attributes) are likely to exhibit. In section 3, we examine the extent to which various techniques, e.g., lattice decomposition in concept analysis, can be used to extract functional aspects. We report the results of our preliminary experiments in section 4, and conclude in section 5.

2 Characterizing functional aspects

2.1 What is a functional aspect

By *functional aspect*, we mean a slice or "subset" of an OO application that addresses a cohesive subset of functional (domain) requirements. We expect any OO application of some complexity to embody several such 'as-

1082-3409/08 $25.00 © 2008 IEEE
DOI 10.1109/ICTAI.2008.149

pects'. Our definition is closest to what Harrison and Ossher described as a *subject* [10]. With traditional OO programming, all of these functionalities will be implemented within the same class hierarchy. The idea of subject-oriented programming– and its descendant [16]– is to represent each functional viewpoint in its own class hierarchy called *subject*, which can be developed and maintained separately, and composed at will.

The motivation and intuition behind Kiczales et al.'s *aspects* [5] are markedly different from that behind Harrison and Ossher's subjects. In Kiczales et al.'s model, aspects are used to package cross-cutting concerns that pervade OO applications. Such concerns are *typically* architectural or design-level concerns (non-functional requirements), that may not affect the functional ⟨input,output⟨ relationship implemented by the software, but influences the way that output is produced. In this regard, the aspects are somewhat *orthogonal* to the underlying, (functional) class hierarchy– called the *dominant decomposition* in [16]–, are expressed in a separate non-object-oriented *aspect-language*, and are integrated using an asymmetrical composition algorithm (aspect *weaving*). While one could arguably express any functional aspect using aspect-oriented programming *a la* AspectJ, we felt that the resulting aspects would not exhibit the conceptual unity found in subjects, and would probably be difficult to mine or reverse-engineer out of legacy code.

2.2 Reverse-engineering functional aspects

Given a legacy OO application that combines several functional aspects, our work deals with exploring ways of uncovering and untangling those aspects from the overall application. Our intuition is that the elements that compose a functional aspect (classes, methods, and attributes) would exhibit greater cohesion to each other, than to the elements of the application as a whole. Elsewhere, we used program slicing techniques to slice-out fragments of Java class hierarchies that contribute to the computation of some desired result[1], and the preliminary results were promising [3]. Here, we explore the use of conceptual clustering to identify aspects within the overall class hierarchy.

One way of characterizing functional aspects in legacy applications consists of doing the following:

1. Identify the various non-aspect oriented techniques that could be used to implement and compose several functional aspects in the same class hierarchy,
2. For each such technique, study the way in which it transforms the original functional aspect, and
3. To the extent that the original functional aspect might still be recognizable in the overall application, develop techniques to extract them.

[1] called *slicing criterion* in the program slicing literature

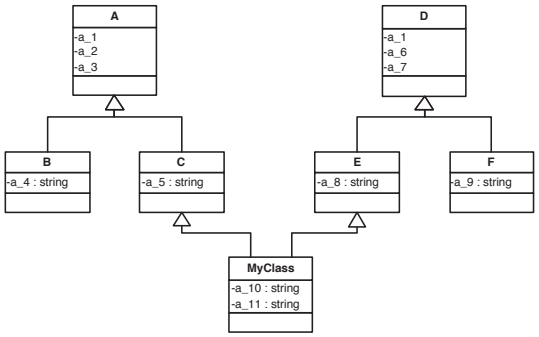


Figure 1. Implementing several functional aspects using multiple inheritance

In what follows, we examine three techniques:

- *Multiple inheritance*. Each functional aspect is represented using its own class hierarchy hence a class combining several aspects simply inherits from the corresponding class hierarchies.
- *Aggregation*. Again, a separate class hierarchy represents each functional aspect. Combining several aspects amounts for a class to aggregating the classes implementing the individual aspects.
- *State multiplication*. No care is taken to separate the functional aspects: They are implemented directly in the classes that use them.

Naturally, when studying a legacy application, it is not know beforehand whether the application embodies different functional aspects, and if it does, which of the above techniques has been used. Both the presence/absence of functional aspects, and the composition technique, are hypotheses to be tested by applying aspect mining.

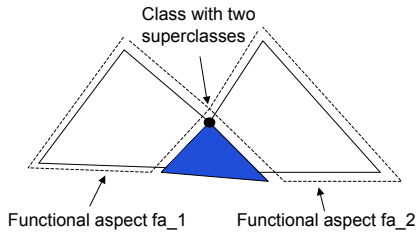


Figure 2. Separating disjoint functional aspects

2.3 Uncovering aspects composed with multiple inheritance

2.3.1 Basics

Figure 1 illustrates the case of each functional aspect being developed in one class hierarchy. Here, a class integrating several functional aspects inherits from one class from each

hierarchy. The simplest of all cases is depicted here, as i) the functional hierarchies have no common root in the upper levels of the hierarchy, ii) the classes of interest inherit from leaf nodes in the various functional hierarchies, and iii) they are themselves leaf nodes.

At first glance, this situation is easily recognizable: simply identify instances of multiple inheritance, i.e. classes in the hierarchy with two or more immediate ancestors[2]. Next, we need to delineate the parts of the class hierarchy that would correspond to individual functional aspects. Intuitively, those parts will need to be disjoint. Further, we take the largest (in the inclusion sense) such disjoint parts. As an illustration, consider class C in Fig. 2 whose superclasses come from two separate hierarchies. Removing C and descendants yields two disjoint hierarchies each representing an aspect.

