Discovering Reusable Functional Features in Legacy Object-oriented Systems

Hafedh Mili, Imen Benzarti, Amel Elkharraz, Ghizlane Elboussaidi, Yann-Gaël Guéhéneuc, Petko Valtchev

Abstract—Typical object-oriented (OO) systems implement several functional features that are interwoven into class hierarchies. In the absence of aspect-oriented techniques to develop and compose these features, developers resort to object-oriented design and programming idioms to separate features as well as possible. Given a legacy OO system, discovering existing functional features helps understand the design of the system and extract these features to ease their maintenance and reuse. We want to discover candidate functional features in OO systems. We first define functional features and then discuss the footprints that such features are likely to leave in an OO system. We identify three such footprints: (1) multiple inheritance, (2) delegation, and (3) ad-hoc. We develop a set of algorithms for identifying such footprints in OO code and implemented them for the Java language using Eclipse JDT. In this article, we present the algorithms, and the results of applying the corresponding tools on five open-source systems: FreeMind, JavaWebMail, JHotDraw, JReversePro, and Lucene. Our experimental results show that: (1) the different algorithms can identify interesting and useful candidate functional features in OO systems, (2) they can identify opportunities for refactoring, and (3) they are complementary and could help developers.

Index Terms—Functional feature, feature discovery, multiple inheritance, delegation, formal concept analysis.

One who can do more can do less.
—Aristotle

1 Introduction

Nontrivial software systems typically implement many functional requirements. For example, a personnel-management system might include a *payroll functionality*, dealing with salary scales and hours worked, and a *production-planning functionality*, dealing with qualifications and availability. Object-oriented programming (OOP), which modularises functionality around the data, would lead to these two functionalities being interwoven in the same classes. Indeed, if techniques to declare and combine cross-cutting concerns (CCCs), i.e., aspect-oriented programming (AOP), are not used, then a single class, e.g., `Employee`, would include separate attributes and methods to support both payroll and production planning.

Given an OO *legacy* system, developed pre-CCC techniques, it is necessary to *(re)discover* features that are interwoven in the same class hierarchy (1) to understand the system, (2) to evolve the system, and–or (3) to migrate the system to new technologies, possibly using CCC techniques. We focus on *functional features*, which are *functionally-cohesive* and (relatively) *self-contained domain functionalities*. The functional features of interest are similar to *subjects* in Harrison and Ossher's *subject-oriented programming* [1] because they pertain to the user domain. They are different from *implementation* CCCs, which can be implemented using AOP techniques, such as AspectJ for Java [2], because these pertain to the implementation, e.g., logging.

A functional feature can be defined as a triple $\langle$ *name, intension, extension* $\rangle$ [3, 4], in which *name* is a shorthand for the feature (e.g., payroll), *intension* is some more or less precise description of the feature, e.g., formal specifications or textual description (e.g., a feature request or bug report), and *extension* is the set of program elements, e.g., fields, and methods that implement that feature.

Most existing feature-identification approaches deal with locating a feature extension given its intension and locating a feature can be seen as the function: *intension* $\rightarrow$ *extension* [4]. They use various analysis techniques ranging from static analysis to dynamic analysis to natural language processing, or a combination thereof. Each of these techniques has limitations [5], among which: (1) they need developers' inputs, (2) they require custom parameterisation and (3) many focus on implementation CCCs.

Our work deals with discovering potential functional features in existing systems: identifying a potential extension in the source code and finding its intension, which can be seen as the function *extension* $\rightarrow$ *intension*. To avoid shortcomings of analysis techniques, we use static code analysis but rely solely on class and method signatures to discover features. Thus, our approach does not require developers' input and complex method-call analyses.

Our approach to discovering functional features in OO legacy systems includes five steps:

1) Define the OO *idioms* that developers *typically* use to implement functional features.
2) Characterise the *footprints* of these idioms in OO source code.
3) Develop *algorithms* that can identify these footprints in some Java source code.
4) Apply these algorithms to a *sample of OO systems* with different sizes, quality and maturity.
5) Assess *manually* whether the uncovered footprints correspond to functional features in the systems.

In Step 1, we distinguish between *deliberate* and *ad-hoc OO idioms* for implementing functional features. *De-*

liberate idioms correspond to the case in which developers (1) recognised a cohesive set of program elements as a *functional feature*, (2) packaged this feature in distinct OO *package-level constructs*, i.e., classes and interfaces, and (3) used OOP techniques to integrate it with the rest of the code—typically using various flavours of inheritance and delegation. The footprints of deliberate idioms are relatively easy to characterise (in Step 2) and recognise (in Step 3), although results of Steps 4 and 5 show that they are not as straightforward as they might first appear.

Ad-hoc idioms correspond to the more interesting case in which developers missed or failed to recognise either the functional cohesion of a set of program elements or their reuse potential, which would warrant separate packaging. The idioms describe sets of program elements at the class level, i.e., fields and methods, that appear *together* in *multiple locations* in the system. The presence of such an idiom, given proper definitions of *togetherness* and *multilocation* (in Step 2) is an indication of the presence of a functional feature.

In Step 2, we describe the footprints of deliberate and ad-hoc idioms in OO source code. We choose to consider only *class definitions*, i.e., field and method *declarations*, excluding method *bodies* for theoretical and practical reasons. Theoretically, functional features should not depend on particular implementation details, i.e., method bodies, but only on the implemented concepts, i.e., method signatures. Practically, we reduce the complexity and increase the applicability of our approach by considering only method signatures: we reduce the "search space" (possibly at the cost of precision, see Section 9), while increasing applicability to systems for which source code is unavailable, in one format or another (text or bytecode).

In Step 3, we propose algorithms to identify occurrences of deliberate and ad-hoc idioms. While some idioms are quite straightforward to identify, others require the use of Formal Concept Analysis (FCA) with which we define *togetherness*, as belonging to the same class sub-hierarchy and *multilocation*, as being two or more locations that are not hierarchically related. We present the definitions of FCA and our encoding of class hierarchies in concept lattices in Section 2 and our use of these concept lattices to discover functional features in Section 6.

