Identifying Functional Features in Legacy Java Code

Hafedh Mili, Amel Elkharraz and Petko Valtchev

Abstract—Typical OO applications implement several functional features that are interwoven into the same class hierarchies. In the absence of aspect-oriented techniques to develop and compose functionalities, developers resort to object-oriented design and programming idioms. Given a legacy OO application, it pays to identify existing functional features to help understand the structure of the application and potentially to extract those features into separate software modules that can be maintained and reused individually. We are interested in the identification of functional features in legacy OO code. We first characterize what we mean by functional feature, and then explore the footprints that such features are likely to exhibit in a legacy application that integrates many of them. We identified three such footprints: 1) aggregation, 2) multiple inheritance, and 3) repetition (ADHOC). We describe a set of algorithms for recognizing such footprints in legacy code, and then present the results of experiments where we applied those algorithms to five open source applications: JHotDraw, FreeMind, JReversePro, JavaWebMail, and Lucene. Our results show that: 1) the different algorithms are able to identify the important functional features/dimensions within an application, 2) they can identify opportunities for reuse and refactoring, 3) they are complementary, and 4) they can serve as the basis for a design aid. We compare our work to related research, and conclude by discussing directions for future research.

Index Terms—aspect mining, feature extraction, refactoring, multiple inheritance, delegation, state composition, formal concept analysis.

1 INTRODUCTION

LEGACY OO applications typically implement a number of functional features. This kind of situation happens when several functional features are requested in the initial development, or when new functionalities are added during maintenance. For example, the same Employee class may be used by a payroll functionality, and thus needs to support data and function members for things such as salary scale, number of hours worked, and by a production planning functionality, and thus needs to support function and data members related to skills, shifts, number of hours worked, and so forth. Our work is concerned with developing techniques for identifying functional features in legacy OO code for those cases where several features affect the same classes. Minimally, this should help us understand the legacy application and maintain a given feature without interfering with the others. This should also help us refactor our application to repackage those features in such a way that they can be developed and maintained separately, and composed on demand.

Ideally, our software abstraction and packaging technique should enable us to implement different functional features in distinct software artifacts that we can develop, maintain, and compose at will [1]. OO abstraction and packaging techniques enable us to separate object-specific features by encapsulating them in classes but do not adequately separate features that affect or crosscut several classes. Aspect-oriented software development (AOSD) was meant to address this problem by proposing artifacts that enable us to untangle certain types of requirements that OO abstractions could not. However, before or without such techniques, designers had to cope with multiple functional features by using traditional OO language constructs or design idioms to, a) package such functional features, as classes or interfaces, and b) compose them, using techniques such as multiple inheritance and delegation, whether built into the programming language, or simulated with design idioms/patterns; we refer to such cases as deliberate codification/packaging and reuse of functional features. There are also cases where developers do not recognize such functional features in their code, and may end-up, unwittingly, coding the same functions in different places of the application, similar to code cloning. We refer to those instances as ad-hoc implementations of functional features. Our work aims at mining reusable functional features in legacy code by identifying both deliberate and ad-hoc occurrences of reusable functional features in the code.

In the next section, we characterize what we mean by functional feature, and take a look at the problem of identifying functional features in legacy code. Sections 3, 4, and 5 present the issues, and algorithms for uncovering functional features composed with multiple inheritance, delegation, and ad-hoc implementation, respectively. Section 6 presents the results of experiments uncovering functional features composed with multiple inheritance (6.2) and delegation (6.3). Section 7 presents the results of experiments uncovering ad-hoc implementations of functional features. We discuss the results in section 8. Related work is presented in section 9, and conclude in section 10.
2 Characterizing functional features

By functional feature, we mean a slice or “subset” of an OO application that addresses a cohesive subset of functional (domain) requirements. By functional requirement, we mean a requirement that deals with the (input, output) relationship implemented by the software, as opposed to non-functional requirements, which deal with the way the output is produced. It has been observed that the elements of an OO application contributing to a functional feature will exhibit stronger “cohesion” than the elements of the application as a whole [2]. Our work consists of developing methods for identifying sets of classes, attributes, and methods that might implement a distinct functional feature. In earlier work, we used code slicing to extract the classes, data members, and methods—actually, statements in such methods—that contributed to the return value of a particular function (e.g., computing the wage), and the results were promising [3]. In this line of work, we focus on the identification of functional features using class signatures (methods and attributes), with no regard for code-level relationships (e.g., call graphs, references, etc.).

To find out how the occurrence of several functional features within the same code base (class hierarchy) manifests itself, we have to consider how a developer might handle several features. The past two decades have seen the proliferation of techniques built on top of the OO paradigm to handle the separate development and packaging of functional features within the same classes, or class hierarchies (see e.g., [4], [5], [6], [7], [8], [9]). These techniques have come to be collectively referred to as aspect-oriented development techniques, named after the most mature and popular of these techniques, namely, Kiczales et al’s aspect-oriented programming [7]. However, in the absence of these techniques, developers have to resort to design and coding patterns to implement several functional features within the same class hierarchy. Our approach relies on recognizing such design/coding patterns in legacy code to identify candidate functional features.

A preliminary analysis identified two such patterns: 1) multiple inheritance, whereby each functional feature is represented using its own class hierarchy, and a class combining several features simply inherits from the corresponding class hierarchies, 2) delegation, whereby a separate class hierarchy represents each functional feature, which are combined by aggregation. In both cases, developers/designers have taken care to, 1) develop and package such functional features as classes and/or interfaces, with their own class/type hierarchies, and 2) to compose them with other base functionalities where needed, using either multiple inheritance or aggregation. Recognizing such classes and interfaces in legacy code is still useful, as it identifies reusable chunks of functionality, and gives maintainers an overview of the functional/design dimensions of the application.

Perhaps a more useful discovery is one that helps maintainers recognize potentially useful/reusable functionality that the initial developers did not see. This can happen if the same functionality has been, inadvertently, coded several times within the application. This can happen with novice developers who fail to recognize commonalities or to refactor them properly, or with large applications that are maintained by several people who are too quick at coding, not taking the time to study the existing code base. This is similar to code cloning where the body of a method (or parts thereof) is copied and used in a different context, as is or close to it. It differs from code cloning in terms of granularity: instead of cloning instructions across methods, we talk about cloning ‘configuration of methods’ that together, implement a useful functionality. We refer to this as ad-hoc implementation of functional features. Simply put, we recognize such situations by the multiple occurrence of ‘configurations of methods’ in different parts of the legacy application (see section 5).

Naturally, when studying a legacy application, it is not known beforehand whether the application embodies different functional features, and if it does, which of the above techniques has been used. Both the presence/absence of functional features, and the composition technique used to compose them, are hypotheses to be tested by applying our techniques. Sections 3 and 4 will study the first two patterns, whereas section 5 will study the ad-hoc case. The algorithms proposed in sections 3, 4, and 5 are validated in sections 6 and 7. Figure 1 provides a conceptual map of our work—and of the paper.

3 Uncovering functional features composed with multiple inheritance

3.1 Implementing multiple functional features with multiple-inheritance

Figure 2 illustrates this situation. In this case, each functional feature is implemented in one class hierarchy. A class that integrates several functional features inherits from one node/class from each hierarchy.

When the programming language does not support multiple inheritance—as many do not, including Java—developers resort to other programming constructs and idioms. For example, while Java does not support multiple implementation inheritance, it enables us to define a type (an interface) as an extension of several non-hierarchically related super-types. Figure 3 shows some of the ways that a Java devel-
need to rely on a combination of our knowledge of the Java API and on some heuristics. For example, if we are studying an MIS application, and we are trying to identify functional features, we can safely ignore all the interfaces that are part of the basic Java API (e.g. packages java.*, javax.*) and related utilities (e.g. org.omg.*, org.w3c.*). If, on the other hand, our application is a graphical editor, say, then all the AWT (java.awt.*) and SWING (javax.swing.*) packages are legitimate. Additional heuristics may consider the kinds of interfaces. Marker interfaces and interfaces that consist only of constants can be ignored. So are single-method interfaces. We expect that experimentation will help us define and refine such heuristics.

The algorithm for identifying classes with multiple inheritance is given below:

1) **foreach** class C in the program P:
   a) Let \( \text{SUPER}(C) \leftarrow \{ \text{superclass}(C) \} \cup \{ \text{the interfaces implemented directly by C} \} \)
   b) Remove from \( \text{SUPER}(C) \) the interfaces that are part of the Java API
   c) Remove from \( \text{SUPER}(C) \) the interfaces that are marker interfaces
   d) Remove from \( \text{SUPER}(C) \) the interfaces with just constants in them
   e) If \( | \text{SUPER}(C) | \geq 2 \), mark class \( C \) as a candidate class implementing several features

Here, we should qualify the step consisting of removing the interfaces that are part of the Java API: as mentioned above, the list of APIs to exclude depends on the application being analyzed. If we are analyzing a graphical framework package, for example, interfaces from the java.awt.* or javax.swing.* packages are relevant to the domain.

Notice that this algorithm may return pairs of classes \( C_1 \) and \( C_2 \) such that \( \text{SUPER}(C_1) = \{ \text{superclass}(C_1), I_1 \} \), for some interface \( I_1 \), and \( \text{SUPER}(C_2) = \{ C_1, I_2 \} \), where \( I_2 \) is a subtype of \( I_1 \). In others words, \( C_2 \) specializes \( C_1 \), and the other interface it implements specializes the interface implemented by \( C_1 \). Here, we do not consider these cases are separate, and for the purposes of identifying potential functional features, we consider superclass(\( C_1 \)) and \( I_1 \) as being representative of those features. We did not automate this filtering of candidate classes in our current implementation: this redundancy came up in our experiments (discussed in section 6).

## 4 Uncovering functional features composed with delegation

### 4.1 Implementing multiple functional features with delegation

Delegation is another common design technique that can be used to support multiple functional features. Figure 3.c shows one such example. The term “delegation” has a precise meaning in delegation-based languages\(^1\). Here, we

\(^1\) In delegation-based languages, classes are not repositories of behavior, but individual objects (prototypes) are: new objects can delegate behavior to existing objects who execute their own methods in the context of the delegator.
use it in the following looser sense: class A delegates to class B iff:

- class A defines an attribute (data member) of type B – call it b;
- class A implements the behavior of B, and
- the A implementation of a B method forwards the call to the attribute b.

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 implementations of the various methods of the interfaces return the results of calling the corresponding methods on the attributes worker and immigrant, respectively. Figure 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 being explicit through the implementation of a common interface. For our purposes, this means that we need to examine the set of methods implemented by each class to ensure that ImmigrantPartTimeStudent implements the behavior of its delegates. In practice, it will be unlikely that all of the methods of the delegates’ classes Worker and Immigrant in our case are implemented by the delegator class ImmigrantPartTimeStudent. Figure 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 implementations. This raises the question of “how partial”. One could use a percentage: in this case, we need to replace the following condition in our definition of delegation:

- class A implements the behavior of B, and by:
- class A implements at least alpha percent the behavior of B.
However, such a heuristic is likely to produce many false-positives and many false-negatives, for two reasons. First, domain classes will typically have a few domain methods, and lots of utility methods (constructors, accessors, serializers, hashers, comparators, 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.

4.2 Extraction algorithm

Like with the case of 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\(^3\). 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:

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

In this case, we cannot say that the Personnel class delegates to the Collection class/interface. These code excerpts illustrate another potential problem that could occur 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 signatures (return type, parameter types).

The algorithm for identifying delegations focuses on the functions implemented by classes and compares them to the methods implemented by their attributes without regard for the interfaces either of them implements. 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. The algorithm that we implemented goes as follows:

Here, DOMINT (C) represents the domain interface of class C. That means, C’s full interface from which we remove accessors and methods inherited 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; recall that 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. As mentioned in section 4.1, the number of delegated methods is not a good indicator of how good a match the delegation is. Thus, for the time being, we filter the list of candidates manually by inspecting the results.