This approach generalizes to an arbitrary number of ancestors for C, belonging to distinct hierarchies. Hereafter, we will focus on the identification of instances of multiple inheritance for languages that *do not* support such mechanism, by considering ways that programmers "fake" (or implement) multiple inheritance. This is discussed below.

2.3.2 Case with single-inheritance languages

Many programming languages do not support multiple inheritance due to the program design and *language design* issues it rises. Implementing design-level multiple inheritance within a language that does not support it, is a delicate task. A few techniques used by Java developers are examined below (illustrated in Fig. 3).

Java distinguishes between types– and subtyping– and classes– and implementation inheritance. In the first case (Fig. 3.a) a class implements two interfaces, but reuses no implementation. In Fig. 3.b, a class inherits from another class that embodies a first functional aspect– and hence, reuses its implementation– and implements another interface, representing a second functional aspect. In the third case (Fig. 3.c), the class `PartTimeStudent` supports the functionalities of `Student` and `Worker` through a combination of implementation inheritance, and delegation (discussed in section 2.4). These cases are by no means exhaustive. First, a class may implement three or more aspects instead of two as shown here. Second, variations are possible: For example, for Fig. 3.a, it is clear just by looking at the class `PartTimeStudent` that it implements two functional aspects. We could have a case, instead, where a designer has already defined an interface called `StudentWorker`, say, that inherits from both interfaces `Student` and `Worker`. Hence we need to dig in the interface hierarchy as well for cases of multiple inheritance. Third, these cases represent a somewhat disciplined use of interfaces. Indeed, for Fig. 3.c, a novice designer may not bother to define an interface for `Worker`, or specify that the class `PartTimeStudent` implements it. Consequently, an algorithm for multiple inheritance detection should reflect all these considerations.

Such an algorithm is also likely to identify many situations that *do not* represent legitimate cases of multiple functional aspects. Indeed, the different "functionalities" may not all be *domain* functionalities, and many will turn out to be infrastructural or utility-like aspects. Indeed, interfaces are *often* used to add functionality to domain objects so that they support a "service-contract" for an infrastructural aspect. For example, in the EJB 2.x architecture, a session bean class had to implement the `SessionBean` interface to benefit from the services provided by the host "J2EE container". Similarly, to be able to compare objets, their class has to implement the `Comparable` interface. We also use interfaces to represent constants, and have domain classes implement those interfaces so that the constants can be referenced with no qualification. Finally, there are so-called *marker interfaces* that include no methods, but that "mark" classes so that are treated in a particular way by other parts of the Java API (e.g. `Serializable`).

This raises the issue of recognizing interfaces representing legitimate functional aspects. There is no simple or single answer. We will need to rely on a combination of our knowledge of the Java API and some heuristics. For example, if we are trying to identify functional aspects within an MIS application, interfaces from the basic Java API (e.g. packages `java.*`, `javax.*`) and related utilities (e.g. `org.omg.*`, `org.w3c.*`) can be safely ignored. Conversely, with a graphical editor, all the AWT (`java.awt.*`) and SWING (`javax.swing.*`) interfaces are legitimate. Additional heuristics may consider the *kinds* of interfaces. Marker interfaces and those made of constants can be ignored. So can single-method interfaces. We expect that experiments will help us ground and refine such heuristics.

2.4 Uncovering aspects composed with delegation

Delegation is another common design technique that can support multiple functional aspects (see Fig. 3.c for an example). The term "delegation" has a precise meaning in delegation-based languages[3]. Here, we use it in the following looser sense: class A delegates to class B iff:

1. class A defines an attribute of type B– call it b;
2. class A implements the behavior of B, and
3. the A implementation of a B method forwards the call to the attribute b.

[2] Actually, we need to look at the highest such nodes in the hierarchy

[3] In these languages individual objects (*prototypes*) rather than classes store behavior specifications: new objects can *delegate* behavior to existing ones who execute their *own* method *in the context of the delegator*.

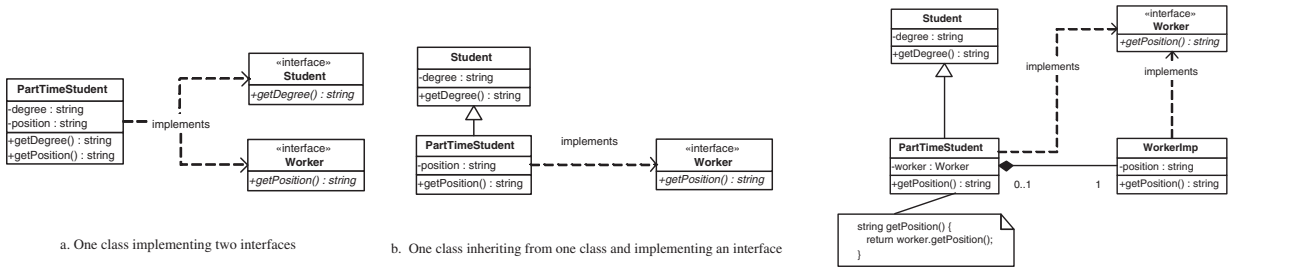


Figure 3. Implementing design-level multiple inheritance in Java: some patterns

In Fig. 4.a, the class `ImmigrantPartTimeStudent` delegates to both `WorkerImp` and `ImmigrantImp` since, a) it has the attributes `worker` of type `Worker`, and `immigrant` of type `Immigrant`, respectively, b) it implements the interfaces `Worker` and `Immigrant`, and c) its implementation of the various methods of the interfaces return the results of calling the corresponding methods on the attributes `worker` and `immigrant`, respectively[4].