In Steps 4 and 5, we apply our algorithms to a sample of OO legacy systems. Our results (Sections 7 and 8) show that:

- Our algorithms can identify all of the occurrences of deliberate idioms.
- Our algorithms can identify occurrences of the ad-hoc idiom and thus discover functional features that developers missed, including in mature Java systems, such as JHtotDraw.
- Our algorithms provide valuable insights into the implementation of the analysed systems, no matter the quality or importance of the discovered features. These insights represent refactoring opportunities to improve the local designs of the legacy systems if not their global designs [6].

Section 2 describes and compares related work, introduces FCA, and presents an encoding of class hierarchies in concept lattices. It also introduces the systems on which we apply our algorithms. Section 3 illustrates and contrasts three main idioms to discover functional features, called *multiple inheritance*, *delegation*, and *ad-hoc*. Sections 4, 5, and 6 describe our approaches to detect these idioms. They follow Steps 1 to 5 of our approach, with matching subsections. Sections 7 and 8 describe the qualitative and user validations of our approach and tool. Section 9 discusses our definitions, algorithms, and detection results as well as threats to their validity. Section 10 concludes with future work.

2 Background

We now discuss previous work and briefly introduce FCA before describing our encoding of OO source code into concept lattices, used in Section 6.

2.1 Related Work

Large software systems typically implement a tangled web of functional features and maintaining such systems is notoriously difficult. Researchers have long been interested in helping developers identify these features and, when those features are known, circumscribe them in the source code. Work on code slicing (e.g., [7]), feature identification (e.g., [8]), concept assignment (e.g., [9]) has been ongoing for over to thirty years (see e.g., [4, 5]).

Work on features can be divided into many ways. For this article, we observe two major categories:

1) **Feature Location:** work that aims at locating *known* features that a system is known to have (e.g., [9, 10, 11, 12]). The intension of a feature is known, its name may/may not be known but is not relevant, and the purpose is to find its extension. It can be seen as the function: *intension* $\rightarrow$ *extension* [4].
2) **Feature Discovery:** work that aims at discovering *unknown* features in a system (e.g., [13]). The extension of a feature is available, if identified, in the source code and the purpose is to find its intention and possibly give it a name. It can be seen as the function: *extension* $\rightarrow$ *intension*.

Many feature-identification works deal with the first problem. Our work fits in the second category: discovering potential functional features in existing systems.

2.1.1 Feature Location

Works on feature location and feature discovery used one of the following analyses or combinations thereof:

- Static analysis of the source code of systems, see e.g., [14]: it assumes that the program elements that implement a feature are functionally- and control-dependent (one element calls, or references, another). Thus, given an element known to participate in a feature, sometimes referred to as a *seed*, forward or backward references provide most program elements participating in the feature.
- Dynamic analysis of execution traces of systems, see e.g., [10, 12, 15, 16, 17]: it recognises that finding a seed program element is difficult and that, given a seed, static analysis returns more program elements than are really exercised when executing a feature.

An execution trace contains the program elements really needed for a feature, including those without statically-analysable references.

- Natural language processing techniques on the artefacts composing a system, see e.g., [12]: they assume that developers choose program identifiers reflecting the features. They match textual descriptions of an *intension* with textual information from the source code possibly implementing this *intension*. They expect that the function of program elements is reflected in their identifier and comments.

Each one of these techniques has its limitations [5]. Modern programming languages, with late or dynamic binding, reflection, and frameworks make the static analysis of systems difficult [18]. Execution traces are complicated to obtain [4], especially in multi-tier systems [17], and depend on user inputs. Natural language processing techniques suffer from the ambiguities of natural languages and inconsistencies in choosing identifiers [12].

Many feature-location works use a combination of techniques, which were shown to yield better results than single techniques alone, e.g., [4, 19]. They require varying degrees of developers' input, especially when addressing feature location, i.e., *intension* $\rightarrow$ *extension*.

Recent work has addressed feature location in relation to the identification of services and microservices. Jin et al. [20] proposed Functionality-oriented Service Candidate Identification (FoSCI), an approach that uses mainly execution traces to discover functional features. While execution traces can reflect precisely a system usage, they cannot completely reflect the system due to the (usually) small coverage of any scenario. Therefore, we choose to use the source code of systems rather than execution traces.

2.1.2 Feature Discovery

Feature discovery has been the topic of less research work than feature location. Some works have been proposed in relation to software product lines, to create such lines from a set of systems, e.g., [21], albeit manually. They also considered the recovery of *feature models*, e.g., [22]. However, to the best of our knowledge, all these works use manual approaches to discover features in some related systems [23]. (Semi-)Automated approaches all use some form of clustering techniques, e.g., [24, 25].

Yet, other works discussed or proposed feature discovery in the context of legacy systems and software modernisation, e.g., challenges of extracting business rules [26] or using slicing to identify components [27, 28]. They also propose to combine existing static and dynamic techniques with visualisation techniques [29]. Yet, our recent survey of service identification in legacy systems showed that such approaches are still in their infancy and that novel approaches are needed [30].

In a preliminary work [31], we tried *static slicing* to extract *self-contained code slices* within class hierarchies but obtained mixed results. Notwithstanding the inherent difficulties of inter-procedural flow analysis within the context of object-oriented languages with reflective capabilities such as Java, the quality of the slices depended almost entirely on the difficult, subjective selection of *program entry points*.

In another preliminary work [32], we studied the discovery of features that should have been implemented via multiple inheritance but could not due to the programming languages of the legacy systems, e.g., Java. We also considered features implemented via aggregation and state multiplication. We proposed tailored algorithms to discover associated features and applied them to three Java systems. We showed that it is possible to discover some features but that a more thorough study of their footprints and more systematic algorithms are necessary. We propose such a study and algorithms in the following.

2.2 Formal Concept Analysis

FCA is an automatic classification technique that has been used extensively for feature discovery, e.g., [10, 16, 33, 34, 35, 36, 37]. We briefly present it in the following and interested readers can refer to previous work [38, 39, 40] for in-depth explanations. FCA allows the construction of conceptual abstractions, or (formal) *concepts*, out of a collection of individual elements, described by their properties.

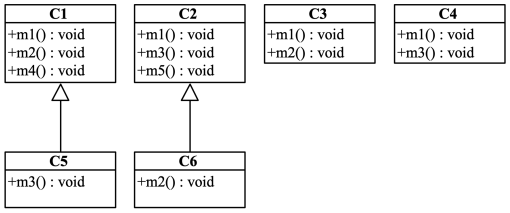
Concepts emerge from a (formal) context $\mathcal{K} = (E, P, I)$ where E is the entity set (formal objects), P the property set (formal attributes), and I (the incidence relation) associates E to P: $(e, p) \in I$ when entity e has property p. Figure 1 provides an example of a context, on the right, where entities are classes (listed vertically) and properties are methods (listed horizontally) from the class diagram shown on the left. Pairs that belong to the incidence relation are denoted by a $\times$ otherwise they are empty.