5 Ad-hoc implementations of multiple functional features

5.1 Characterizing ad-hoc implementations of multiple features

We use the term ad-hoc implementation to refer to cases where the developer did not take any special measures to separate the code aspects that implement a particular functional feature from the rest of the class hierarchy. In this case, class members that implement the feature are simply in-lined in those classes that support that functional feature. How, then, do we recognize that a given set of class members (data and function) contribute to the same functional feature? The basic premise of our approach is that a functional feature that is not represented intentionally, i.e. factored as a class or an interface (as illustrated in Fig 3), will be represented extensionally, i.e. expanded in all those places of the class hierarchy/tree– or class forest– where it is used. The key to recognizing that a set of function and data members embodies a functional feature is then to have that set occur in many places in a class hierarchy or forest.

Let us first look at a special case of ad-hoc implementation– that we call state multiplication– to understand the intuition behind our characterization; we will then give a more general characterization of the problem. State multiplication is best understood in the context of a situation where we want to combine several features, each of which comes in different flavors. Consider the example of cars, which come in many body types, including sedan, coupe, and station wagon; cars can also have various power plants. Figs. 5.a and 5.b show what hierarchies of body types 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 Figs. 5.c and 5.d includes the sum of the “state variables” of the body type and engine components (all combinations are present). Incidentally, the hierarchies in Figs. 5.c and 5.d correspond to typical situations that would warrant the use of the strategy pattern. Indeed, the strategy pattern handles such situations by, i) externalizing the combined features into their own hierarchies (so-called strategy objects), i.e. recovering the hierarchies in Figs. 5.a and 5.b, and ii) by letting the base object (Car) refer to their features (body type and engine type) through aggregation/delegation [10].

How would such an implementation arise? Most likely, if a developer—even an inexperienced one—has given the Chassis and Engine type hierarchies beforehand, s/he will try to combine them, by using either aggregation or multiple inheritance. Hierarchies such as the ones 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, a novice developer who is given all four combinations of features at once might still propose such a solution.

We have shown in [11] how perfect cases of state multiplication (all possible combinations of features) have precise mathematical characterizations⁴ that lend themselves to well-known and efficient factorization algorithms (see e.g. [12]). In real-world applications, such complete and symmetrical combinations of features are unlikely to occur, for several reasons:

- **A class is more than the combination of its functional features.** Indeed, a class hierarchy would embody some basic core functionality—what Tarr et al. called the dominant decomposition [8]—on top of which functional features might be added,
- **The sparseness of the domain.** In Fig. 5, all combinations of chassis type and engine are possible and of interest. In real life, some of them may not be technologically possible or just not interesting; there may be no market for a station wagon with a turbo-compressed engine!
- **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, we will be interested in the more general case illustrated in Fig. 6. We recognize a functional feature by the occurrence of a structural pattern of data and function members in different places in the hierarchy, and we assume that they impart a particular behavior on the classes to which they attach. In this case, the functional feature represents what it means to be a production resource: a production resource has capabilities and a schedule; it specializes into assembly line resources (both machine tools and people), and transportation resources (rolling stock and drivers).

In the next subsection, we will present an algorithm for identifying such patterns in legacy class hierarchies.

### 5.2 Extraction algorithm

One way of identifying recurring patterns of attributes and functions, such as the one illustrated in Fig 6, is to perform some sort of a clustering of hierarchy fragments based on the set of attributes and functions that they contain, and their placement in the hierarchy. We remain purposely vague on what ‘hierarchy fragment’ means, and about the relationship between those fragments and the attributes / functions; these will be explained in section 5.2.2. As for the clustering technique, we propose to use formal concepts analysis (FCA) [13] which supports the construction of conceptual abstractions, or **concepts**, out of a collection of individuals with properties. FCA has been used extensively in software engineering research (see e.g. [14], [15], [16]), and we have used it ourselves to factor out class hierarchies [17]. We start by providing a brief introduction to FCA in 5.2.1. We then discuss how we are going to encode our problem using FCA (see section 5.2.2). We finally present the feature mining algorithm in section 5.2.3.

#### 5.2.1 Formal Concept Analysis

(Formal) concepts analysis (FCA) [13] addresses the construction of conceptual abstractions, or **concepts**, out of a collection of individuals described by properties. In FCA, 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 (on the bottom) where entities are classes (listed vertically) and properties their members (listed horizontally). Here, pairs that belong to the relation are denoted by \( X \) and those that do not by \( 0 \).

Any entity set \( X \subseteq E \) has an image in \( P \) defined by \( X^I = \{ p \in P \mid \forall e \in X, (e, p) \in I \} \). Symmetrically, any property set \( Y \subseteq P \) has an image in \( E \) defined by \( Y^I = \{ e \in E \mid \forall p \in Y, (e, p) \in I \} \). In our example, let \( X = \{c1, c4\} \), then \( X^I = \{m1, m3\} \), whereas for \( Y = \{m1, m3\} \), \( Y^I = \{c1, c2, c4\} \). A **concept** is a pair \( (X, Y) \) where \( X \subseteq E, Y \subseteq P \), such that \( X^I = Y \) and \( Y^I = X \). In Fig. 7, \( \{c1, c2, c4\}, \{m1, m3\} \) is a concept. \( X \) and \( Y \) are called the **extent** and the **intent** of the concept, respectively (denoted, for a concept \( c \), \( ext(c) \) and \( int(c) \)).

The specialization between concepts corresponds to extent inclusion (or intent containment). The specialization relationship between the concepts defines a partial order that can be represented by the concept lattice—call it \( \mathcal{L} \). Fig. 8 shows such a lattice where the concept \( \{c1, c2\}, \{m1, m2, m3\} \) specializes \( \{c1, c2, c4\}, \{m1, m3\} \).

A straightforward interpretation of the concept in Fig. 8 could suggest the design of a new class or interface to host the declaration of method \( m1() \), which is shared by four classes. Such interpretation underlies the application of FCA to class hierarchy refactoring [18], whose benefits include maximal property factorization and conformity of inheritance to member set inclusion.
5.2.2 Encoding class hierarchies for feature mining

Based on the discussion of section 5.1 and the example in Fig 6, our goal is to cluster 'hierarchy fragments' based on: a) the set of class members that appear in them, and b) their localization in the hierarchy fragment. Figure 9 shows the type of incidence relation we are interested in. The entities would be hierarchy fragments; in this case, a class X with two subclasses. The properties are the set of data members occurring in the order shown: capabilities and schedule in the top class, and assemblyLine and licenseClass in subclasses. However, such an encoding is both prohibitively complex and too restrictive. First, for a given hierarchy fragment— says class Machinery with subclasses MachineTool and RollingStock— the number of possible attribute set patterns in exponential in the number of attributes of the classes of the fragment. Second, the matching of attribute set patterns among themselves—a graph matching problem—is itself fairly complex. The incidence relationship in Fig 9 is also too restrictive. For example, requiring the attributes assemblyLine and licenseClass to occur in immediate subclasses of class X is unnecessarily restrictive.

Accordingly, we choose to simplify the entity set and the property set, and relax the incidence relation. Entities now correspond to subhierarchies, represented by their root class. In effect, this means two things: a) we do not care about the shape of the structure underneath the root, and b) we exclude class patterns whose member classes do not have a common ancestor. To each subhierarchy, we associate the attributes that appear anywhere in the subhierarchy, regardless of relative position. Does that mean that the structure does not matter? Not necessarily. However, we believe that the same set of attributes is unlikely to appear in two places in the hierarchy in a different hierarchical order. Figure 10 illustrates the new incidence relation.

With this new incidence relationship, the concepts contained:

5. Let l, m, and n be the number of attributes of Machinery, MachineTool, and RollingStock, respectively. There are (2l−1) non-empty subsets of attributes for Machinery, and (2m−1) non-empty subsets for MachineTool, and so forth. Thus, there are (2l−1)×(2m−1)×(2n−1) possible attribute set patterns.

6. With the example of Fig 9, if assemblyLine and licenseClass occurred in one place as ‘descendants’ (so to speak) of the attribute set {capabilities, schedule}, they are unlikely to appear above the attribute set elsewhere.
Fig. 7. **Left:** Initial classes; **Right:** encoding as context.

Fig. 8. The concept lattice of the context in Fig. 7.

Fig. 9. An example of the incidence relation to use for our context, based on Fig 6.

Fig. 10. A simplified context, based on Fig 6.

Fig. 11. The three occurrences of \{capabilities, schedule, assemblyLine, licenseClass\} are not independent.

Actually, we are not there yet. Consider the following candidate node, uncovered by our algorithm for the case of JHotDraw. Our purge of non-minimal classes was motivated by the fact that classes automatically ‘reverse-inherit’ the methods implemented by their subclasses, and thus, they should not count as independent occurrences. However, in the case of figure 12, the class Command implements the methods boolean isExecutable() and void execute() locally and does not have them solely through ‘reverse inheritance’. Thus, the class Command should count as an independent occurrence of the intent. Thus, our extent purge algorithm only purges those non-minimal classes that do not locally implement the methods of the intent. This refined purge has the added advantage of making cases such as assemblyLine, licenseClass}, and yet, we cannot say that the attribute set ‘occurred several times’, the reason being that the three occurrences correspond to three nested subhierarchies (see Figure 11). Thus, not every concept with an extension containing more than one class (subhierarchy) suggests a feature: the classes that are in the extent should not be hierarchically related. Practically, this means that before considering the intent (i.e. set N) of a concept (A, N) as a potential feature, we should not consider the cardinality of A, as is, but we should consider the number of minimal classes in A with respect to the order defined by subclass relationship. In the example of Fig 11, the extent \{Resource, Personnel, ShopFloorStaff\} has a single minimum with respect to the subclass relationship, which is the class ShopFloorStaff, and thus, based on the right branch alone, we cannot say that the set \{capabilities, schedule, assemblyLine, licenseClass\} represents a potential feature.
'multiple inheritance' and 'delegation with aggregation' recognizable in node extents (see section 7.1).

Notice that if a given set of attributes has two or more independent occurrences, making it a candidate feature, _a fortiori_, any subset of that set of attributes _also_ has at least as many independent occurrences, and some of those subsets may themselves qualify as features. Going back to the example of Fig 6, according to our definition, either subset (singletons) \{assemblyLine\} or \{licenseClass\} also has two independent occurrences (see Fig 13). For a given number– and site– of occurrences, we see the largest feature, _i.e._ the one including the largest number of attributes.

In the next section, we present a formal algorithm used to identify the candidate features, based on the definitions and observations discussed above.

### 5.2.3 The algorithm

Let \( P \) be a program with a class hierarchy \( \mathcal{H} = \langle C, \leq \rangle \), where \( M \) is the set of data and class members. We build a context \( \mathcal{K} = \langle C, M, I \rangle \) using the incidence relation \( I \) discussed in the previous section, which associates to each subhierarchy with root \( c \) all the members (data or function) that occur anywhere in the subhierarchy. To extract the candidate functional features, we analyze the concept lattice of \( \mathcal{K}, \mathcal{L} \). At a pre-processing step, for each concept \( n = (X, Y) \) from \( \mathcal{L} \), the minimal classes in its extent, \( \min(X) \), are calculated.

The mining algorithm as shown below takes the concept lattice \( \mathcal{L} \) as input and outputs the list of candidate features \( \text{CandFeature} \) hereafter. For each concept \( (X, Y) \), starting from the lattice top \( (T_{\mathcal{L}}) \), the algorithm checks that there are at least two classes in \( \min(X) \), and that there is no immediate predecessor concept with the same extent number of minimal classes. Concepts satisfying both conditions are stored in \( \text{CandFeature} \) for further manual inspection.