Fig. 4.b shows an "undisciplined" use of delegation. In this case, the fact that `ImmigrantPartTimeStudent` implements the behavior of `Worker` or `Immigrant` is only implicit in its API, as opposed to the implementation of a common interface. For our purposes, this requires the examination of the methods implemented by each class to ensure that it implements the behavior of its delegates. In practice, it is unlikely that *all* of the methods of the delegates– here, classes `Worker` and `Immigrant` – are implemented by the delegator (`ImmigrantPartTimeStudent`). Fig. 4.b shows that class `ImmigrantPartTimeStudent` implements / delegates *only one* of `Immigrant`'s methods, namely `isAuthorizedToWork()`. An algorithm that detects instances of delegation in legacy code will have to contend with partial interface implementation. This raises the question of "how partial". One could use a threshold, hence condition 2 in our delegation definition must be replaced by:

2'. class A implements at least $\alpha\%$ the behavior of B

However, such a heuristic is likely to produce many false-positives and false-negatives, for two reasons. First, domain classes will typically have a few domain methods, and lots of utility methods (constructors, accessors, serializers, hashers, etc.). Thus, a percentage alone is not indicative: it depends on which methods are being delegated. Second, a delegator need only reuse / delegate *one* aspect– read, method– to make it a legitimate delegation instance.

Like multiple inheritance, the legitimacy of delegation depends also on what is being delegated to. In Fig. 4, the delegator and the delegates are clearly domain classes[5]. We should distinguish those from instances where a domain class delegates to a utility class. A typical example is when a class uses a *collection* to refer to associated objects: one would typically add delegate methods to enumerate the elements of the collection, or to add an element to the collection. The following code excerpts illustrate this:

```
class Personnel {
  private Collection members;
...
Iterator members() {
return members.iterator();
}
...
public Object add(Employee emp) {
return members.add(emp);
}
...
}
```

Here, the `Personnel` class and the `Collection` class/interface do not form a delegation instance of interest. The situation illustrates yet another potential problem within legitimate instances of delegation: a developer might rename a delegated method in the delegator. In our example, the `iterator()` method of `Collection` has been renamed `members()` in the class `Personnel`. Thus, we may need to relax our definition of a method delegation: we should ignore method names, and focus on their types (return type, parameter types).

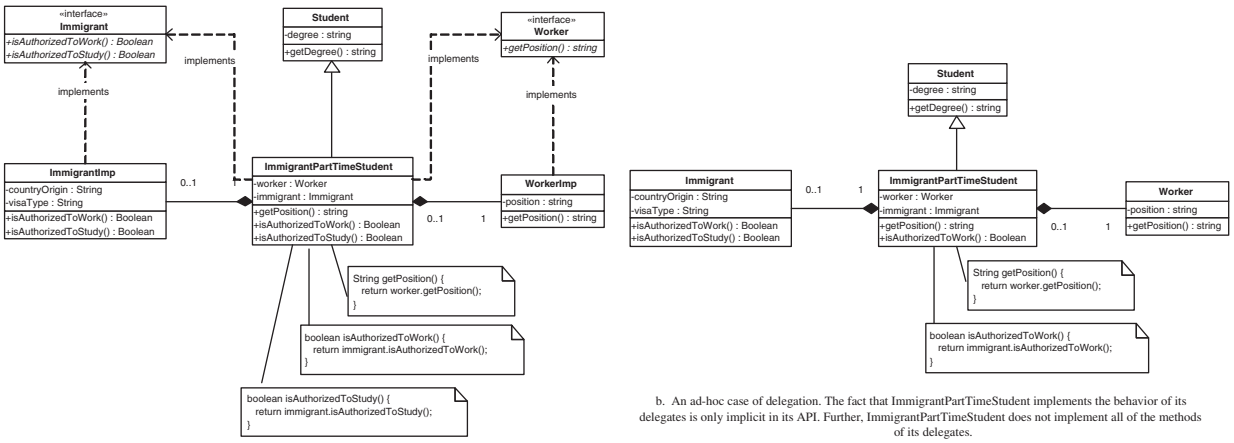
A delegation detecting algorithm should take into account all these variations. Like in the case of multiple inheritance, it should use a combination of general-purpose heuristics with some knowledge of the application domain and of the Java API. Again, only experimentation will help us define and refine these heuristics.

2.5 Uncovering aspects composed with state multiplication

'State multiplication' means different functional aspects are embedded in the same class with no special artifact.

[4]This departs from the "standard" delegation since `worker`, for example, executes `getPosition()` in *its own* context, as opposed to the context of the forwarding `ImmigrantPartTimeStudent` instance.

[5]They rather *belong to the same domain* since, as indicated before, if the application domain is GUIs, delegates from `java.awt.*` or `javax.swing.*` represent legitimate delegation instances.



a. A disciplined case of delegation. One class inheriting a functional feature from another class, and delegating two additional functional features to two other classes.

b. An ad-hoc case of delegation. The fact that ImmigrantPartTimeStudent implements the behavior of its delegates is only implicit in its API. Further, ImmigrantPartTimeStudent does not implement all of the methods of its delegates.