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 the example, if $X = \{c1, c4\}$ then $X^I = \{m1\}$, i.e., $m1$ is the only common method to both $c1$ and $c4$. However, for $Y = \{m1\}$, $Y^I = \{c1, c2, c3, c4\}$, i.e., $m1$ is shared by all of $c1$, $c2$, $c3$, and $c4$.

A concept is a pair (X, Y) where $X \subseteq E$, $Y \subseteq P$ are such that $X^I = Y$ and $Y^I = X$. Intuitively, (X, Y) is a concept if Y is the set of *all* properties that are common to the elements of X, *and* if there are no other objects outside of X that has *all* the properties in Y. In Figure 1, $(\{c1, c3\}, \{m1, m2\})$ is a concept, and so is $(\{c2, c4\}, \{m1, m3\})$ but not $(\{c1, c4\}, \{m1\})$. X and Y are called the *extent* and the *intent* of the concept, respectively, denoted, for a concept c by $ext(c)$ and $int(c)$.

The specialisation between concepts corresponds to extent inclusion or, conversely, intent containment. It is a partial order that, furthermore, represents a complete lattice, the *concept lattice* $\mathcal{L}$ of the context. Figure 2 (on the left) shows the concept lattice of the context from Figure 1. In that lattice, the concept $(\{c1, c3\}, \{m1, m2\})$ (Node ④) specialises the concept $(\{c1, c3, c6\}, \{m2\})$ (Node ①).

Given the choice of the *incidence relationship* in the example, which associates to each class its methods, the "clustering" provided by the concept lattice groups together the classes that define the same sets of methods. In the example in Figure 1, several methods are defined by multiple classes and, to the extent that their implementations are similar or identical, the resulting clustering may suggest a more efficient organisation.



Classes \ Methods	m1()	m2()	m3()	m4()	m5()
C1	×	×		×	
C2	×		×		×
C3	×	×			
C4	×		×		
C5			×		
C6		×			

Fig. 1. **Left:** Classes and their methods. **Right:** Context of classes × methods.

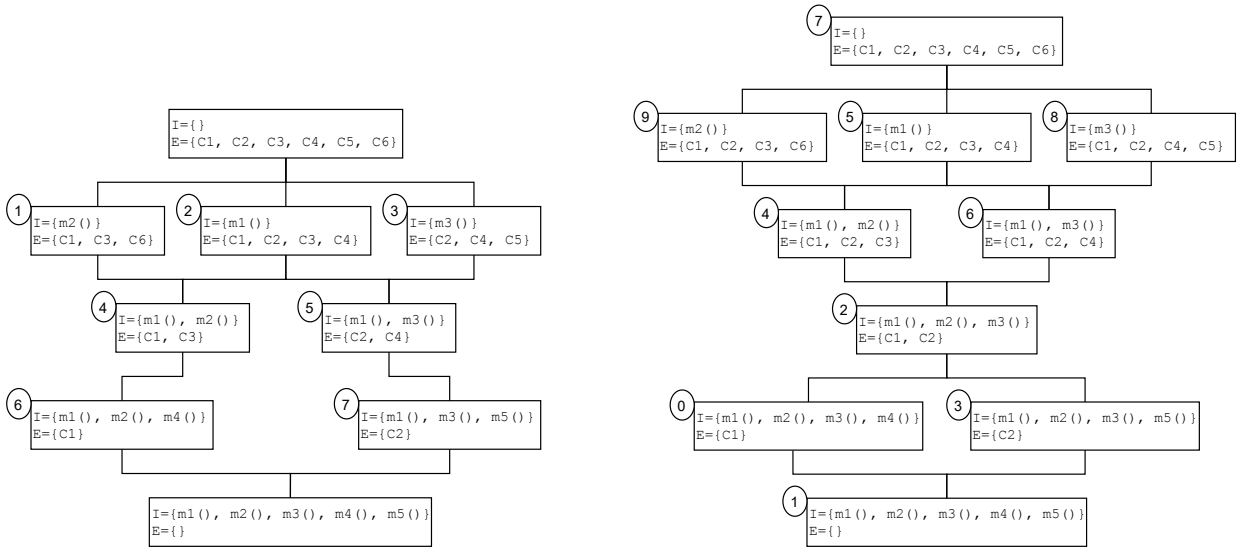


Fig. 2. The concept lattices of the contexts in Figure 1 (on the left) and Figure 3 (on the right).

For example, the method $m1$ is defined in $c1$, $c2$, $c3$, and $c4$. Node ② in Figure 2 captures this "sharing" and suggests creating a class with just $m1$, which would have two "subclasses", Node ④ and Node ⑤, inheriting methods $m2$ and $m3$, respectively, from Node ① and Node ③. Thus, the lattice in Figure 2 suggests reorganising the class hierarchy in Figure 1 into seven classes, represented by Node ① to Node ⑦, in which the five methods are declared only once.

This example illustrates the application of FCA to class hierarchy refactoring [39], [40]. FCA has been used extensively in software-engineering research (see e.g., [10, 16, 35]). In particular, FCA has been used to detect features within variants of a software product line, e.g., [36, 37].

2.3 FCA Encoding of Program Elements

The example of the previous section showed one potential "encoding" of a class hierarchy, i.e., associating each class with its methods. It illustrated the use of the resulting lattice for maximal refactoring. Another FCA-based method could exploit a different relationship within the class hierarchy to draw different inferences useful to another application.

In our study, we use FCA to detect occurrences of the *ad-hoc idiom* to further discover functional features from hierarchies, where several of those are interwoven. This idiom is a symptom of developers failing to recognise a functional feature, defined as a cohesive set of code elements.

In the following, we choose to consider public methods and attributes as the elements of interest, which we encode using parts of their declarations. For a method, we use its name, and a list of the types of its arguments in order of appearance (together called its *signature*). For an attribute, we use its type and name.