### 6 Validating Detection of Features Composed with Inheritance and Delegation

The algorithms presented in sections 3 and 4 were implemented in Java under the Eclipse environment. In particular, we used Java's reflection package, and Eclipse's JDT API to analyze the code that we wished to mine for functional features. The code analysis consisted of:

- Extracting class and interface signatures, which consist of the set of public functions– and non-static public data members, for classes.

---

**Fig. 12.** A case where we should not eliminate a non-minimal class \( \text{Command} \) from the extent because it 'owns' the intent independently of its descendants.

**Fig. 13.** A candidate feature can induce several candidate subfeatures.

**Fig. 14.** Excerpts from the concept lattice for Fig 13.
• Extracting class/interface relationships, consisting of, a) extension relationships between classes, b) extension relationships between interfaces, c) implements relationships between classes and interfaces, and d) aggregation relationships between classes, i.e. cases where class A has a data member that is of type class B.

We excluded from our analysis the code that came from common libraries. For example, when analyzing the JHotDraw package (see below), if a JHotDraw class inherits from Java’s JComponent, we exclude JComponent from the analysis. This has led to some false positives, as explained later, but as we will see, such false positives can be filtered out easily.

To validate our algorithms, we applied them to a number of open-source java applications. We start by describing the experimental data (see section 6.1). The detection of functional features combined with multiple inheritance is discussed in section 6.2, and that of features combined with delegation is discussed in section 6.3; the detection of ad-hoc implementations is discussed in section 7.

6.1 Experimental data

To validate our approach, we applied it to a number of open-source Java applications. The applications covered a wide range of application areas, from a graphical editing framework (JHotDraw), to a tool for reverse engineering compiled Java code (JReversePro), to a web mail client (JavaWebMail), logging (Log4j), free text search (Lucene), and mind mapping (FreeMind). The applications have different maturity levels, going from version 0.7.1 for FreeMind, to version 5.2, for JHotDraw. This, combined with the type of the application—system-type applications versus domain applications—meant that the applications did not have the same design quality. The five open-source packages are presented briefly below:

• JHotDraw is a graphical user interface framework that was developed by Erich Gamma and Thomas Eggenschwiler based on the Smalltalk original developed by John Brant [19]. 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 object-oriented design. Consecutive versions of JHotDraw added basic functionalities to the framework, included new applications based on the framework, but also involved regular refactorings. We used version 5.2, which contains about 160 compilation units (.java files) and 171 types (user-defined classes and interfaces).

• 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, 2) class disassembly, and 3) class decompilation. JReversePro is relatively small with 85 classes and interfaces, and about 12000 lines 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.

• JavaWebMail is a servlet-based Java web mail client that can connect to IMAP or POP mail boxes. The version we used, JavaWebMail 0.7, dates back to 2002. The next version (1.0.1) was released in October 2008, and seems to be the work of a single developer. The application was later rewritten ‘completely’ in 2014 (versions 2.0 and beyond), with ‘minor versions’ coming out regularly since. Like JReversePro, Java WebMail 0.7 seemed to be the work of its creator(s).

In terms of design quality and robustness, we don’t expect it to be as mature as JHotDraw.

• FreeMind is a mind mapping tool. A mind map is a graph whose nodes represents concepts or ideas, and whose links represent associative relationships between those concepts/ideas, a sort of an informal semantic network. Mind maps are used to ‘take notes’ or record ideas in loosely structured or vaguely understood domains. Mind maps and FreeMind have evolved over the years to include search functionality, links to other sources, AND/OR goal trees, and the like. We performed our experiments on version 0.7.1. The latest ‘production’ version (1.0.1) dates back to April 2014. FreeMind appears to have five main developers, with many smaller contributors.

• Lucene is a Java-based text-search engine library developed under the Apache foundation. Lucene is used by hundreds of open source projects, web applications, commercial products, and web sites, including Apple, IBM, the Eclipse platform, JIRA, CodeCrawler, AOL, and Comcast. The latest release, 5.5.0, was released in February 2016. The version we used in our experiments is version 1.4.

Table 1 provides some quantitative data about the different packages. The column “com units” stands for compilation units, i.e. Java source files. The column ‘Types’ represents the number of distinct classes and interfaces defined. The authors of the FreeMind package made an extensive use of member/internal classes, hence, the number of types (classes or interfaces) largely exceeds that of source files.

<table>
<thead>
<tr>
<th>Software</th>
<th># LOC</th>
<th># comp units</th>
<th># Types</th>
<th># Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeMind 0.7.1</td>
<td>65490</td>
<td>92</td>
<td>198</td>
<td>4785</td>
</tr>
<tr>
<td>JavaWebMail 0.7</td>
<td>10707</td>
<td>111</td>
<td>115</td>
<td>1079</td>
</tr>
<tr>
<td>JHotDraw 5.2</td>
<td>9419</td>
<td>160</td>
<td>171</td>
<td>1229</td>
</tr>
<tr>
<td>JReversePro 1.4.1</td>
<td>9656</td>
<td>83</td>
<td>87</td>
<td>663</td>
</tr>
<tr>
<td>Lucene 1.4</td>
<td>15480</td>
<td>160</td>
<td>197</td>
<td>1270</td>
</tr>
</tbody>
</table>

| TABLE 1 | Basic size metrics about the tested software packages |

6.2 Multiple inheritance

We applied the algorithm described in section 3.2 to the five packages. The preliminary results are summarized in table 2.
As we will see below, a number of these instances turned out to be ‘false positives’:

- **JHotDraw.** Of the 29 candidates, 22 cases were deemed to be legitimate. Nineteen (19) of these cases corresponded to a class A that extends a class B and implements one or several interfaces. The implemented interfaces are most often GUI event listeners, both from the Java GUI library (e.g. ItemListener), and from the JHotDraw framework itself (FigureChangeListener). For example, the class CompositeFigure extends the class AbstractFigure and implements the interface 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 some kinds of events. We can think of it as *infrastructure functionality*. The remaining three legitimate cases (for a total of 22) corresponded to interfaces that extend two or more interfaces, as in the interface ConnectionFigure, which extends the interfaces Figure and FigureChangeListener. The seven (7) false positives corresponded to cases where a class A inherited from a class B and implemented a *marker interface*\(^8\) C, or where an interface A extended several interfaces, all of which (or all but one) were marker interfaces. Marker interfaces can be easily identified by the fact that they have no methods, whether declared locally or inherited. Thus, the false positive cases could have been filtered automatically.

- **JavaWebMail.** In the JavaWebMail case, all twenty three (23) candidates for “multiple inheritance” were deemed appropriate. Twenty (20) of these cases corresponded to Java Web Mail plug-ins. In the Java Web Mail plug-in architecture, plug-ins had to implement two interfaces, ...server.Plugin, and ...server.URLHandler. The remaining three were cases where a class A extended a class B and implemented a non marker/constant interface, as in ...server.WebMailServlet extends ...server.WebMailServer, implements: javax.servlet.Servlet.

- **JReversePro.** In this case, fourteen (14) out of twenty four (24) candidates were false positives: they corresponded to cases where a class A implemented one or more *constant* interfaces\(^9\). The remaining 10 cases were legitimate cases of ‘multiple inheritance’ (i.e. a class inheriting/implementing two or more behaviors). However, all the valid cases dealt with the GUI of JReversePro. For example, we have the case of jreversepro.awtui.JDlgFont, which extends java.awt.Dialog and implements the interfaces, java.awt.event.ItemListener and java.awt.event.ActionListener. Admittedly, this is only a secondary aspect to the functionality of JReversePro.

- **FreeMind.** The thirteen candidate instances of multiple inheritance were deemed legitimate. Ten such candidates dealt with GUI functionality, as in freemind.main.FreeMindApplet, which extends javax.swing.JApplet and implements freemind.main.FreeMindPlugin and freemind.modes.EdgeAdapter, which extends freemind.modes.LineAdapter and implements freemind.modes.MindMapEdge. The remaining three cases correspond to an application of the adapter pattern, as in LinkRegistryAdapter\$ID_UseStateAdapter which extends LinkRegistryAdapter.ID_BasicStateAdapter and implements MindMapLinkRegistry.ID_UseState. While these cases are somewhat awkward, they are legitimate.

- **Lucene.** For the case of Lucene, six candidates turned out to be false positives, corresponding to marker interfaces, as in *.*.search.Query which implements both java.io.Serializable and java.lang.Cloneable (four cases), and constant interfaces, as in *.*.analysis.standard.StandardFilter which extends *.*.analysis.TokenFilter and implements *.*.analysis.standard.StandardTokenizerConstants. The four legitimate cases do correspond to a true combination of two or more domain functionalities, as in *.*.index.SegmentTermPositions which extends *.*.index.SegmentTermDocs and implements *.*.index.TermPositions.

Table 3 shows the synthesized results, once we use filters for marker and constant interfaces.

<table>
<thead>
<tr>
<th>Software</th>
<th>JHotDraw</th>
<th>JavaWebMail</th>
<th>Lucene</th>
<th>JReversePro</th>
<th>FreeMind</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>5.2</td>
<td>0.7</td>
<td>1.4</td>
<td>1.4.1</td>
<td>0.7.1</td>
</tr>
<tr>
<td>candidates</td>
<td>29</td>
<td>23</td>
<td>10</td>
<td>24</td>
<td>13</td>
</tr>
</tbody>
</table>

*TABLE 2*

Number of candidate instances of ‘simulated multiple inheritance’ in the tested software packages

<table>
<thead>
<tr>
<th>Software</th>
<th>JHotDraw</th>
<th>JavaWebMail</th>
<th>Lucene</th>
<th>JReversePro</th>
<th>FreeMind</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>5.2</td>
<td>0.7</td>
<td>1.4</td>
<td>1.4.1</td>
<td>0.7.1</td>
</tr>
<tr>
<td>legitimate</td>
<td>22</td>
<td>23</td>
<td>4</td>
<td>10</td>
<td>13</td>
</tr>
</tbody>
</table>

*TABLE 3*

Number of candidate instances of ‘simulated multiple inheritance’ after applying the constant and marker interface filter

The results of this experiment confirm that the interfaces that occur in configurations such as those in Fig 3 typically correspond to legitimate, independent functional features. This is not surprising as a developer’s act of making a class implement an interface that is neither a utility, marker, or constants interface, necessarily comes from a *deliberate* choice.

\(^8\)A *marker* interface is an interface with no methods that is used solely to mark the objects that implement it so that they may be processed in a particular manner by *other* code. For example, the Serializable interface contains no methods, but is used so that the writeObject(Object o) method of ObjectOutputStream, say, knows what to do.

\(^9\)Constant interfaces are interfaces that consist solely of constants/public static data member: a class that implements such an interface can refer to its constants in its code with no qualification, i.e. without bothering with the “dot” notation.
wish to impart some domain functionality to a class, one it cannot inherit because it is already inheriting other domain functionality!

That being said, there are some differences between the five packages. For the case of JHotDraw and JavaWebMail, and to some extent, Lucene, the 'inherited' functionalities are central to the application domain in question. For the case of JReversePro and FreeMind, the inherited interfaces dealt with GUI aspects, which are not central to FreeMind\(^\text{10}\), and even less so for JReversePro, whose domain abstraction include things so as syntax/expression nodes, symbol tables, constant pools, and the like. More on the differences between packages after we study the instances of aggregation.