Figure 4. Supporting multiple functional aspects with delegation: two patterns

This is best understood in the context of a situation where we want to combine several features, each of which comes in different flavors. Let us consider the example of cars. Cars come in many body types, including sedans, coupe, and station wagon. Cars can also have various power plants. Figs. 5.a and 5.b show what hierarchies of chassis, and power plants, might look like. A given car will thus have a combination of these two features. Figs. 5.c and 5.d show two possible car classifications. To clarify our use of the term "state multiplication", consider the class `Car`, which in both figures 5.c and 5.d includes the sum of the "state variables" of the chassis and engine components (all combinations are present). Incidentally, note that the hierarchies in Figs. 5.a and 5.b show a special case of classification: the extension of the attribute set. Naturally, a subclass can also redefine (specialize) inherited methods.

How would such an implementation arise? Most likely, if a developer– even an inexperienced one–is given the `Chassis` and `Engine` type hierarchies *beforehand*, she/he will try to apply them, by using either aggregation or multiple inheritance. The hierarchies in Figs. 5.c and 5.d would probably result from an incremental specialization, i.e. starting with the first feature of the problem space (engine type in Fig. 5), and then specializing the leaf nodes of the class hierarchy based on the second feature. Also, an novice developer who is given all four combinations of features at once might still propose such a solution.

Regardless of how such a hierarchy would come about, a common symptom of state multiplication is recurring specialization patterns found in different places of the class hierarchy. This manifests itself with the occurrence of some class features (attributes or methods) in different places in the class hierarchy without being properly factored. In Fig. 5.d, the attribute `compressionRatio` is defined in two leaf nodes of the hierarchy. If we want to refactor it so that, a) it is defined in one place, and b) it is accessible to both `TurboI4Sedan` and `TurboI4StationWagon`, it should go to the class `Car`. But a `compressionRatio` makes no-sense for a non-turbo engine (V6). This is usually the dilemma of inexperienced or inattentive designers when working with single inheritance languages.

Graphically, a case of state multiplication would look like some product of the underlying feature hierarchies. Assume that we represent the feature hierarchies with partial orders F_1 = (V_1,R_1), and F_2 = (V_2,R_2), where V_1 and V_2 represent the set of values of F_1 and F_2, respectively, and R_1 and R_2 the partial orders defined on those values. The *cartesian product* of F_1 and F_2 is defined as follows:

$$F_1 \otimes F_2 \equiv (\ V_1 \times V_2,\ R_{1\otimes 2}\ )$$

where $R_{1\otimes 2}$ is defined as follows:

$$(\ \forall \langle x,y\rangle and \langle x',y'\rangle such\ that\ x,x' \in V_1\ and\ y,y' \in V_2)\langle\langle x,y\rangle,\langle x',y'\rangle\rangle \in R_{1\otimes 2}) \Leftrightarrow ((((\langle x,x'\rangle \in R_1) \wedge (y = y')) \vee ((x = x') \wedge (\langle y,y'\rangle \in R_2)))$$

The cartesian product of the features in Figs. 5.a and 5.b is shown in Fig. 6. As shown in [1] cartesian products can be recognized in $O(m\log(n))$, where m and n are the numbers of edges and of vertices in the graph, resp.

A more general product is the *tensor* (or *direct*) *product*, where the relation $R_{1\otimes 2}$ is:

$$(\ \forall \langle x,y\rangle and \langle x',y'\rangle such\ that\ x,x' \in V_1\ and\ y,y' \in V_2)(\langle\langle x,y\rangle,\langle x',y'\rangle\rangle \in R_{1\otimes 2}) \Leftrightarrow ((\langle x,x'\rangle \in R_1) \wedge (\langle y,y'\rangle \in R_2))$$

In the tensor product, *both* features are specialized in a subclass. If the relations R_1 and R_2 are reflexive, the reader can check that the tensor product *includes* the cartesian product.