We define *togetherness* to mean "occurring within the same class sub-hierarchy" and *multilocation* as "occurring in two sub-hierarchies that are not hierarchically related", i.e., such that one sub-hierarchy is not included in the other. We provide justifications for these definitions of *togetherness* and *multilocation* in Section 6.

For now, we show the context and the lattice corresponding to these definitions of togetherness and multilocation and of a relationship that we call *reverse inheritance*. Figures 3 and 2 show the reverse-inheritance context and lattice, respectively, for the class hierarchy in Figure 1. Rows denote class sub-hierarchies with root c as opposed to class c. Consequently, rows for $c1$ and $c2$ now include $m3$, from $c5$, and $m2$, from $c6$, respectively. Figure 2 puts side-by-side the lattices obtained from the relation defined in the previous Section 2.2 and from the relation of reverse-inheritance.

In the reverse-inheritance lattice, in Figure 2 on the right, we identify $\{m1, m2, m3\}$, $\{m1, m2\}$, $\{m1\}$, $\{m1, m3\}$, $\{m3\}$ and $\{m2\}$ as candidate functional features because the extents of the corresponding concepts, Node ②, Node ④, Node ⑤, Node ⑥, Node ⑧ and Node ⑨, each have more than two hierarchically-unrelated classes, as further explained in Section 6.1. We also discuss in Sections 6 and 9 whether a method, declared in many classes, is an *interesting* functional feature.

Classes \ Methods	m1()	m2()	m3()	m4()	m5()
C1	×	×	×	×	
C2	×	×	×		×
C3	×	×			
C4	×		×		
C5			×		
C6		×			

Fig. 3. Reverse-inheritance context, whose lattice is shown in Figure 2 (on the right).

2.4 Subject Systems

In the rest of this article, we use the following systems to evaluate and validate our algorithms. We choose these systems because they are old and with diverse histories, thus mimicking legacy systems. They have been used in previous works, e.g., by Meng et al. [41], Wen et al. [42], or Molnar and Motogna [43], and included in datasets, e.g., the Qualitas Corpus [44]. They have different maturity levels and design quality[1], going from version 0.7.1 for FreeMind to version 5.2 for JHotDraw. They cover a wide range of domains: a graphical editing framework (JHotDraw), a Java reverse-engineering tool (JReversePro), a Web-mail client (JavaWebMail), a free-text search (Lucene), and a mind-mapping software (FreeMind).

JHotDraw is a graphical user interface framework that was developed by Erich Gamma and Thomas Eggenschwiler based on the Smalltalk original developed by Brant [46]. From its inception in Smalltalk, 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 in *good* object-oriented design. Consecutive versions of JHotDraw added functionalities to the framework, included new applications based on the framework, and also involved regular refactorings. We use version 5.2, which contains about 160 compilation units (Java files) and 170 types (user-defined classes and interfaces).

JReversePro is a Java program for reverse engineering compiled Java code. It takes as input a Java classfile and produces one of three outputs, depending on the call parameters: (1) the class constants pool, (2) the class disassembly, and (3) the class decompilation. JReversePro is relatively small with 85 classes and interfaces, and about 12,000 lines of code. It does not use outside libraries, except for the standard Java library. Unlike JHotDraw, which has seen the contribution of many designers, JReversePro *appears* to be essentially the work of its creator.

JavaWebMail is a servelet-based Java Web e-mail client that can connect to IMAP or POP e-mail boxes. The version that we use, 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 system was later rewritten *completely* in 2014 (version 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).

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 AOL, Apple, CodeCrawler, Comcast, Eclipse, IBM, and JIRA. The latest release, 8.7.0, was released in November 2020. The version that we use in our experiments is version 1.4 from 2004.

FreeMind is a *mind mapping* tool. A *mind map* is a graph whose nodes represent concepts or ideas, and whose links represent associative relationships between those concepts/ideas; a sort of an informal *semantic network*. Mind maps, and FreeMind, evolved over the years to include search functionality, links to other sources, etc. We use 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 other contributors.

Table 1 provides some quantitative data about these systems. Column "Comp. Units" is the number of compilation units, i.e., Java source files. Column "Types" contains the numbers of distinct classes and interfaces. The authors of FreeMind used extensively member/local classes, hence, the number of types greatly exceeds that of compilation units.

TABLE 1
Some metrics for the chosen OO systems.

Systems	#LOCs	#Comp. Units	#Types	#Methods
FreeMind 0.7.1	65,490	92	198	4,785
JavaWebMail 0.7	10,707	111	115	1,079
JHotDraw 5.2	9,419	160	171	1,229
JReversePro 1.4.1	9,656	83	87	663
Lucene 1.4	15,480	160	197	1,270

3 Problem Formulation and Examples

In the absence of programming language-level constructs to represent functional features in program units that can be developed, maintained, and composed at will (aspect-oriented programming languages propose such constructs, e.g., static or dynamic aspects [2]), developers will resort to traditional OO design idioms to package and compose functional features. Thus, we want to discover such functional features in the source code of object-oriented software systems, using static analyses and FCA, independently of whether developers (1) knew about these functional features and (2) implemented them separately from one another.

Based on our review of the literature and experience with the development and analyses of software systems, we identified three scenarios that developers follow when implementing functional features:

- Scenario 1: *multiple inheritance*, whereby each functional feature is represented using its own class hierarchy, and a class combining several features inherits from the corresponding class hierarchies.
- Scenario 2: *delegation*, in which a separate class hierarchy represents each functional feature, which are combined by aggregation and called by delegation.
- Scenario 3: *ad-hoc implementation*, when developers have missed useful/reusable functionalities, for example when the same functionality was, inadvertently, coded several times in a system.

In Scenarios 1 and 2, developers took care to (1) *develop and package* functional features as classes or interfaces, with

1. Although version numbers are somewhat arbitrary, others noted that increasing numbers should reflect increasing maturity, e.g., [45].

their own class or type hierarchies and (2) *compose* them with other features, where needed, using either multiple inheritance or aggregation. Recognising such classes and interfaces in legacy code is useful, to identify reusable pieces of code and report existing functional features. However, most OO programming languages–including Java–do not support multiple inheritance, or delegation in the true sense (only the Self[2] language supports true delegation). Consequently, we resorted to *emulations* of multiple inheritance and delegation with the constructs of Java language (see Sections 3.1 and 3.2).