6.3 Delegation with aggregation

For delegation, our results were produced in a two-stage process. The first stage is automated, and for each pair <class, attribute>, determines the percentage of the attribute's domain interface, that is implemented by the class itself. For the purposes of this experiments, we computed the domain interface of a type as the set of all of its methods (locally defined and inherited), from which we subtract:

1) its constructors,
2) its accessor methods identified with the Java Beans patterns, i.e. <some type> get <some name>{} for getters, and void set <some name>{ <param name> <param type> for setters. Note that we did not check, in either case (getter or setter) whether the type had an attribute with name <some name>, nor did we look inside the code to check whether the method was indeed returning the value of attribute. This filter is reliable to the extent that developers adhered to the Java naming conventions\(^\text{11}\).
3) methods inherited from classes 'outside' of the application domain of the class. For example, all classes inherit from java.lang.Object, which defines the methods hash() and equals(Object obj), among others. Those methods should be discounted from the domain interface. The question, then, is how to recognize, among the ancestors of a class, which ones provide domain-specific methods, and which ones provide non-domain specific methods. This is hard to assess ... and depends on the domain. If we were dealing with business packages, say, we could safely say that any class from the java.* or javax.* packages, say, is outside of the domain. However, in our case, most packages include some measure of system functionality, and some of these packages may be legitimate, domain-relevant ancestors. For example, for the JHotDraw package, the javax.swing.* and java.awt.* packages are domain-relevant, and event handling functionality, say, is a central aspect. Thus, short of filtering/excluding packages by hand, as a first approximation, we considered that all methods defined by ancestors defined outside of the current project are to be excluded\(^\text{12}\).

Given a <class, attribute>, we compute the ratio:

\[
\text{ratio} = \frac{\text{size}(\text{DOMINT}(\text{attribute}) \cap \text{DOMINT}(\text{class}))}{\text{size}(\text{DOMINT}(\text{attribute}))}
\]

where \(\text{DOMINT}(x)\) is the domain interface of \(x\). A ratio of 1 means that all the domain methods of the attribute are implemented by the containing class. We call this a perfect match, suggesting a 'strong' candidate for delegation\(^\text{13}\).

The second stage of our experiment is manual and consists of analyzing the potential candidates to check whether, for each method \(f()\) that is implemented by both class and attribute, whether the class's implementation of \(f()\) 'delegates' to the attribute\(^\text{14}\), as in void \(f()\) { attribute.\(f()\)}. The latter is the simplest case of 'call forwarding': most often, the call to attribute.\(f()\) is drowned somewhere in the middle of more or less complex logic, as in:

```java
class A {
    private AttributeType attribute;
    ...

    ReturnType f() {
        if (attribute != null) {
            return attribute.f();
        } else {
            // do something else
        }
    }
}
```

While it is possible to programatically detect/validate instances of call forwarding to an attribute, for the purposes of this experiment, we performed the analysis by hand.

The preliminary results are presented below per software package. A summary of results will be presented at the end. Note that for each package, we only analyzed the \(\text{class, attribute}\) pairs whose coverage ratio exceeded 50%. Of those, we identified the perfect matches.

6.3.1 Delegation with aggregation: JHotDraw 5.2

Table 4 shows the results for the JHotDraw 5.2 package. We analyzed manually only those candidates with coverage ratio greater than or equal to 50%, i.e. 15 cases in all.

\(^{10}\) the notion of graph is, of course, central to FreeMind, but things such as event listeners are peripheral to the business of drawing mind maps
\(^{11}\) we came across both false negative accessors (accessors that were not filtered by our program) and false positive ones (filtered methods that were not accessors) in the less mature packages
\(^{12}\) The five software packages were analyzed as Eclipse Java projects, and Eclipse's JDT package enables to us to identify which ancestors 'live' within the project, and which ones live outside
\(^{13}\) A perfect match may be too strong a condition for a legitimate case of delegation, as a class may not need all the functionality of an attribute. However, this is a useful benchmark.
\(^{14}\) As mentioned earlier, this is not the proper definition of delegation, which requires that the delegatee's method executes within the context of the delegator, but Java does not support unrestricted delegation.
The ‘perfect match’ cases (the domain interface of the attribute is included in the domain interface of the class) are divided into three categories:

- cases where the attribute type is an interface that is also implemented by the class, directly, or through an ancestor class (five cases, see Figure 15)
- instances of the singleton pattern (two cases). Indeed, in this case, the class has a static data member that is an instance of the class. This was the case for the IconKit and Clipboard cases. These could easily have been filtered by checking the scope of the attribute (static or not).
- what we might call a fortuitous match: this is the case with class StandardDrawingView, which has an attribute of type PointConstrainer. The interface PointConstrainer has three methods, int getStepX(), int getStepY() and void constrainPoint(Point p), and the class StandardDrawingView supports the protected method void constrainPoint(Point p). Because the first two methods were considered accessors by our domain interface calculation, this qualified as a perfect match.

![Fig. 15. Potential candidates where class implements the entire domain interface of the attribute](image)

By inspecting the cases where a class implements the type of an attribute, we realized that ‘attribute delegation’ was used for various purposes:

- one case was an instance of the decorator pattern (see Figure 16). Interestingly, in this pattern, it is the class (the decorator) that represents the modular behavior that we wish to add to the attribute (the decorated type), and not the other way around.

<table>
<thead>
<tr>
<th># candid.s.t. ratio=100%</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td># candid. 10% ≤ ratio &lt; 100%</td>
<td>7</td>
</tr>
<tr>
<td># candid. ratio &lt; 10%</td>
<td>25</td>
</tr>
<tr>
<td>Total cand. s.t. ratio &gt; 50%</td>
<td>15</td>
</tr>
</tbody>
</table>

**Table 4** Number of candidate instances of delegation through aggregation for JHotDraw 5.2

![Fig. 16. Potential candidate for delegation](image)

- in one case (SelectionTool, which implements the interface Tool through its ancestor, delegates to the attribute fChild of type Tool), the class delegates to an attribute so that its behavior can be changed during runtime, i.e. some sort of a strategy pattern. In this case, the interface Tool represents a common abstraction for different behaviors.
- in one case (TextFigure, which implements Figure through its ancestors, delegates to the attribute fObservedFigure, also of type Figure), the attribute represented a model while the class represented the view, within the context of MVC. In this case, no Figure behavior was actually delegated.
- two cases seem to be instances of the chain of responsibility pattern, and in those cases, the class methods do delegate to methods of the attribute; one such case is shown in figure 17.

When we looked at the seven ‘imperfect’ matches (one case at 75%, three cases at 66.66%, two cases at 62.5% (5/8ths), and one at 50%), we found the following:

- a JHotDraw-specific utility interface (Storable), which is inherited by many interfaces and implemented by many classes, ‘polluted’ the results by increasing the size of the ‘domain interfaces’ of both attributes and classes and increasing the match percentage.
- cases of deprecated methods, which ‘pollute’ domain interfaces— and thus, coverage percentages. Deprecated methods can increase both false positive (if both class and attribute have the same deprecated methods) and false negatives (if only the attribute type has them).
- two cases that should have yielded a perfect match, were it not for the fact that the method of the class, which delegates to a method of the attribute, used a different signature. We expected such cases to occur, but we were not sure of their prevalence.
- ‘substantial’ shared behavior between the interfaces Connector and Handle, which was ‘missed’ by the designers of JHotDraw.

15. We chose to exclude getter/setter methods from domain interfaces to not clutter such interfaces with attribute accessors, which we consider implementation details. However, when such methods appear in interfaces, as is the case here, one might question whether they should be excluded from the domain interface. In fact, one could argue that the domain interface of a Java interface is the interface itself.

16. The behavior of a SelectionTool depends on what we select: the background of a figure, a handle on the figure, or the figure itself. The attribute fChild enables us to support different behaviors, depending of what is selected.
6.3.3 JReversePro

Table 6 shows the results for the JReversePro package. The algorithm identified four cases with coverage equal to 0. We analyzed manually only those candidates with coverage ratio greater than or equal to 50%, i.e., 2 cases in all. The two candidates with perfect match invoked the class `jreversepro.reflect.JInstruction`, which has two attributes, `prev` and `next`, which point to the previous, and `next Instruction` – also a `JInstruction`, respectively. There is no behavioral delegation in this case.

6.3.4 FreeMind

Table 7 shows the results for the FreeMind package. The algorithm identified twenty three cases with coverage equal to 0. We analyzed manually only those candidates with coverage ratio greater than or equal to 50%, i.e., 3 cases in all. One candidate with perfect match concerns the class `freemind.modes.StylePattern`, which has an attribute, `ChildrenStylePattern` of type `freemind.modes.StylePattern`. However, upon inspection, no behavior was delegated to the attribute. The other two candidates corresponded to the class `freemind.modes.NodeAdapter`, with attributes, `preferredChild` and `parent`, both of type `freemind.modes.MindMapNode`, which is the interface implemented by `freemind.modes.NodeAdapter`. In this case, the attributes are used as linked structure 'pointers/links', with no actual behavior delegation.

6.3.5 Lucene

Table 8 shows the results for the Lucene package. The algorithm identified seventy eight cases with coverage equal to 0, including thirty-two perfect matches, and three candidates with percentage coverage between 50% and 100% (0.5, 0.5, and 0.6). We analyzed manually only those candidates with coverage ratio greater than or equal to 50%, i.e., 3 cases in all. By analyzing the thirty-five cases, we found the following:

- sixteen (16) cases of true delegation, including eleven (more or less clear-cut) instances of the `wrapper` pattern, and one instance of the `proxy` pattern
- nineteen (19) false positive cases that fall into the following subcategories:
  - eleven (11) false positives corresponding to cases where a class of type T has one or several

For the remaining four packages, we will be content to summarize the findings, elaborating only on new issues, as they are encountered.

6.3.2 Java Web Mail

Table 5 shows the results for the Java Web Mail package. We analyzed manually only those candidates with coverage ratio greater than or equal to 50%, i.e., 3 cases in all. One of the perfect matches corresponded to an instance of the `singleton` pattern (class `net.wastl.webmail.server.WebMailServer`, with singleton instance server), and should/could have been filtered. The other instance, for class `net.wastl.webmail.server.PluginDependencyTree` and the attribute `logger` of type `net.wastl.webmail.server.Logger`, is a valid case of delegation: the methods of `FileStorage` do forward/delegate to the corresponding methods of the logger. The third case, at coverage 66%, involving the class `net.wastl.webmail.server.PluginDependencyTree` and the attribute `node` of type `net.wastl.webmail.server.Plugin`, is sort of an instance of the `composite` pattern, and may be considered as a legitimate case of delegation.

17. A PluginDependencyTree is a hierarchy of plug-ins, where the children of plugin p require the services provided by p. In this case, `net.wastl.webmail.server.PluginDependencyTree` delegates the methods `register(WebMailServer s)` and `String provides()`, which concatenates the services provided by all the plugins of the three
attributes, also of type T, used to create linked data structures (e.g. lists). In such cases, the attributes are used to navigate along the data structure, but no behavioral delegation takes place.
- six (6) false positive cases corresponding to public static attributes.
- two (2) cases of aggregation with no delegation.

6.4 Analysis of the results

From the above analysis, the following preliminary conclusions emerge:

- delegation through aggregation has many uses; many more than those illustrated by Figure 4 in section 4.1. As shown above– as if we needed to be reminded!– it is used in many design patterns, including the observer, decorator/wrapper, proxy, some variant of strategy, and chain of responsibility. While in all these cases the attribute type represented a distinct and domain-meaningful functional feature, the aggregation was not necessarily used for behavioral composition.