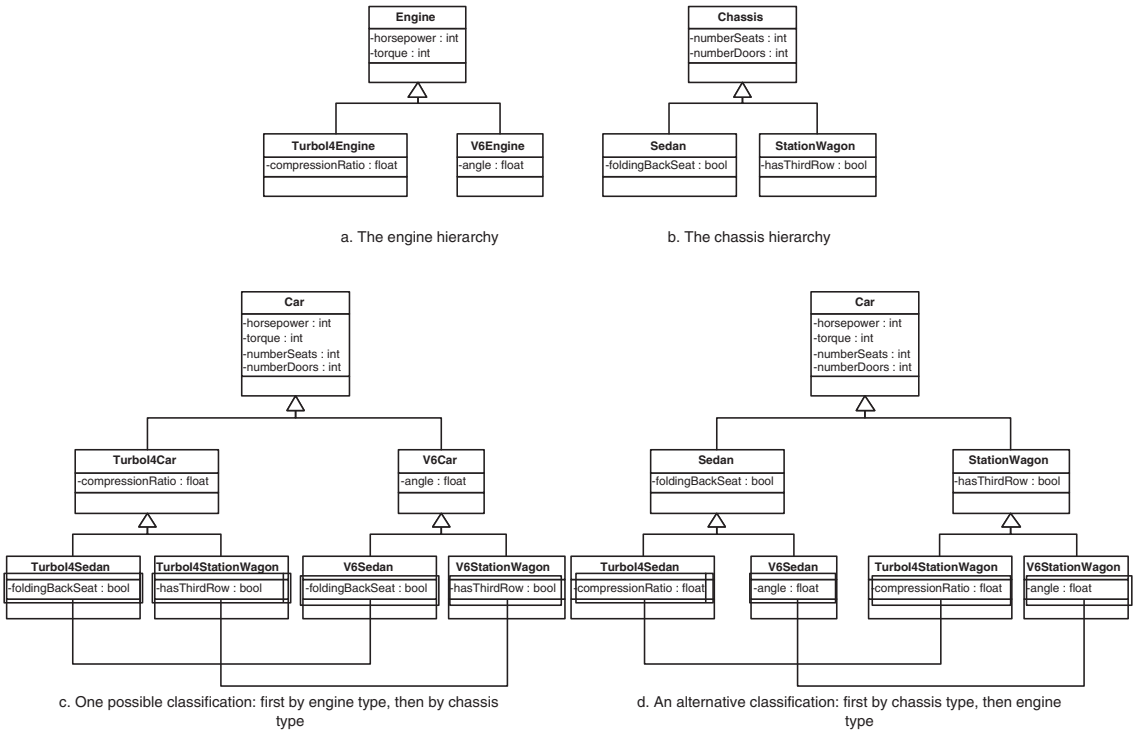


Figure 5. Supporting multiple functional aspects with state multiplication

Imrich developed a polynomial-time algorithm that recognizes tensor product graphs and finds the factors [12].

In real-world applications, class hierarchies are unlikely to match exactly cartesian or tensor products. For example, Figs. 5.c and 5.d represent *subsets* of the cartesian product in Fig. 6. consequently, some instances of "state multiplication" won't be detected by exact factorization algorithms.

We argue that real-life instances of state multiplication will fail these perfect factorizations, for two reasons:

1. *The sparseness of the domain*. In Fig. 6, all combinations of chassis type and engine are possible and of interest. In real life, some of them may not be *technically possible* or simply, not interesting: Assume a marketer argues that the customer base for a station wagon care little for turbo-compressed engines.

2. *Less than perfect factorization with the combinations at hand*. This failure is not related to the business domain, but to the developers' encoding of the domain information they are given.

Thus, instead of applying the factorization algorithms directly to the class, we chose to build an intermediary structure that abstracts away the developer's possibly imperfect encoding of the problem domain, so that we can focus on the domain structure. In particular, we build *Galois lattices* based on the original class hierarchies, which are maximally factored lattice structures (see e.g. [8]). Next, we perform a number of analyses on the resulting lattice and the underlying binary relation. Among others, one can apply specific decompositions such as *subdirect* one [7] (see below).

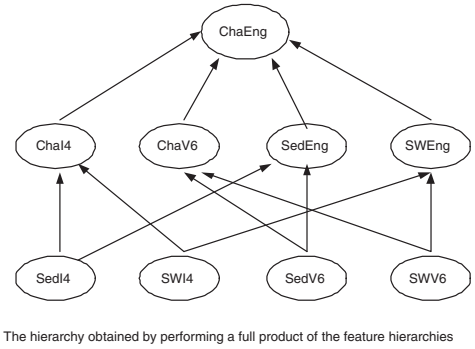


Figure 6. The full product of feature hierarchies ("Cha","Eng", "Sed", and "SW", stand for Chassis, engine, Sedan, and Station Wagon, resp. while "I4" and "V6" are engine types).

3 Functional aspect extraction algorithms

As mentioned in section 2.2, in the absence of AOSD techniques, developers have to resort to traditional OO pro-

gramming and design techniques to implement multiple functional aspects/features. In particular, we identified three such techniques: a) multiple inheritance– possibly faking it, b) delegation through aggregation, and c) what we called state multiplication. In what follows, we discussed issues involved in discovering instances of these techniques and present effective *algorithms* therefor. While for state multiplication, our work is still exploratory, the first two methods have been experimentally validated (see the next section).

3.1 Identifying instances of multiple inheritance in Java programs

As Java does not support multiple inheritance, we focus on ways that programs would fake it with Java constructs, here the use of interfaces. The algorithm for identifying classes with multiple inheritance is given below:

foreach *class C in the program P:*

1. *Let SUPER(C) $\longleftarrow$ $\{ superclass(C) \} \bigcup \{ the\ interfaces\ implemented\ directly\ by\ C \}$*
2. *Remove from SUPER(C) all interfaces from Java API*
3. *Remove from SUPER(C) all marker interfaces*
4. *Remove from SUPER(C) all interfaces of constants*
5. *If $|\ SUPERC\ (C)| \geq 2$, mark class C as a candidate class implementing several aspects*

As to the removing the interfaces from Java API, the list of packages to exclude depends on the application being analyzed. If we are analyzing a graphical framework, for example, interfaces from the `java.awt.*` or `jaxax.swing.*` packages are relevant to the domain.

Notice that this algorithm may return pairs of classes C_1 and C_2 such that SUPER(C_1) = $\{superclass(C_1),\ I_1\}$, for some interface I_1, and SUPER(C_2) $=$ $\{C_1,\ I_2\}$, where I_2 is a subtype of I_1. In others words, C_2 specializes C_1, and implements a specialization of the interface implemented by C_1. Here, we don't consider these cases are separate, and for the purposes of identifying potential functional aspects, we consider superclass(C_1) and I_1 as being representative of those aspects. Our current implementation does not automate the filtering of candidate classes: this redundancy came up in our experimental results (discussed in section 4).