In Scenario 3, developers, possibly novices/possibly due to lack of time, failed to (1) recognise a set of related program elements as being a distinct functional feature and (2) package them so that they can be composed. This scenario is similar to code cloning, in which a subset of program elements is copied and used in a different context [47, 48]. Yet, it differs from code cloning in granularity: instead of cloning statements across methods or methods across classes, it pertains to "cloning" sets of related program elements across classes/packages. This scenario describes the *ad-hoc implementation* of functional features. It is recognised by the multiple occurrences of a set of program elements scattered in different parts of a legacy system.

When analysing legacy systems, we do not know beforehand which of the above scenarios were used by developers to embody functional features. However, both the *deliberate emulations* (multiple inheritance and delegation) and the *non-deliberate* implementation (ad-hoc) of functional features have a recognisable footprint at the interface level of a class, i.e., visible without parsing method bodies. We hypothesised that the occurrence of such footprints in legacy code may be indicative of the presence of reusable functional features (candidate features). Consequently, both the presence of distinct functional features and the techniques (emulations) used to compose them are hypotheses to be tested by applying our algorithms.

Different functional features within the same legacy system may have different maturity levels: the mature ones may have been implemented in separate abstractions (classes and interfaces) whereas the less mature/emerging ones appear as multiple occurrences of the same sets of program elements. Thus, functional features are worth identifying, both for; a) deliberate ones to gain a better understanding of the structure of the code, and b) ad-hoc ones to suggest refactorings that will enhance the program maintainability.

To validate our hypotheses, we:

- Develop algorithms that help recognise the footprints in legacy software.
- Evaluate the candidate features identified by the algorithms for their relevance using three methods:
  1) Our own evaluation, by studying thoroughly a sample of the tested software systems (JHotdraw, JReversePro).
  2) A user evaluation, by conducting a controlled experiment with independent users.
  3) A comparison of our candidate features with those of a comparable technique.

2. https://selflanguage.org/

We now illustrate the three scenarios with simple examples. In these examples, we use three relationships: class inheritance (white triangle, solid line), type implementation (white triangle, dashed line), and aggregation (white diamond, solid line). Among the possible *use* relationships [49] of association, aggregation, and composition, we use aggregation relationships to illustrate that developers may want to express *conceptually* a composition but that they can only implement *practically* an aggregation.

3.1 Scenario 1: Multiple Inheritance

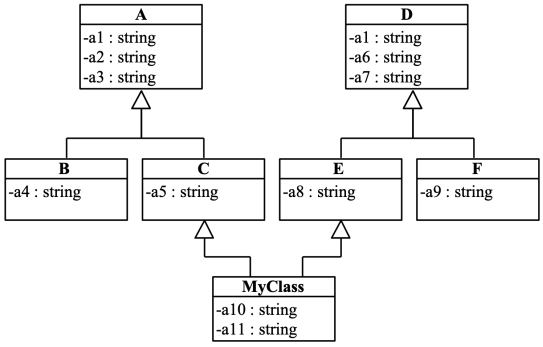


Fig. 4. Implementing functional features using multiple inheritance.

An object-oriented technique for composing reusable features is multiple inheritance. If a developer wants to develop a class `MyClass` that combines, uses, and–or offers two *independent* functional features, implemented by two classes `C` and `E`, then she makes `MyClass` an *extension* of classes `C` and `E`, as illustrated in Figure 4.

However, for many reasons outside of the scope of this article, e.g., [50, 51, 52], few OO languages support *implementation/class-level* multiple inheritance, e.g., C++, as opposed to *type-level* multiple inheritance, e.g., Java. Many OO languages choose to offer single-class inheritance and interface multiple inheritance, like Java with its *interfaces* to specify that a class implements several, hierarchically-independent types (which is the language of the subject systems presented in Section 2.4).

Developers nonetheless *think* in terms of multiple inheritance because it is often natural with respect to the application domain, for example, a `PartTimeStudent` *conceptually* is a `Student` and a `Worker`. However, developers must implement the domain concepts using single-class inheritance and multiple interface inheritance, which are the scenarios that we want to identify.

Regarding feature discovery, this scenario is less interesting because, in Java, it corresponds to the use of interfaces to describe and compose features. We only succinctly discuss this scenario in the following.

3.2 Scenario 2: Delegation

This scenario corresponds to a situation where a class A, which must implement two functional features embodied in classes B and C, references an instance of each class, to which it delegates the corresponding behaviour. Delegation in OO programming languages has a *precise meaning*: the delegate/component executes its methods in the context of

the delegator/aggregate. However, some programming languages, like Java, have limited support for *true* delegation, which may lead to the fragile-base class problem [53]. In the following, we use *delegation* to mean a combination of aggregation and method forwarding.

Figure 5 shows two idioms for composing functional features with delegation. We implement the behaviour of a part-time, immigrant, student as an aggregation of the behaviours of *student*, *immigrant*, and *worker*. In Figure 5a, the class `ImmigrantPartTimeStudent` delegates to both `WorkerImp` and `ImmigrantImp` through its attributes `worker` of type `Worker` and `immigrant` of type `Immigrant` and its implementations of the methods of interfaces `Worker` and `Immigrant`.

Figure 5b shows an "undisciplined" or partial use of delegation. Class `ImmigrantPartTimeStudent` implements the behaviour of classes `Worker` or `Immigrant` *implicitly*, because it does not implement interfaces `Immigrant` or `Worker`, and partially, because the method `isAuthorizedToStudy()` from `Immigrant` is not implemented by `ImmigrantPartTimeStudent`.

The idiom in Figure 5a is identified by our algorithms for detecting instances of multiple inheritance in the previous Section 3.1. Also, other idioms exist for delegation. We present our algorithm for the discovery of functional features implemented using delegation in Section 5.

3.3 Scenario 3: Ad-Hoc Scenario

This scenario happens when a developer did not recognise that a set of program elements forms a cohesive whole that implements a distinct functional feature and, thus, did not package them in a construct offered by the OO programming language—interfaces and classes. The program elements, methods and attributes that implement the feature, are defined and duplicated in all the classes that support that functional feature.

We recognise, as explained in further detail in Section 6, that sets of program elements scattered throughout class hierarchies contribute to the same functional feature using togetherness and multilocation: (1) if some program elements *together* provide a functional feature, then they are likely to be defined in the same class and–or its siblings and (2) if they represent a useful and reusable/reused functional feature, they are likely to occur in *multiple locations* in the class hierarchies of a system.