- our domain interface calculation, and aggregation detection algorithm, lacked precision. We identified a number of issues which could have been addressed programmatically, namely:
  1) instances of singleton, and more generally, attributes which are static
  2) refraining from removing methods that fit the getter/setter patterns from interfaces, as such methods are domain-meaningful and should be counted
  3) deprecated methods. While those can be detected easily, figuring out how to handle them is not straightforward: should they be discounted from the attribute's domain interface? from the class/aggregate interface? what if behavior delegation still goes through the deprecated methods? etc.
  4) validating when the method f() of class A 'delegates' (forwards to) a same-signature method of the attribute of type B is relatively easy ... for the simple cases. Unfortunately, the absence of evidence of method forwarding ... is not evidence of absence. A call graph/flow analysis is needed for more precise results.
- Our definition of aggregation is too restrictive. We say that A is a component of B if B has a data member that is of type A. However, B could point to a collection of As. If developers use genericity, we can recognize such cases by considering type parameters of Collection-like data members as components. However, genericity became available late in the (Java) game, and some developers still use non-typed collections, despite the compiler warnings!!!
- There is an interesting case with the interfaces Figure and FigureChangeListener. We know that a Figure is supposed to register a bunch of FigureChangeListeners, which would be a case of aggregation (provided we use parameterized collection type). However, that will not be visible at the interface level (which does not include data member definitions). Figure does have the methods void addFigureChangeListener(FigureChangeListener l) and void removeFigureChangeListener(FigureChangeListener l), and so we can guess the aggregation relation, but cannot ascertain it.
- notwithstanding the (perfectible) imprecisions mentioned above, perfect match/coverage, or more generally, a high degree of match, is neither a necessary, nor a sufficient condition for a legitimate case of behavioral composition through aggregation. A class may need only a subset of the functionality offered by class B, and thus delegate only some methods to an attribute of type B. Conversely, some 'utility behavior', as in the case of the interface Storable above, can pollute the results and lead to false positives (high match scores).

7 Detecting ad-hoc implementations of multiple functional features

For this experiment, we implemented the algorithm described in section 5.2.3 as an Eclipse plug-in, also using the JDT package. Recall that our algorithm uses formal concepts analysis (FCA) to detect multiple independant occurrences of the same set of methods, in different subhierarchies of the same project, using an incidence relationship (between types and public methods) that we called reverse inheritance, where we associated each type (class or interface) with the set of methods defined locally, or in any of its descendants (see section 5.2.2). As with the case of the experiment described in section 6.3, for each type, we used its domain local interface, as opposed to its (full)local interface, by excluding methods that are redefinitions of methods defined in classes outside of the project. For example, if a class defines locally a method toString() or equals(Object another), we exclude those methods from the incidence relationship since they are mere redefinitions of a generic behavior inherited from the class Object.

We start by presenting the basic metrics for the five software packages. As we explained in sections 5.2.2 and 5.2.3, not all nodes of the generated lattices correspond to true independent occurrences, and additional criteria were used to keep only those. Thus, we make a distinction between, 1) nodes/concepts of the lattice, 2) candidate features, and 3) candidate features involving two or more methods. Note, however, that we cannot dismiss multiple independent occurrences of a single method as not interesting; as mentioned...
previously, Java’s Comparable interface does have a single method, and marker interfaces, such as Serializable don’t even have a method.

<table>
<thead>
<tr>
<th>Software</th>
<th># lat. concepts</th>
<th># cand.</th>
<th># cand. &gt;= 2 methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeMind</td>
<td>251</td>
<td>69</td>
<td>42</td>
</tr>
<tr>
<td>JavaWebMail</td>
<td>162</td>
<td>50</td>
<td>29</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>394</td>
<td>154</td>
<td>123</td>
</tr>
<tr>
<td>JReversePro</td>
<td>105</td>
<td>27</td>
<td>18</td>
</tr>
<tr>
<td>Lucene</td>
<td>276</td>
<td>91</td>
<td>64</td>
</tr>
</tbody>
</table>

Number of candidate features for some open source packages

Recall that the original motivation behind this algorithm is to find ad-hoc implementations of functional features in code, i.e. cases where designers or developers implemented the same set of methods in different parts of the software without recognizing that similarity, and encapsulating it in a class or interface. However, the incidence relationship—and the algorithm—will also tag cases where the designer did recognize the commonality and encapsulated it in a class or interface. To take the example of JHotDraw, designers recognized the concept of ‘figure’, materialized in an interface CH.ifa.draw.framework.Figure, and provided several implementations of Figure, including EllipseFigure, RectangleFigure, and so forth. The incidence relationship will ‘reveal’ that the methods that make up the interface Figure ‘occurred’ independently in those classes. Thus, for the case of JHotDraw, we cannot say that our algorithm ‘discovered’ 154 (or 123) ‘interesting features’: we need to eliminate cases such as the Figure example. We also need to eliminate configurations that we called ‘delegation with aggregation’ (see sections 4 and 6.3), because those too will be tagged by the algorithm.

We developed a categorization of the various candidate nodes to distinguish between the various cases. We start by describing the categorization in section 7.1, and then examine the results for each of the five software packages.

7.1 A categorization of candidate nodes/features

As mentioned in the introductory paragraphs of section 7, the instances of multiple independent occurrences of a set of methods that our algorithm will uncover (candidate features) will include both:

1) cases where designers / developers recognized such sets of methods as features, which they encapsulated in interfaces or classes, and integrated using any of the design patterns discussed in sections 3.1 and 4.1, and whose experimental extraction was presented in sections 6.2 and 6.3. We can refer to these as deliberate features.

2) cases that were not seen by designers / developers. We refer to those as ad-hoc features.

Ultimately, the value of our algorithm lies in identifying cases of the second category. Thus, to validate our algorithm, we need to assess the extent to which those ad-hoc candidate features represented domain-meaningful, and generally useful behavior, that developers / designers should have encapsulated / packaged as interfaces or classes, or could do so as part of a refactoring effort.

In our first experimentations, we analyzed the candidate features by hand, dismissing deliberate features, and focussing on ad-hoc ones. However, the classification of candidates into deliberate versus ad-hoc was done manually, by locating the elements (classes) of the extent in the project type hierarchy, and manually analyzing their interrelationships. Not only was this tedious, but a good part of it could be automated. Thus, we set out to perform the categorization automatically, refining the categories, and developing metrics, as we went along, to better categorize the candidate features. We ended up with thirteen (13) categories, and two metrics to get a better understanding of the features. For the sake of understandability, we will introduce them gradually.

At a basic level, the deliberate features fall into three categories:

1) cases where the extent consists of an (Java) interface, along with its implementing classes. Call it EXPLICIT_INTERFACE_IMPLEMENTATIONS. Examples of such a category, from JHotDraw, include a candidate feature where the extent consisted of the interface Figure, and its various implementations (e.g. TriangleFigure, EllipseFigure, RectangleFigure, etc.).

2) cases where the extent consists of an (often abstract) Java class, along with its subclasses, where the subclasses redefine (some of) the methods of parent class. Call it EXPLICIT_CLASS_SUBCLASS_REDEFINITIONS. Examples of such category include the candidate feature where the extent consisted of the abstract class Command, along with the various command classes (e.g. CopyCommand, AlignCommand, CutCommand, etc.).

3) cases where the extent consists of a class A—call it component—and a set of classes that have A as a member—call them aggregates. Call it EXPLICIT_AGGREGATIONS. Examples of such category include the candidate feature where the extent included the classes Figure (the ‘component’) and CompositeFigure and DecoratorFigure (the ‘aggregates’).

Figure 18 shows this first-level categorization. Notice that for each of the deliberate categories, one type from the extent played a special role: the interface being implemented, the class being subclassed, or the class / component included in the other classes. We will refer to such special types as anchor types for the extent. A closer inspection of the output produced by this classification introduced an additional nuance. Indeed, for each of deliberate categories, we needed to distinguish between cases where the full extent consisted of such a configuration, versus cases where a subset of the extent consisted of such a configuration, while the remaining classes were unrelated to the anchor type. For example, for the case of JHotDraw the examples used above for EXPLICIT_INTERFACE_IMPLEMENTATIONS and EXPLICIT_CLASS_SUBCLASS_REDEFINITIONS corresponded to a full extent case: in the first case, the extent
consisted *solely* of the interface `Figure` and its implementations, and in the second case, the extent of the candidate node consisted *solely* of the abstract class `Command` and its subclasses. However, we also had a candidate feature with intent `{void animationStep()}`, whose extent consisted of the interface `Animatable`, the implementing class `BouncingDrawing`, but also the 'unrelated' class `AnimationDecorator`.

Thus, for each of the above deliberate categories, we made the distinction between `FULL_EXTENT_*` and `PARTIAL_EXTENT_*`, hence, `FULL_EXTENT_EXPLICIT_AGREGATION`, say, and `PARTIAL_EXTENT_EXPLICIT_AGREGATION`, etc. For these partial extent cases, we refer to the elements that are related to the 'anchor type' as the *related types*. For example, in the case of JHotDraw, a candidate feature whose extent consisted of the methods `{Rectangle displayBox(), boolean containsPoint(int x, int y), void read(StorableInput dr), void write(StorableOutput dw)}`, included in its extent the abstract class `AbstractFigure` and some of its concrete subclasses, namely: PolygonFigure, PolyLineFigure, RectangleFigure, TextFigure, `DecoratorFigure`, which we called *related classes*, along with some unrelated classes.

We know, by definition (see section 5.2.1), that the shared behavior among the *anchor type* (`AbstractFigure`) and its *related types* is a *subset* of the intent– since together they represent a *subset* of the extent. We got curious as to the relative size between that shared behavior and the intent. Specifically, for such "PARTIAL_EXTENT" cases, we computed the ratio: 
\[
\text{CONFIGURATION BEHAVIOR COVERAGE} = \frac{\text{size(related behavior between anchor type and related types)}}{\text{size(intent)}}
\]

For the case mentioned above, the `CONFIGURATION_BEHAVIOR_COVERAGE` equalled 0.8. This means that the four methods that make up the intent represent 80 percent (4/5ths) of the shared behavior between `AbstractFigure` and those of its subclasses that are included in the extent (see footnote). This is to be expected in general: if the Galois lattice construction algorithm groups in the same node a 'configuration' (an interface with its implementations or a class with its subclasses), *along with unrelated classes*, the intent will be subset of the shared behavior within the ‘configuration’. However, that is not always the case: there are cases where the intent of a candidate node is the same size as the common behavior of a partial configuration (e.g. a class and its subclasses). This could happen if the designer did recognize the feature, and codified as a class or interface. But s/he (or some other project contributor) recoded the same functionality all over again in some other part of the application. Such cases would be as ‘insightful’ as a pure ad-hoc cases.

Thus, each one of our original three deliberate categories (see Figure 18) now yielded four subcategories each, corresponding to various combinations of _EXTENT and _BEHAVIOR coverage, namely, FULL_EXTENT_FULL_BEHAVIOR_*, FULL_EXTENT_PARTIAL_BEHAVIOR_*, etc.; we will refer to these combinations using the word initials, as in FEBF for FULL_EXTENT_FULL_BEHAVIOR; FEPB for FULL_EXTENT_PARTIAL_BEHAVIOR, and so forth. Further, we now have a metric to compute the behavior coverage of the intent relative the configuration common behavior.

Finally, an inspection of the FULL_EXTENT_* candidate nodes revealed sometimes surprising results. For example, one candidate node categorized as FEBF_EXPLICIT_CLASS_SUBCLASS_REDEFINITIONS had `AbstractFigure` as an anchor type, i.e. the extent consisted of `AbstractFigure` and (some of) its concrete subclasses. However, its intent consisted of a mere four methods: `Insets connectionInsets(), Vector handles(), void read(StorableInput dr), void write(StorableOutput dw)`, whereas `AbstractFigure` has thirty three methods, which are thus shared by its subclasses: how come the intent has only four of those? well, of course, in this case, the subclasses *need only redefine* the abstract methods of `AbstractFigure` ... and there only *four* of those. Thus, even in those cases where a particular configuration covers the entire extent, the intent may consist of only a fraction of the behavior of the anchor type. We measured that fraction by the ratio: 
\[
\text{ANCHOR_TYPE_BEHAVIOR_COVERAGE} = \frac{\text{size(intent)}}{\text{size(domain interface of anchor type)}}
\]

In summary, in addition to classifying candidate features into one of thirteen categories (ADHOC, and 4 combinations (FEBF_, FEPB_, PEFP_, PEFP_) for each of the three deliberate categories in figure 18), we do the following:

- we compute `ANCHOR_TYPE_BEHAVIOR_COVERAGE` for every candidate feature
- we compute `CONFIGURATION_BEHAVIOR_COVERAGE` for all PARTIAL_EXTENT_* (FEPB_ and PEFP_ configurations) candidate features.