3.2 Identifying instances of aggregation

The aggregations detection algorithm focuses on the functions implemented by classes and compares them to the methods implemented by their attributes regardless of the implemented interfaces. Referring back to Fig. 4, we look for cases of ad-hoc delegation (Fig. 4.b), which are weaker than disciplined delegation (Fig. 4.a). This enables us to catch both through the algorithm below:

foreach *class C in the program P:*

1. *Let $DOMINT(C) \longleftarrow \{\ m\ domain\ method\ of\ C \mid signature(m)\ \}$*
2. ***foreach*** *attribute at of C whose type T is not in Java API:*
   - *(a) compute $DOMINT$(type(at))*
   - *(b)* ***if*** *$DOMINT(C) \bigcap DOMINT(at) \neq \Phi$* ***then***
     - *i.* ***if*** *for any signature in $DOMINT(C) \bigcap DOMINT(at)$, the corresponding methods delegate to each other,* ***then*** *mark class C as potentially implementing $DOMINT$(type(at))*

Here, DOMINT (C) represents the *domain interface* of class C, i.e. its full interface minus accessors and methods from Java API (e.g. `clone()`, `hash()`, `toString()`, etc.). DOMINT(at) for an attribute at is a short for the domain interface of the type of at. Also, the signature of a method m is defined as a pair ((Inp_1,Inp_2,...,Inp_n),Out), where Inp_i is the type of the i^{th} input parameter of m, and Out is the type of its output parameter. We are ignoring method names, since methods are typically renamed in the context of delegation. Finally, notice that if we find a *single* method of the class C that delegates (part of) its processing to a method of one of its attributes, we flag C as a potential implementor of the attribute's interface. Yet the number of delegated methods is not the best indicator of how good a match the delegation is. Thus, we currently filter the list of candidates manually by inspecting the results.

3.3 Identifying instances of state multiplication

Formal concepts analysis (FCA) [7] addresses the construction of conceptual abstractions, i.e. intentionally described clusters, out of a collection of entities described by properties and the ordering of those abstractions w.r.t. generalization. Hence FCA is suitable for the discovery of cohesive groups of entities like in class hierarchy analysis [15] or construction [4, 8, 9, 11]. Here we explore its benefits for aspect mining.

3.3.1 Formal Concepts Analysis

In FCA, a set of concepts provided with a specialization order (the *concept* or *Galois lattice*) emphasizes commonalities in descriptions (by property sets) of entities. Concepts emerge from a formal context $\mathcal{K} = (E, P, I)$ where E is the entity set (formal objects), P the property set (formal attributes) and I (the incidence relation) associates E to P: $(e, p) \in I$ when entity e owns property p. Fig. 7 provides an example of a context where entities are classes and properties their members (in the table, abbreviations are introduced for subsequent use).

Any entity set $X \subseteq E$ has an image in P defined by $X' = \{p \in P \mid \forall e \in X,\ (e, p) \in I\}$. Symmetrically,

any property set $Y \subseteq P$ has an image in E defined by $Y' = \{e \in E \mid \forall p \in Y,\ (e,p) \in I\}$. In the example, let $Y = \{nb, b\}$, we have $Y' = \{CA, MA\}$, while for $X = \{CA, MA\}$, $X' = \{nb, b, cr\}$. A *concept* is a pair (X, Y) where $X \subseteq E$, $Y \subseteq P$, $X' = Y$ and $Y' = X$. In Fig. 7, $\{\{CA, MA\}, \{nb, b, cr\}\}$ is a concept. X (resp. Y) is usually called the *extent* (resp. *intent*) of the concept.

The specialization between concepts corresponds to extent inclusion (or intent containment). The set of all concepts and the corresponding order compose the *concept lattice*, $\mathcal{L}$, of the context (aka the Galois lattice). In Fig. 7, the concept $\{\{CA\}, \{nb, b, cr, d\}\}$ specializes $\{\{CA, MA\}, \{nb, b, cr\}\}$. Fig. 7 shows the lattice (top) and an interpretation thereof as UML model (bottom). In this model, a new $Account$ class emerges through the top concept of the lattice which can be considered as a learned abstraction. Class hierarchies designed through FCA present benefits including maximal property factorization and conformity of inheritance to member set inclusion.

3.3.2 Decomposition algorithms

FCA theory has contributed a large number of composition/decomposition operators for lattices. whereby the *subdirect* product which seems the most promising one for state multiplication discovery. Unlike *direct* product, i.e. the lattice extension of carthesian product, the subdirect one does not comprise all combinations of classes from the feature. It nevertheless represents a sublattice of the direct product whereby both factor lattice structures are still preserved (i.e., can be homomorphically mapped onto). In this respect, the subdirect product better reflects the natural requirement for a parsimony in class hierarchy design.

Yet the subdirect product comes with a price: First, there may be several subdirect decompositions of a lattice. This means choices will have to be made. Second, the perfect subdirect decomposability is relatively rare in arbitrary lattices. This in turn means that rather than a choice criterion for alternative decompositions, what one would need is a good metrics to detect cases of approximative decomposability, i.e., hierarchies whose Galois lattices would almost represent a subdirect product. We are currently studying such a metric based on an edit distance between the *arrow* relations (see [7]) of two contexts.