Figure 6a shows the example of a hierarchy of resources. This hierarchy divides into two sub-hierarchies containing, respectively, classes `Machinery` and `ShopFloorStaff`, which both offer some capabilities and a schedule as well as resources pertaining to machines, stocks, and drivers, be them machines, items, or people. We want to recognise such a scenario and discover the functional feature "manufacturing resource" that has some capabilities and a schedule.

The example in Figure 6a, albeit simple, is realistic because developers could implement such hierarchy *without* seeing the commonalities between the two sub-hierarchies for many reasons: (1) a large/poorly documented class hierarchy difficult to grasp by any developer, (2) different developers working on the different sub-hierarchies, possibly at different times, (3) urgent maintenance requests precluding refactoring, or (4) novice developers with limited understanding of the whole hierarchy.

However, it is indeed unlikely to find methods and attributes with this level of regularity in several sub-hierarchies of legacy systems, i.e., two *homomorphic hierarchical slices*. We show in Section 6 more realistic examples and algorithms to discover less regular functional features, using the FCA encoding introduced in Section 2.3.

4 Scenario 1: Multiple Inheritance

The identification of candidate features using idioms related to multiple inheritance was the focus of one of our previous works [32]. Consequently, we only summarise this previous work here for the sake of completeness and refer the interested reader to the original work for all the details.

4.1 Steps 1 and 2: Idioms Definitions

Figure 7 shows the different idioms that developers can use to implement multiple features. In the first idiom, in Figure 7a, class `PartTimeStudent` implements two interfaces but reuses no implementation. In Figure 7b, `PartTimeStudent` inherits from class `Student`, which provides a functional feature. It implements an interface representing a second functional feature, which it must implement. In the third idiom, in Figure 7c, `PartTimeStudent` combines the functional features of `Student` and `Worker` through a combination of inheritance and aggregation.

These idioms are not exhaustive. A class could implement three or more features and we could have variations on the presence and location of the interfaces in the inheritance trees. For example, it is obvious in Figure 7a that class `PartTimeStudent` implements two functional features through the use of two interfaces. Developers could have defined an interface `StudentWorker`, inheriting from the two interfaces and implemented by `PartTimeStudent`, giving the impression that it implements only one feature.

4.2 Step 3: Idioms Detection

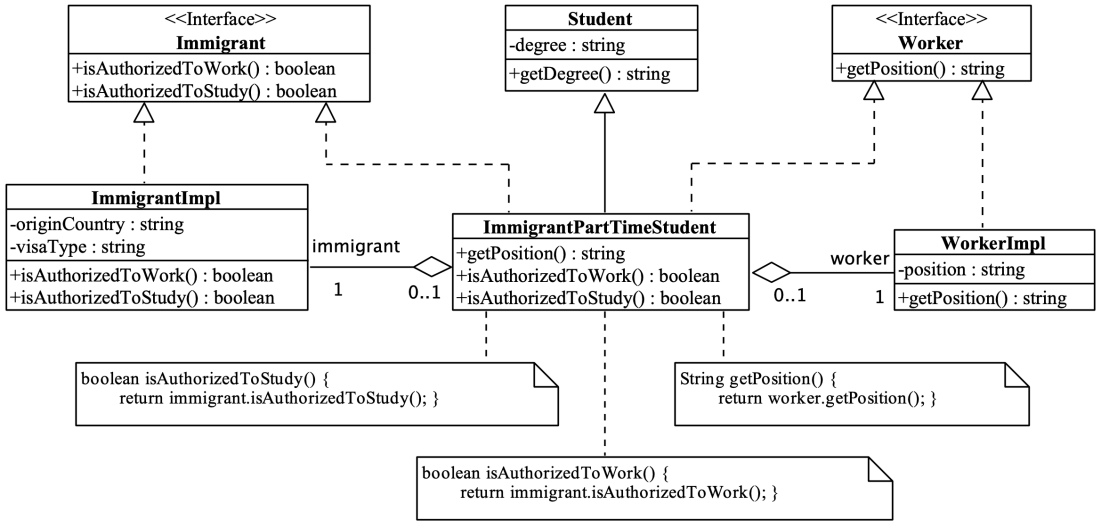
Algorithm 1 [32] identifies classes that potentially implement multiple features by implementing multiple interfaces.

It returns the set of classes that implement more than one interface that is not in the Java class libraries, not a marker interface, e.g., `org.springframework.stereotype.Controller`, and not an interface defining only constants. Such classes are implementing multiple *functional* interfaces that may indicate developers' intent to use multiple inheritance.

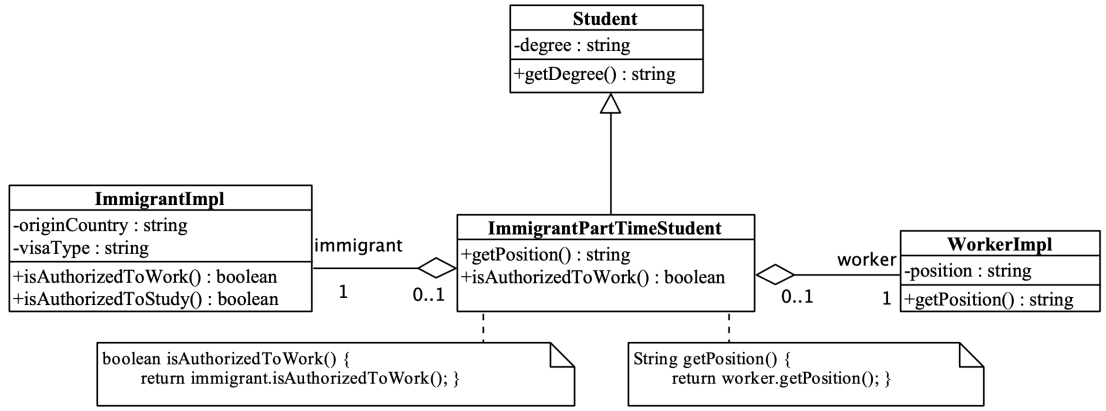
4.3 Steps 4 and 5: Detection Results

There are occurrences of the idioms in which the interfaces implemented by a class do not represent *functional features*: they represent infrastructural features, e.g., service contracts with infrastructure service like JEE, or Spring; implementation contracts, e.g., the method `compareTo` of interface `Comparable`; constant pools; or *marker interfaces* without methods but defining some capabilities, e.g., `Serializable`.