Note that our program may produce several classifications/tags for the same candidate feature node. For example, a candidate feature node can have several partial configurations. For example, in JHotDraw, a candidate feature node with extent `{PointConstrainer, GridConstrainer, StandardDrawingView}` and intent the single method `{Point constrainPoint(Point p)}`, has two partial configurations:
- A **PEFB_EXPLICIT_INTERFACE_IMPLEMENTATIONS**, with anchor type (the interface) `PointConstrainer` and related type(s) = `GridConstrainer`,
- A **PEFB_EXPLICIT_AGGREGATIONS**, with anchor type (the component) `PointConstrainer` and related type(s) = `StandardDrawingView`.

For both partial configurations (sharing the anchor type), the anchor type behavior coverage is 0.33, i.e. the intent `{Point constrainPoint(Point p)}` represents one third of the full interface (thus, three methods) of `PointConstrainer`. Further, the same candidate node can have a **FULL_EXTENT_*_tags** as well as several **PARTIAL_EXTENT_*_tags**. Figure 19 shows the example of a candidate node that was assigned three tags: a **FULL_EXTENT_*_INTERFACE_IMPLEMENTATIONS**, with the interface `Figure` as an anchor type, meaning that all the other classes of the extent (the meaning of **FULL_EXTENT**) are implementations of `Figure`, then a **PARTIAL_EXTENT_*_CLASS_SUBCLASS_REDEFINITIONS** with anchor type `AbstractFigure` and all its descendants, and finally, a **PARTIAL_EXTENT_*_AGGREGATIONS**, again with anchor type `Figure`, which is a component of `FigureDecorator`, or, conversely, `FigureDecorator` is a wrapper of `Figure`. For the remainder of this section, we will use the following shorthands, unless specified otherwise:

- we will use the prefixes **PEFB**, **PEPB**, **PEFB**, **PEPB** to represent the four combinations full/partial extent/behavior
- **EXPLICIT_INTERFACE_IMPLEMENTATIONS** will be referred as **INT**
- **EXPLICIT_CLASS_SUBCLASS_REDEFINIRIONS** will be referred as **SUB**
- **EXPLICIT_AGGREGATION** will be referred to as **AGR**.

```
<table>
<thead>
<tr>
<th>Software</th>
<th># Cand</th>
<th># AD</th>
<th># INT</th>
<th># SUB</th>
<th># AGR</th>
<th># PART.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeMind</td>
<td>69</td>
<td>32</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td>JavaWebMail</td>
<td>50</td>
<td>23</td>
<td>0</td>
<td>5</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>154</td>
<td>54</td>
<td>41</td>
<td>18</td>
<td>0</td>
<td>41</td>
</tr>
<tr>
<td>JReversePro</td>
<td>27</td>
<td>25</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Lucene</td>
<td>91</td>
<td>47</td>
<td>4</td>
<td>23</td>
<td>4</td>
<td>13</td>
</tr>
</tbody>
</table>
```

The following table shows a summary of results for the five packages. The last five columns represent five disjoint subsets. In other words, the column ‘PART.’ (for partial configurations) refers to cases where the extent of a node could not be accounted for, in full by any of the previous three configurations. Thus, a case such as that shown in figure 19 will be counted in the **INT** column (full extent, full behavior, INTERFACE implementation), and not the ‘PART.’ column, even though the node has two **PARTIAL_EXTENT_*_tags**.

The table 11 shows the distributions of the different categories for the five packages. If we associate a high percentage of **ADHOC** (or low percentage of **PEFB_*_nodes**) candidate features with low code factorization/maturity, then we can easily conclude that:

1) **JHotDraw** is the most mature/best ‘factorized’ software package, with 35% **ADHOC** candidate features, 38.32% full extent, full behavior candidates, and 26.62% partial configurations,
2) **JReversePro** is the least mature/factorized software package, with 92.6% **ADHOC** candidate features, and 7.4% full extent, full behavior candidates,
3) **FreeMind**, **JavaWebMail**, and **Lucene** are somewhere in the between, with **Lucene** as the odd one out, with second highest **ADHOC** percentage (51.65%) but also second highest full extent, full behavior percentage (34.05%).

At first glance, these results make sense. At one end of the spectrum, we have **JHotDraw**, which was developed as a case study in design patterns, and the version we studied (5.1.2) means that the design had gone through several iterations already. Thus, its designers/contributors had had plenty of opportunities to recognize common functional
features, and to encode them as such, using interfaces, abstract classes coupled with class inheritance, and aggregation. By contrast, JReversePro is a package developed by two researchers, with a very specific/narrow focus: ‘reverse engineering’ (decompiling) Java bytecode. FreeMind and JavaWebMail were the work of several contributors (which pleads for better factorization) and implement relatively complex, multimodal applications (which plead for poorer, because more complex, factorizations). Such conclusions depend, in part, on the ‘quality/insightfulness’ of the candidate features that were tagged ADHOC— and to a lesser extent, PARTIAL_EXTENT_∗.

In the next few sections, we take a closer look at the packages, separately.

<table>
<thead>
<tr>
<th>Software</th>
<th>% AD</th>
<th>% INT</th>
<th>% SUB</th>
<th>% AGR</th>
<th>% PART.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeMind</td>
<td>46.38</td>
<td>1.45</td>
<td>8.7</td>
<td>8.7</td>
<td>34.78</td>
</tr>
<tr>
<td>JavaWebMail</td>
<td>46</td>
<td>0</td>
<td>10</td>
<td>14</td>
<td>30</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>35.96</td>
<td>26.62</td>
<td>11.69</td>
<td>0</td>
<td>26.62</td>
</tr>
<tr>
<td>JReversePro</td>
<td>92.6</td>
<td>3.7</td>
<td>3.7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Lucene</td>
<td>51.65</td>
<td>4.4</td>
<td>25.27</td>
<td>4.4</td>
<td>14.3</td>
</tr>
</tbody>
</table>

### 7.2 jHotdraw

Out of 154 candidate feature nodes, JHotDraw had 54 nodes tagged at ADHOC (no relationships between the classes of the extent), 41 as configurations consisting of an interface and a subset of its implementing classes, covering the full extent (FEFB_INT), 18 configurations consisting of a class and some of its subclasses, again covering the full extent (FEFB_SUB), and 41 candidate features where the extent contained one of the previous two configurations, along with unrelated classes. Ultimately, the usefulness of our technique resides in the usefulness of the ADHOC nodes (54) and the partial configuration nodes (41) to the extent that those reveal behavioral commonalities (features) that were not recognized by the developers. However, for the purposes of this paper, we will examine all the categories to get a better understanding of what our algorithms are capable of identifying, and to provide some context for the results.

With regard to the interface implementation configurations, the 41 configurations corresponded to the following interfaces:

- **Figure**: 18 nodes/configurations in all, each one consisting of Figure, along with different subsets of its implementing classes, where each configuration shares a different set of methods (intents).
- **Tool**: 10 nodes/configurations in all, each one consisting of Tool, along with different subsets of its implementing classes and different intents
- **Handle**: 8 nodes/configurations, same as above
- **Connector, DrawingEditor, FigureEnumeration, Locator and Painter**: with one node/configuration, each.

For one given interface, referred to as anchor type in section 7.1, the nodes are hierarchically linked in relatively ‘deep’ hierarchies (see elliptical/yellow nodes in figure 20), where the children of a node A whose extent consists of an interface and a set of its implementing classes, have an extent that includes the same interface, but smaller subsets of implementing classes— and naturally, sharing a larger number of methods than the classes in node A. For example, in figure 20, the extent of node 40 contains the interface Figure, the class AbstractFigure, which implements Figure, and a bunch of descendants of AbstractFigure. Children of node 40 (nodes 41, 50 through 55, 66, and 67) have extents that are subsets of the extent of node 40, that keep the interface Figure.

With regard to the class-subclass configurations (full extent, full behavior), the 18 nodes are centered around five classes (five different anchor types) as follows:

- **AbstractFigure**: 13 nodes/configurations, i.e. nodes whose extents correspond to 13 different sets of descendants of AbstractFigure
- **Command**: 2 nodes/configurations, and
- **DrawApplet, DrawApplication, and ChopBoxConnector**: one configuration each.

Interestingly, many such configurations are direct or indirect descendants of interface implementations configurations. Referring back to Figure 20, 12 out of 13 class-subclass configurations are descendants of interface implementation configurations. This is no surprise. If we look at the candidate feature node illustrated in figure 19, a subset of its extent that does not include the interface Figure will be a class-subclass configuration.

With regard to the adhoc nodes, as mentioned in section 7.1, 1) a node that has been tagged ADHOC will have all of its descendants tagged ADHOC, and 2) many ADHOC nodes will be children of deliberate (partial or full) configurations. Both are illustrated in Figure 20: many ADHOC (diamond) candidate nodes hang from deliberate configuration nodes (ellipses of squares), and naturally, the descendants of an ADHOC node can only be ADHOC nodes. Indeed, if we take a class-subclass configuration, and we remove the root class from the extent, as in removing the class AbstractFigure from a node whose extent consisted of AbstractFigure and some of its subclasses, we get an ADHOC node. How could AbstractFigure be removed from the extent as we go down the lattice? Remember that node extents are supposed to be purged of non-minimal classes, with respect to the class-subclass relationship, prior to identifying candidate features, and a class such as AbstractFigure will only ‘survive the purge’ if it implements locally all of the methods of intent. However, as we go down the lattice, we are bound to find nodes whose extents include methods not directly implemented by AbstractFigure, and hence, AbstractFigure is purged from the extent of such nodes, and then we get ADHOC nodes.

Note that all of the nodes shown in Figure 20 are induced by the Figure abstraction/feature: 18 interface-implementation nodes, 12 class-subclass nodes, and 11 ADHOC nodes. This, not counting partial configuration nodes, which are descendants of the full configuration nodes of Figure 20.

A study of JHotDraw’s 54 ADHOC nodes revealed three categories:
Nodes that are descendants of full/partial configuration candidate nodes, such as the diamonds of Figure 20. There are 39 of those. The question is, how interesting are such ADHOC nodes likely going to be? We will attempt an answer below.

Nodes that are induced by types that have more than one independent ancestors. For example, the interface ConnectionFigure extends both Figure and FigureChangeListener, which are not related hierarchically, and do not share any methods. Thus, our incidence relationship will count the API (we called it domain interface) of ConnectionFigure, and its implementing classes, as two independent occurrences since we arrive at it (through ‘reverse inheritance’) from both Figure and FigureChangeListener. Accordingly, we ended up with a candidate ADHOC node whose extent consists of the interfaces {Figure,FigureChangeListener}, and whose intent includes the union of the methods of the two interfaces18. A similar situation occurs between an interface I and any class C such that one of its subclasses—call it D—implements I; in this case, we will have a node (candidate feature) whose extent includes the pair {I, C }, and whose intent includes the methods found under the D subhierarchy. Those are clearly ‘false positives’ because the ‘multiple occurrences’ of a feature are, in reality, the same occurrence, but arrived at from different paths. For JHotDraw, we identified six such nodes.

Nodes that represent truly fortuitous co-occurrences.

For JHotDraw, there were nine (9) of those.

As a first approximation, if we wanted to focus the attention of the design on the candidate nodes that are potentially interesting, we would:

- Disregard the ADHOC nodes that are descendants of nodes representing deliberate configurations (e.g. the diamonds in Figure 20), as such nodes will reflect commonalities between classes that are known to extend the same class or implement the same interface.
- Disregard the ADHOC nodes that are due to types having multiple ancestors, as those are clearly false positives. Actually, our candidate feature identification algorithm could be modified to filter out such false positives19.
- Focus on the remaining ADHOC nodes—9 for the case of JHotDraw.