4 Preliminary experiments

Over the past year, we started experimenting with the three families of algorithms. The results we got with multiple inheritance and delegation helped us refine the algorithms presented in the previous section. Our work on state multiplication (Galois lattice and associated algorithms) is still fairly exploratory, and promises to be more challenging than the other two methods, for several reasons. First, the factorization inherent in Galois lattice is a double-edged sword. While it enables us to abstract away developers' imperfections, the more poorly designed a class hierarchy, the wider the difference between the original class hierarchy and the resulting Galois lattice. This makes the results harder to interpret. Second, a Galois lattice is not much more likely to yield a perfect factorization than the original class hierarchy. Third, Galois lattices are notoriously difficult to visualize, even for moderately sized class hierarchies– and lattices. We continue to explore heuristics that combine statistical / numerical methods with structural ones to get a handle on state multiplication.

Below, we first describe the experimental data, and then we summarize our findings.

4.1 The experimental data set

4.1.1 JHotDraw

JHotDraw is a Java open-source graphical user interface framework that was developed by Erich Gamma and Thomas Eggenschwiler based on the Smalltalk original developed by John Brant [2]. From its inception in Smalltalk, and through its porting to Java by Gamma and Eggenschwiler, and its current evolution as an open source project, one of the main objectives of JHotDraw has been to serve as an exercise or laboratory in OO design. The JHotDraw open-source project adds basic functionality to the framework, builds new applications based on the framework, but also includes lots of refactoring. We used version 5.1, which contains about 140 classes and 20 interfaces.

4.1.2 JREVERSEPRO

JREVERSEPRO is a Java program for reverse engineering compiled Java code. It takes as input a java .class file (the result of compiling a java source file), and can produce one of three structures, depending on the call parameters: 1) the class constants pool[6], 2) class disassembly, and 3) class decompilation. JREVERSEPRO is relatively small with 85 classes and interfaces, and about 12000 of code. JREVERSEPRO does not use outside libraries, except for the basic Java API. Unlike JHotDraw which has seen the contribution of many designers, JREVERSEPRO *appears* to be essentially the work of its creator.

4.1.3 JavaWebMail

JavaWebMail is an open source project aimed at a standard Java web mail client. The version we used, JavaWebMail 0.7, contains about 60 classes and 30 interfaces. Notice that

[6] a sort of a symbol table in the Java virtual machine standard

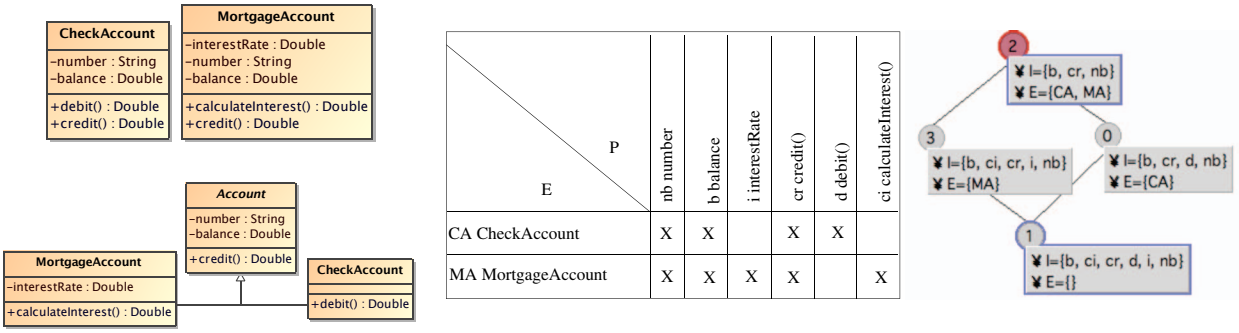


Figure 7. Left: Initial account classes (**top**) and resulting class hierarchy (**bottom**); **Center:** A formal context describing classes; **Right:** The concept lattice of the context.

it has not yet been officially released, and the project does not appear to be very active on the SourceForge site.

4.2 Results

In this section, we present the results for multiple inheritance and aggregation. Our experiments are not exhaustive by any means. However, the results have helped us gain insight into the underlying design issues. For each technique, we look briefly at the three software packages described above.

4.2.1 Multiple inheritance

We applied the algorithm described in section 3.1 to the three packages. We observed the following:

1. *JHotDraw*. We found several classes that extend a class and implement interfaces. A number of the implemented interfaces represent *listeners*, e.g. `CompositeFigure` extends `AbstractFigure` and implements `FigureChangeListener`. This is a common pattern in graphical frameworks based on the Java event model. The `FigureChangeListener` interface embodies some sort of a contract that graphical figures need to fulfill to react appropriately to specific events. We can think of it as *infrastructure functionality*.

2. *JavaWebMail*. All cases of "multiple inheritance" were filtered out by the algorithm: utility interfaces (e.g. `Iterator`), marker interfaces (`Serializable`) or constant interfaces (`JwmaInboxInfo`).

3. *JREVERSEPRO*. The results for JREVERSEPRO were comparable to those for Java Web Mail.

4.2.2 Delegation with aggregation

The preliminary results are as follows:

1. *JHotDraw*. We found several cases of delegation, e.g. the class `GraphicalCompositeFigure` (see Fig. 8). Its author writes in the class comment[7] "The GraphicalCompositeFigure manages contained figures like the CompositeFigure does, but delegates its graphical presentation to another (graphical) figure which ... mainly has a presentation purpose". In essence, the author of `GraphicalCompositeFigure` did not like the fact that the `CompositeFigure` class combined the functionalities of a composite *and* those of a figure, all in the same object. He is *resplitting* those functionalities which he would have preferred to see separated in the class `CompositeFigure`. Interestingly, `GraphicalCompositeFigure` is in the package "contrib"; it is not part of the framework core. Other cases included implementations of the *wrapper* pattern. An abstract class called , appropriately enough, `DecoratorFigure`, which wraps a `Figure` to which it delegates all of the methods of `Figure`. Concrete subclasses of `DecoratorFigure` can add pre- or post-processing to those delegated calls.