Also, the definitions of functional features depend on the domain. For example, the GUI functionality of *JReversePro*



(a) A disciplined case of delegation. One class inherits a functional feature from another, and delegates two additional functional features to two other classes.



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

Fig. 5. Implementing functional features with delegation.

(see Section 2.4) is secondary to its *main feature*, which is the analysis of Java byte code. Thus, observing that one of its classes implements the interface `EventListener` is less interesting than if this interface appears in a class of JHotDraw, which *is* a graphical system.

In JHotDraw, we observed several classes extending a class and implementing one or many interfaces. For example, `CompositeFigure` extends `AbstractFigure` and implements `FigureChangeListener`. Such a combination of class inheritance and (multiple) interface implementation is common in graphical frameworks in single-inheritance programming languages, like Java. `FigureChangeListener` is a contract that graphical figures must fulfill to react appropriately to events.

In JavaWebMail and JReversePro, no case of multiple inheritances corresponds to interesting features. Existing cases are either utility interfaces (e.g., `Iterator`), marker interfaces (e.g., `Serializable`), or constant interfaces (e.g., `JwmaInboxInfo`).

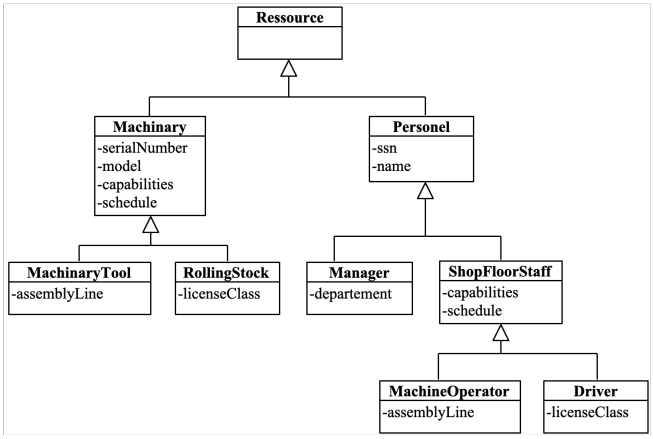
We do not further discuss the idioms related to multiple inheritance and their detection and focus on the more interesting scenarios of using delegation and ad-hoc idioms to represent and combine features.

5 Scenario 2: Delegation

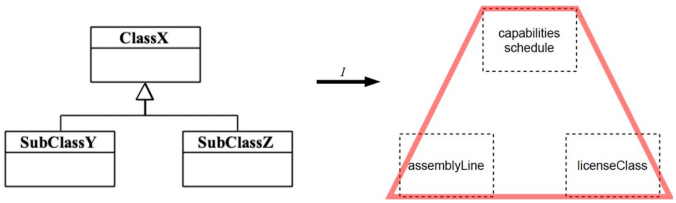
As summarised in the introduction, our approach for discovering functional features in OO legacy systems includes five steps. We now describe, for functional features implemented and composed using delegation, the idioms (Steps 1 and 2) in Section 5.1, algorithms (Step 3) in Section 5.2, and results (Steps 4 and 5) in Section 5.3.

5.1 Steps 1 and 2: Idioms Definitions

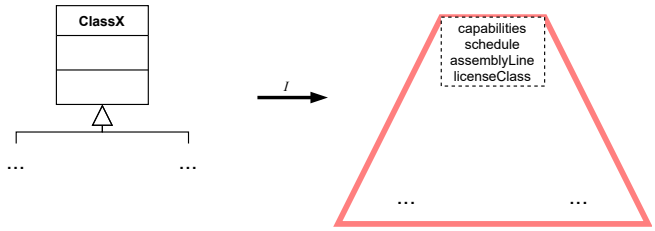
Figure 5 showed two idioms for composing functional features with delegation: (1) when the delegator `PartTimeImmigrantStudent` *explicitly* implements the interface of the delegate, in Figure 5a, and (2) when it does not, in Figure 5b. The first is an aggregation-based "emulation" of multiple inheritance, which was illustrated in Figure 7c. Advantages of this idiom include *assignment compatibility*, i.e., we can use a `PartTimeImmigrantStudent` object where an `Immigrant` or `Worker` is expected.



(a) Ad-hoc implementation of a functional feature.



(b) Incidence relation based on Figure 6a: class attributes patterns.



(c) New incidence relation based on Figures 6a and 6b: considering the sub-hierarchy with `ClassX` as root to include all attributes in and "below" `ClassX`.

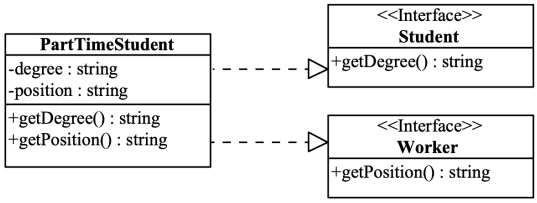
Fig. 6. Example of ad-hoc implementation and incidence relations.

The second idiom, in Figure 5b, does not provide assignment compatibility but better control of the delegation. It allows choosing which behaviour of the delegate is offered by the delegator. It also allows adapting the delegate's behaviour to the delegator context by changing its signatures, e.g., method name or parameter list, for example by using aggregate-specific default values for some.

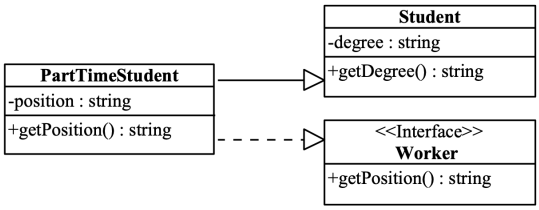
5.2 Step 3: Idioms Detection

Notwithstanding changes to method signatures, to detect instances of delegation in OO code, an algorithm must contend with scenarios in which the delegator implements only parts of the public interface of the delegate. In Figure 5b, `ImmigrantPartTimeStudent` implements *only* one method of `Immigrant`, i.e., `isAuthorizedToWork()`.