The study of those 9 nodes revealed the following:

- 7 nodes out of 9 revealed interesting abstractions that were not identified by the designers/developers of JHotDraw 5.2:
  - the concept of a figure having an indexed set of points, which appears in ‘cousin classes’ PolylineFigure and PolygonFigure (see node 5 in appendix ??)
  - the concept of connection, as distinct from Connector (see nodes 18, 19 and 20 in appendix ??).
  - the concept of indexable storable object (see node 21 in appendix ??).
  - similarity—and redundancy—of sibling classes DrawingChangeEvent and FigureChangeListener (see node 24 in appendix ??).
  - greater similarity between the classes DrawApplet and DrawApplication,

19. This can be done by implementing a sort of reference counting for elements of the intent where we count the number of independent paths to a particular method, starting from the elements of the extent. Any method (intent element) that has a reference count of 1 should be removed because a count of 1 means that it ‘has occurred independently only once’
much beyond what is embodied in the DrawingEditor interface that is implemented by both classes (see Figure 22, and node 110 in appendix ??).

- Two nodes represented ‘uninteresting concepts’ where the algorithm was ‘misled’ by the multiple occurrence of ‘secondary’ methods (see nodes 128 and 135 in appendix ??).

With regard to the FEFB_SUB candidate node, it has anchor type JBlockObject, with anchor type coverage 0.17, and subclasses such as JCaseBlock, JCatchBlock, JSwitchBlock, JDoWhileBlock, etc, and intent {String getEntryCode(), String getExitCode()}, which return the beginning and ending string of each block type. Interestingly, the extent consists of the entire contents of the class package jreversepro.reflect.method. The relatively ‘low’ anchor type coverage (0.17), which represents the ratio between the size of the intent of the node (2 methods) and the domain interface of the anchor type (JBlockObject) could be interpreted in one of two ways:

- as a sign of good factorization: most of the behavior (83% of it) of the classes under JBlockObject was encoded in JBlockObject itself, leaving little to be redefined in the subclasses, or
- as a sign of poor factorization: there are few (17%) common methods between JBlockObject and its subclasses.

Which is which? we have to look at the domain interfaces of the subclasses of JBlockObject: if all they implement is the two methods of the intent, then we have an instance of good factorization. If, on the other hand, each subclass of JBlockObject implements a different, larger set of methods, sharing only two methods in common, then we have a case of a poor factorization/abstraction. In this particular case, ten out of twelve subclasses of JBlockObject implement just the methods of the intent. However, the classes JForBlock and JDoWhileBlock define far more methods than the intent; they both represent iterators who have more complex structures. Appendix ?? contains a detailed analysis of the 25 ADHOC candidate nodes. For the purposes of this section, we summarize the findings:

- 9 ADHOC candidate nodes that consisted of nodes whose extents consisted of two or more interfaces, and whose intents consisted (roughly) of the union
of the domain interfaces of the classes that implemented those interfaces. For example, if we have interfaces Interface1 and Interface2 that were both implemented by classes Classe1 and Classe2, then both Classe1 and Classe2 will appear under the type hierarchies with roots Interface1 and Interface2—that is how JDT does it—and all the methods of Classe1 and Classe2 will appear as if they occurred twice. We had similar false-positive nodes with JHotDraw—six of them. Such false positives can be filtered a-posteriori, as mentioned earlier (see subsection 7.2).

• 3 ADHOC nodes that corresponded to interesting abstractions that were clearly not identified/recognized by the developers; see nodes 12, 20, and 22 in the Appendix.

• 6 ADHOC nodes that corresponded to abstractions that were visibly recognized by the developers, but that were not implemented in a way that exploited the commonalities. This was sort of a deliberate code cloning (nodes 16, 17, 18, 21, 24, and 25 in Appendix). The six nodes had extents consisting of pairs of graphical classes, with different class names, but belonging to different packages, that offered the same functionalities, once using the java.awt.* library, and once using the javax.swing.* library. Surely there are other ways of handling this, without resorting to code duplication, e.g. using the bridge or factory patterns.

• 6 ADHOC nodes that corresponded to ‘mildly interesting’ abstractions that, perhaps, did not warrant explicit codification; see nodes 8, 11, 14, 15, 19, and 26 in the Appendix.

7.4 Analysis of the results

From the categorization of candidate nodes (section 7.1), and the results for JHotDraw (section 7.2) and JReversePro (section 7.3), the following observations can be made:

1) The categorization of nodes, and the accompanying metrics (configuration behavior coverage, and anchor type behavior coverage) provided us with a fine and insightful characterization of the ‘state of factorization’ within the different software applications, and a good basis for comparing them in terms of design maturity. Unsurprisingly, JHotDraw, the GUI development framework meant as a case study in design patterns came out as the most mature of the five applications, and JReversePro, a research prototype for reverse engineer Java bytecode developed by two researchers, came out as the least mature one.

2) The hierarchical representation of candidate feature nodes, such as the ones in Figures 20 and 23 provide a good insight into the ‘factorization landscape’ within a software package, and enables us to focus on those nodes that are most-interesting to explore—namely, ADHOC nodes that are not descendents of deliberate configuration nodes. This observation should guide the design of tool support for functional feature mining.

3) Some counter-intuitive insights about the quality of factorizations within class hierarchies, thanks in part to the anchor type behavior coverage metric (see section 7.1 and to the results for JReversePro discussed in section 7.3). This led us to speculate on what ‘good factorization’ looks like within a class subhierarchy. We will argue that in the lower hierarchy levels, a ‘good factorization’ is a combination of, a) only a small fraction of the behavior of the root of the subhierarchy is redefined by its descendants (anchor type behavior coverage metric), but at the same time b) descendants define little additional behavior to the one of the root; a too small anchor type behavior coverage metric, with no/little additional behavior by the subclasses, and one might question the need for separate subclasses, a too big (say, over 0.5) anchor type behavior coverage, or ‘too much’ extra behavior defined by the subclasses and one might question the quality of the factorization.

4) Notwithstanding false positive ADHOC nodes induced by classes that are reachable from more than one type from the extent, which can be filtered out automatically (see discussion in section 7.2), and ADHOC nodes that descend from deliberate configuration nodes (which are readily identifiable), most of the ADHOC nodes (7 out of 9 for JHotDraw, and 9 out of 15 for JReversePro) represent ‘interesting abstraction’ that either had been missed by the developers. This suggests that a code browsing tool that focusses the developer’s attention on such nodes will, more often than not, identify interesting abstractions that may warrant refactorings.

More general issues, not specific to ADHOC implementations, will be discussed next (section 8).

8 DISCUSSION

In this section, we discuss the results of the experiments

8.1 Are the methods complementary?

The short answer is: yes, the methods are complementary, in two ways. First:

• it pays to uncover those functional features that have been codified in the program, using aggregation or multiple inheritance–real or simulated. The reason for that is that such information gives the developer a sort of cartography of the application, giving developers the various ‘dimensions’ that can be combined to build new applications

• it pays to uncover abstractions that have not been identified—and codified—by the developers, such as the ADHOC candidate features discussed in section 7, and these could guide future refactorings of the application, as illustrated in sections 7.2 and 7.3.

Second, regarding the deliberate features that have been codified by developers, the algorithms presented in sections 3, 4, and 5 are also complementary: they produce intersecting but different sets of features:
For example, for JHotdraw, the algorithms in sections 3 and 4 identified 24 types, whereas the one of section 5 identified 13 types induced by FEFB* configurations, only 6 of which are shared with the algorithms of sections 3 and 4.

### 8.2 Does structure matter

Going back to section 5.1, we envisioned a functional feature as a set of class members (data and function members) that are distributed between a number of classes, that together implement a particular function. Our intuition has been that the structural distribution of the different attributes and methods that contribute to a given feature mattered, as illustrated by the examples in Fig. 5 and 6. However, notwithstanding the complexity and computational intractability of a structure-sensitive incidence relationship, we felt that configurations such as the ones illustrated in Fig. 5 were highly unlikely to occur in practice. Hence, we proposed a simplified incidence relationship which associated each class with the set of attributes/operations that were defined in the class or any of its descendants, as explained in 5.2.2; we called this relationship reverse inheritance (see section 5.2.2).

In earlier work, we used Galois lattices on class interfaces to identify optimal factorizations of class hierarchies [20]. The example of Figure 22 shows a situation where our current algorithm uncovered an ‘interesting abstraction’ consisting of a set of methods that occurred directly in the classes of the extent (DrawApplet and DrawApplication), i.e. without resorting to ‘reverse inheritance’. Which raises the question: how likely are we to find multiple occurrences of features—i.e. sets of methods—that are spread over several classes. In other words, do we really need reverse inheritance? couldn’t we identify the same features just by looking at the local interfaces of classes?

The example of Figure 22 notwithstanding, the answer is: we do need reverse inheritance; the sets of methods that constitute a legitimate features will often be spread over several classes. Figure 24 illustrates the situation, where we showed methods corresponding to a subset of the intent. This is the candidate ADHOC node that ‘recognized’ the concept of a connection, as locus of linkage between figures that react to figure change events. In the right-hand ‘occurrence’, we get the ‘connection point’ behavior from PolyLineFigure, and the ‘changeability’ from the subclass LineConnection. In the left-hand ‘occurrence’, the same methods are spread over three classes, two of which are not hierarchically related. Based on the right-hand occurrence alone, one could think of using the local-inherited interface of classes for the incidence relationship. However, for a set of local-inherited methods to count as two independent occurrences, we should disregard the methods that are inherited from a common ancestor. Which means that the inherited methods will only be counted if they belong to distinct inheritance paths, i.e. different class subhierarchies. That is pretty much what we are doing by, a) using what we called ‘reverse inheritance’, and b) disregarding co-occurrences where the roots of the subhierarchies are hierarchically related (see 5.2.2).

![Fig. 24: Structure matters. The concept of connection is spread over several classes.](image)

### 8.3 Towards a metaphor for functional feature detection

In this section, we answer the more general question: what do to about all the ‘insights’ produced by the program? Recall that, broadly speaking, the set of algorithms presented in this paper do two things: 1) provide a cartography of the various functional ‘dimensions’ of the application, and 2) identify additional opportunities for factorization, refactoring, and reuse– the so-called ADHOC feature nodes. In section 7, we talked about two general classes of ADHOC candidate nodes/features:

- Those that are descendants of deliberate configurations are most likely not interesting, as they result from the combinatorics of associating sets of classes to their shared set of functions, that is inherent in FCA.
- Those that are not descendants of deliberate configurations: our experiments showed that most of those embody interesting abstractions that have been missed by the designers. We will refer to them as independent ADHOC nodes.

We see our algorithms embedded/embodied in a software design and maintenance tool that analyzes the source code of applications and:

- Identifies and displays the deliberate, designer recognized, classes and interfaces that embody reusable functional features. Such classes and interfaces
would have been identified by the algorithms of sections 3, 4, and as anchor types by the algorithms in section 5.

- Identifies and displays the independent ADHOC nodes.
- Selecting a class or interface that participates in a multiple inheritance `configuration' (Section 3) or aggregations (Section 4), would display the configurations in which it occurred, similar to the examples shown in section 6 (e.g. 15, 16, 17)
- Selecting an anchor type would display the hierarchy of deliberate candidate features induced by that type, similar to the hierarchies shown in section 7 (e.g. figures 20 and 23), but ignoring the descendant ADHOC nodes.
- Selecting a deliberate feature node would display the corresponding configuration, along the lines of Figure 19, along with its metrics.
- Similarly, selecting an independent ADHOC node would display its configuration, similar to the example of figure 22.