2. *Java Web Mail*. It did not exhibit cases of delegation, but we did find a couple of instances of aggregation where a component represented a complex attribute of the aggregate, and the aggregate provided methods to access values of that complex attribute.

3. *JREVERSEPRO*. A quick analysis of the JREVERSEPRO program did not identify cases of delegation, but we did find many cases of aggregation. This was somewhat surprising because the program is rather designed. We would have expected the three functionalities of the program (constant pool retrieval, disassembly, and decompilation) to be delegated to different components of the same object. Interestingly,

[7] the comments have been slightly rearranged for our purposes

JREVERSEPRO uses the *visitor* pattern to implement those functions! So, indeed, the three functionalities are delegated to three different objects– the visitors– but those objects are *not* statically bound to the aggregate through data member references: they are passed as arguments to the `acceptVisitor()` method!

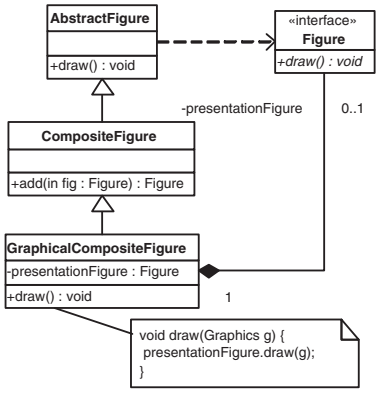


Figure 8. Splitting the functionality of a multi-function class

5 Conclusion

Aspect-oriented software development builds upon OO development by offering additional abstraction boundaries that help us separate different types of requirements into so-called aspects. Given a legacy OO application, it pays to identify existing aspects to help understand the structure of the application and potentially to extract those aspects into separate software modules that can be maintained and reused individually. We are interested in the extraction and repackaging of *functional* aspects. To identify such functional aspects, we analyzed the different design and programming patterns that developers would use to implement multiple functional aspects *without* aspect-oriented language constructs. Based on these analyses, we developed algorithms that help recognize such design and programming patterns in legacy code.

We tested two of our techniques on three packages, JHotDraw, JavaWebMail, and JREVERSEPRO, which were all developed in open-source. However, the similarity stops there. JHotDraw was meant as a live design laboratory, and has an active community. At the other end of spectrum, we have JavaWebMail, which has yet to release version 1.0. Of the three packages, the one that has the most identifiable cases of classes supporting multiple functional features is JHotDraw. It is also the package that will benefit *the least* from aspect-oriented (or any other kind) of refactoring, since it is already fairly well-factored. A number of researchers and practitioners have been saying from the late 90's that most textbook cases of aspect-oriented programming can actually be handled fairly well with the traditional OO techniques. Our experiments, as preliminary as they are, seem to further that claim. Our experiments were nevertheless useful as they enabled us to gain some insights in the packages that we looked at. The bigger challenge is to identify such aspects in poorly design programs, which tend to use variations on the state multiplication pattern, and that is where our techniques will add the most value.

References

[1] F. Aurenhammer, J. Hagauer, and W. Imrich. Cartesian product factorization at logarithmic cost per edge. *Computational Complexity*, 2(4):331–349, 1992.

[2] J. Brant. Hotdraw. Master's thesis, 1995. Master's Thesis.

[3] B. Dagenais and H. Mili. Slicing functional aspects out of legacy code, submitted, 10 p.

[4] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On Automatic Class Insertion with Overloading. In *Proc. of ACM OOPSLA'96*, pages 251–267. ACM Press, 1996.

[5] G. Kiczales et al. Aspect-oriented programming. In *ECOOP'97, pages 220–242, June*, 1997.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[7] B. Ganter and R. Wille. *Formal Concept Analysis, Mathematical Foundations*. Springer, Berlin, 1999.

[8] R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using Galois lattices. In *Proc. of OOPSLA'93*, pages 394–410. ACM Press, 1993.

[9] R. Godin, H. Mili, G. Mineau, R. Missaoui, A. Arfi, and T.T. Chau. Design of Class Hierarchies Based on Concept (Galois) Lattices. *TAPOS*, 4(2):117–134, 1998.

[10] W. Harrison and H. Ossher. Subject-Oriented Programming: A Critique of Pure Objects. In *Proc. of OOPSLA'93*. ACM, 1993.

[11] M. Huchard, H. Dicky, and H. Leblanc. Galois lattice as a framework to specify algorithms building class hierarchies. *Th. Informatics and Applications*, 34:521–548, 2000.

[12] W. Imrich. Factoring cardinal product graphs in polynomial time. *Discrete Mathematics*, 192:119 – 144, 1998.

[13] H. Mili, H. Sahraoui, H. Lounis, H. Mcheick, and A. Elkharraz. Concerned about separation. In *Proc. of FASE 2006)*, pages 247 – 261. Springer, 2006.

[14] M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proc. of AOSD'05*, pages 111–122. ACM Press, 2005.

[15] G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, 22(3):540–582, 2000.

[16] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proc. of ICSE 1999*, pages 107–119, 1999.