The tool could also display some global metrics, for the entire software application. We reproduce below a synthesized version of Table 11 where we grouped in a single column the deliberate configurations (FEFB_INT, FEFB_SUB, FEFB_AGR) for the five software applications we evaluated. Such a table can help compare applications or subsystems, among each other, and compare them against quality benchmarks. Such a tool would have several advantages:

- Provides a synthesized indicator—if not a metric—of the quality of factorization within applications.
- Requires no developer input. This is due to two important characteristics. First, unlike most feature location work (see Section 9), our goal is not to locate a feature we know the product has, but to identify the independent features the product might have. Second, our algorithms rely on programming language semantics and formal concept analysis. Thus, we do not depend on implicit semantics, such as variable names and comments, or the ambiguities of natural language.
- Our experiments have shown that most of the independent ADHOC nodes embody interesting abstractions that may warrant refactorizations.

All of these should make such a tool a valuable design and maintenance aid.

### 8.4 Threats to validity

There are potentially three threats to the validity of the results reported in this paper: 1) what constitutes an ‘interesting’ or ‘useful’ functional feature (construct validity), 2) whether we are reliable judges of that (internal validity), and 3) whether the results can be generalized to other software packages (external validity).

With regard to the first question, we have argued elsewhere that reusability is a combination of usefulness and usability [21] where usefulness refers to the extent to which a software artifact is often used, and usability refers to the ease with which it can be used. The algorithms for identifying cases of multiple inheritance and delegation identify artifacts that are de-facto usable: classes and interfaces. Thus, our evaluation had to focus on usefulness, which roughly means in this context that it reflects meaningful domain abstractions. We were careful, both in our algorithms (e.g. the concept of domain interface) and our analyses, to focus on domain semantics, and we highlighted cases where the identified features were secondary to the domain in question (e.g. GUI features for the case of JReversePro and FreeMind, see section 7.4). Further, the usefulness of many of such features was independently confirmed by their multiple occurrences within the code, since many of the same features were also uncovered by the algorithm for adhoc features (see sections 7.4 and 8.1). With regard to the adhoc features, the usefulness comes de-facto, since all candidate features (notwithstanding false positives than can be filtered algorithmically) did occur multiple times, and the usability was not an issue. Thus our evaluation had to focus on the functional cohesion of the set of methods in each case, which was presented in the appendices.

With regard to the reliability of the authors in assessing the quality of the features, we should stress that the proper interpretation of the results requires: 1) a deep knowledge of design patterns and coding practices, and 2) good familiarity with the software packages—or willingness to delve into their design. Earlier attempts to have graduate students evaluate the results failed, due to a lack of 1) or 2)—or both. The authors have, at one time or another worked with the packages studied for other reasons (e.g. we used JReversePro to implement a slicer [3]), and we have developed an intimate knowledge of the packages studied here.

In terms of the generalizability of the results, clearly, more experiments are needed. However, we were careful in choosing our data set to cover a broad spectrum of software packages in terms of, 1) size (10 KLOC versus 70 KLOC), 2) maturity (JHotDraw versus JavaWebMail), 3) framework (JHotDraw versus monolithic application (JReversePro), 4) mostly graphic (JHotDraw, FreeMind) versus mostly command line (JReversePro), and 5) system (JReversePro) versus end-user application (JavaWebMail). The results of our preliminary experiments enabled us to automate a good part of the analysis of the results (e.g. categorization of candidate features), and focus on the interesting ones—a few dozen nodes per software application (see appendices ?? and ??). This should enable us to test our algorithms on more and bigger applications.

<table>
<thead>
<tr>
<th>Software</th>
<th>% ADHOC</th>
<th>% FULL CONF</th>
<th>% PARTIAL CONF</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeMind</td>
<td>46.38</td>
<td>18.85</td>
<td>34.78</td>
</tr>
<tr>
<td>JavaWebMail</td>
<td>46</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>35.06</td>
<td>38.31</td>
<td>26.62</td>
</tr>
<tr>
<td>JReversePro</td>
<td>92.6</td>
<td>7.4</td>
<td>0</td>
</tr>
<tr>
<td>Lucene</td>
<td>51.65</td>
<td>34.05</td>
<td>14.3</td>
</tr>
</tbody>
</table>

**TABLE 12**

Percentages of candidate feature categories


9 Related work

Large software applications typically implement a tangled web of functional requirements, and maintaining such applications is notoriously difficult. Researchers have long been interested in providing maintainers with help in identifying the various functional features of a program, and, when those features are known, in circumscribing them in the source code. Work on code slicing (e.g. [22]), feature extraction, concept assignment (e.g. [23], and the like has been ongoing for over to thirty years (see e.g. [24], [25]). A program feature can be seen as a triple \(<name, intension, \text{extension}>\) [25], [26], where ‘name’ is a shorthand to refer to the feature (e.g. ‘journaling’), ‘intension’ is some more or less precise or formal description of the feature (formal specifications, or textual description, as in a feature request or bug report), and ‘extension’ is the set of software artifacts that implement the feature. The work reviewed in [24], [25], for example, explored eliciting one component of the triad given the other (two).

Work on feature location can be divided in many ways. For the purposes of this paper, we identify two major classes, based on the problem that we want to solve:

- Work that aims at locating features that a software application is known to have (e.g. [14], [23], [27], [28]). To use the above triple \(<name, intension, \text{extension}>\), this is the case where the intension is known, the name may or may not be known (it does not matter), and the purpose is to find the extension. This is the true feature location problem in the strict sense, where the location problem can be seen as eliciting the function: \(\text{intension} \rightarrow \text{extension}\) (25).

- Work at discovering features in the code (e.g. ). In this case, roughly speaking, we try to identify the presence of features of a particular type in the code based on the typical signature that features of that type tend to have in the code. This can be seen as eliciting the reverse relation of feature location, namely, finding intensions based on extensions: \(\text{extension} \rightarrow \text{intension}\).

A significant share of feature location work dealt with the first problem. Our work fits squarely into the second category: identifying potential (functional) features within existing software.

Work on feature location has traditionally used one, or a combination, of the following types of analyses, in part based on the available information:

- Static analysis of the source code (see e.g.): the idea being that the artifacts that contribute to a feature tend to be functionally and control dependent (one artifact calls, or references another). Thus, given an artifact known to “belong” to feature—sometimes referred to as seed, by following forward or backward reference links, we are likely to find the other artifacts.

- Dynamic analysis of program execution traces (see e.g. [14], [15], [28], [29], [30]). Because static analysis typically captures more dependencies than is usually exercised in general, and while executing a specific feature, an execution of the application that invokes the feature will be limited to those program elements that are actually exercised for that feature ... but only for the specific inputs that produced it.

- Natural language processing techniques, to match a textual description of the intension of a feature with textual information taken from the software artifacts (see e.g. ). Indeed, to the extent that developers choose program identifiers that reflect the underlying semantics, or that they document their code, it is expected that the underlying functional semantics of artifact be reflected in the textual information used, such as identifier names and comments.

Each one of these techniques has its limitations [24], [25]. Modern programming languages, with late/dynamic binding and computational reflection, and application frameworks, with implicit dependencies, make the analysis of applications difficult [25], [31]. Dynamic tracing is complicated to set-up [25], especially in multi-tier applications [30]. Natural language processing techniques suffer from two inherent problems, 1) the ambiguities of natural language, and 2) our assumption that developers voluntarily properly document their code, and astutely choose their identifiers to reflect function/application semantics at the appropriate level of abstraction—e.g. domain semantics as opposed to design patterns [28]. Many of the reported approaches use a combination of techniques, which were shown to show better results than single-technique approaches (see e.g. [25], [32]), but require varying degrees of developer input, especially for the feature location problem (intension \(\rightarrow\) extension). Our work focuses on static code analysis, but without the complexities of method call analyses—we rely solely on signatures—and require no developer input.

While these techniques have been used for both feature location (intension \(\rightarrow\) extension) and feature identification (extension \(\rightarrow\) intension), natural language processing and information retrieval (IR) techniques have mostly been used for feature location when the intension of a feature is provided as natural language text, as in a feature request or a bug report. Conversely, formal concept analysis (FCA), which is an automatic classification technique, has been used extensively for feature identification (extension \(\rightarrow\) intension) (e.g. [14], [15], [16], [33], [34], [35], [36]). Given a binary relation \((X,Y)\)—called a context—FCA identifies a set of concepts, which are pairs of sets \((E,I)\) where \(E \subseteq X\) and \(I \subseteq Y\), such that \(E\) is the largest subset of \(X\) whose elements are related to all the elements of \(I\), and \(I\) is the largest subset of \(Y\) whose elements are all images of the elements of \(E\). Researchers have been creative about what these sets represent; usually, one of the sets \((X\ or \ Y)\) represents software artifacts (classes, methods, etc.) while the other represents directly or indirectly, feature intensions. FCA has even been used to detect features within variants of a product family within the context of software product line engineering (see e.g. [35], [36]). In our case, \(X\) is the set of classes, and \(Y\) is the set of class features, and the relation associates a class with the set of features that appear in the subhierarchy under the class.

The advent of aspect-oriented programming [7] has increased interest in the feature identification problem (extension \(\rightarrow\) intension), but also the use of FCA, in part because the feature we are looking for tend to cross-cut the other
functional features of the code (see e.g. [37], [38], [14], [29], [39], [40], [15], [2], [41], [42], [16], [32]). Most of the aspect-mining work focused on cross-cutting features that correspond to infrastructure or architectural services. This influences both the granularity of the analysis– method code– and the type of patterns that we look for– for example, performing a fan-in analysis [39] or looking for code clones [40]. Our definition of functional features corresponds best to [14] features, which embody externally visible behavior of the objects.

10 Conclusion and Future Work

Our work aims at developing methods for mining functional features from OO code. To this end, we explore various hypotheses about how developers, in the absence of aspect-oriented programming abstractions, would implement several functional features within the same code base. In this paper, we studied three such hypotheses, which fall into two general categories:

- The developer/designer had recognized the functional feature as such, and packaged it in a way that lends itself to separate maintenance and reuse (as a class or interface), and composed it using OO design/programming idioms. We identified two such idioms in the OO context: a) composition through multiple inheritance–or a simulation thereof, and b) composition through aggregation. We referred to those as deliberate features.

- The developer/designer has not recognized the functional feature as a potentially usable cohesive set of functionality, but ended up coding it in the different places where it was needed. We referred to those as adhoc features.

We developed algorithms that rely solely on class and method signatures to uncover deliberate features (see sections 3 and 4), and adhoc features (section 5), and tested them on five open source packages (see sections 6 and 7). Our results showed that:

1) Notwithstanding false positives, which can be readily (programmatically) filtered out, multiple inheritance and composition through aggregation were overwhelmingly used to compose legitimate functional features (see sections 6.2 and 6.4)

2) Our algorithm for uncovering adhoc implementation and composition of functional features (see section 5) enables us to identify potentially useful functional features among so called independent adhoc nodes, but also some deliberate features not following the multiple inheritance/aggregation patterns (see 7)

3) The two sets of algorithms are complementary, and should be made available to designers to help them, a) get a cartography of the design/functional dimensions of the application, and b) identify opportunities for refactoring and reuse (see 8.3)

4) The outputs of the algorithms provide indicators of the design maturity of the software application under study.

This work is being extended in many directions. First, we are looking at ways of validating the adhoc features a-posteriori by checking if the set of methods (intent) of an ADHOC feature ended up embodied in a deliberate feature in subsequent versions of the applications. Indeed, we have access to the historical versions of the five software packages, and we should be able to trace the evolution of the different kinds of features (see 7.1) throughout their revision history. We are also in the process of packaging the algorithms described in this paper within the context of a designer workbench, along the lines of the discussion in section 8.3, developed within the Eclipse ecosystem.

Acknowledgments

This work is made possible through a grant from Canada’s Natural Sciences and Engineering Research Council (NSERC).

References


Hafedh Mili Hafedh Mili is a full professor of Computer Science at the University of Quebec in Montreal.

Amel Elkharraz Amel Elkharraz is a professor at the CEGEP Bois de Boulogne, in Montreal. Part of this work was performed within the context of her doctoral research.

Petko Valtchev Petko Valtchev is an Associate Professor of Computer Science at the University of Quebec in Montreal.