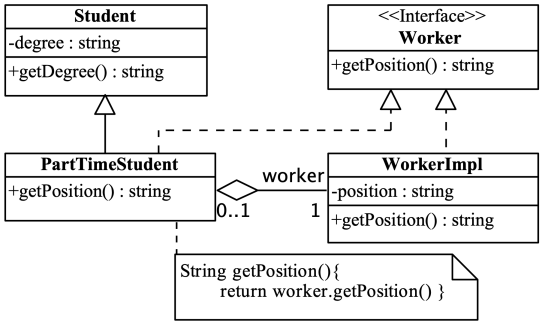
We could use a percentage of the implemented public interface to discover features based on delegation. When a class A (1) has a data member b of type B and (2) supports more than $\alpha\%$ of B interface through delegation to b, then we could qualify these methods and attribute a feature. However, such a heuristic would likely produce many false-positive and false-negative features: domain classes typically have a few domain methods and many utility methods (constructors, accessors, comparators, etc.). Thus, a percentage alone is not indicative: the feature depends on which methods are being delegated. Thus, we introduce the *domain interface* of a class C, noted as $DOMINT(C)$, as the full interface of C, from which we remove constructors,



(a) One class implementing two interfaces.



(b) One class inheriting from another, implementing an interface.



(c) One class inheriting from another, aggregating yet another, and implementing the latter's interface.

Fig. 7. Implementing functional features with multiple inheritance.

foreach class C in S **do**
  let $SUPER(C) = \{ superclass(C) \} \cup \{ implemented\ interfaces(C) \}$
  foreach element E in $SUPER(C)$ **do**
    if E in Java API **then**
      $SUPER(C) = SUPER(C) \setminus E$
    end
    if E in Marker Interface **then**
      $SUPER(C) = SUPER(C) \setminus E$
    end
    if E in Constants Interface **then**
      $SUPER(C) = SUPER(C) \setminus E$
    end
  end
  if $|SUPER(C)| \geq 2$ **then**
    mark C as potentially implementing the feature in $SUPER(C)$
  end
end

Algorithm 1: Feature discovery with multiple inheritance.

accessors, and methods inherited from the Java API (e.g., `clone()`, `hash()`, `toString()`, etc.).

The feature also depends on what methods are being delegated. In Figure 5, the delegator and the delegates are domain classes[3] while a domain class could also delegate to a utility class. For example, if a `Student` class has a `set` attribute to contain the courses taken by the student, then `set` and any of its methods delegated by `Student` is not a functional feature of `Student`.

Our algorithm for discovering functional features based on delegation considers the methods implemented by classes and compares them to the methods implemented by their attributes, disregarding the interfaces that they implement. In Figure 5, we look for scenarios of ad-hoc delegation (Figure 5b), which are "weaker" than disciplined delegation (Figure 5a) to identify both scenarios using Algorithm 2.

foreach class C in S **do**
  let *DOMINT*(C) = { $signature(m) \mid m$ is a domain method of C }
  foreach attribute a in C with $type(a)$ **not** in the Java API **do**
    let *DOMINT*$(type(a))$ = { $signature(n) \mid n$ is a domain method of $type(a)$ }
    if *DOMINT*$(C) \cap$ *DOMINT*$(type(a)) \neq \emptyset$ **then**
      mark C as potentially implementing the feature *DOMINT*$(type(a))$
    end
  end
end

Algorithm 2: Feature discovery with delegation. (*signature(·)* returns a method signature, *type(·)* returns the type of an attribute (i.e., fully qualified name in Java)).

Thus, we considered a pair $\langle C, a \rangle$ as a potential functional feature by delegation if the *domain interfaces* of class C and attribute a have one or more methods in common. We also compute the *coverage ratio* for a $\langle C, a \rangle$ as:

$$coverage = \frac{size(DOMINT(a) \cap DOMINT(C))}{size(DOMINT(a))} \quad (1)$$

with which ratio of 1 means that all the domain methods of the attribute are implemented by the delegator. A ratio of 1 is too stringent a condition because a class may not need *all* the methods of an attribute but it is a useful benchmark.

5.3 Steps 4 and 5: Detection Results

We apply our algorithms to the five subject systems described in Section 2.4. Table 2 summarises the results for the five systems. We manually analyse the discovered candidate features: all the candidates for JHotDraw, regardless of coverage, and *only* those with coverage greater or equal to 50% for the other systems. The full manual analysis is available online[4].

3. We could rather write, *belong to the same domain*: if the domain is GUI, then delegates that are part of Java graphical frameworks, e.g., `java.awt.*` or `javax.swing.*`, *do* represent interesting delegations.
4. https://github.com/hafedhmili/featurediscovery
https://www.ptidej.net/downloads/replications/tse23a/

We observe that our algorithm lacks precision for coverage greater than 50% for four reasons:

- There are only a few candidate functional features discovered by our algorithm per system. In particular, in JReversePro and FreeMind, only 10 candidate features are found and, because all of them are false positive, the reported precision values are zero. In other systems with higher absolute numbers of candidate features, precision values are higher.
- We considered an implementation of delegation in which an attribute of the delegate exists in the delegator. We observe candidate features in which this attribute exists merely to simplify the code, not as a true delegate, e.g., a class `Person` that has an attribute `Address address`: an `address` is not a delegate for `Person`.
- We do not handle implementations in which *collections* of attributes are used to represent features. Generic collections could help identify delegates but genericity is a recent addition to Java and many developers/legacy systems use un-typed collections.
- We excluded accessors from *DOMINT*(C) but observed features in which accessors contribute to some domain-related computation and should have been kept in *DOMINT*(C).

We also observe differences among the five systems. In JHotDraw, JavaWebMail and, to some extent, Lucene, the functional features are central to their domains. In FreeMind and JReversePro, these features relate to GUI aspects, which are not central to the system functionalities.

We finally observe that delegation is used in many design patterns to implement features, including Chain of Responsibility, Decorator, Observer, Proxy, and Strategy.

We conclude that:

- The algorithm of Section 5.2 can discover reusable functional features created via delegation.
- The output of the algorithm provides valuable insights into the design of the systems, whether the candidate features are true features or not.
- Short of analysing method bodies, which requires access to the source code, simple heuristics could significantly improve its precision.

However, a functional feature may *not* be represented *intensionally*, i.e., as a class or an interface, as in Figures 4, 7, or 5, but *extensionally*, i.e., repeated/duplicated in many classes. *Discovering* that some program elements embody a functional feature requires that these elements occur *together* (togetherness) in *many locations* (multilocation).

6 Scenario 3: Ad-Hoc Implementation

We show, for ad-hoc implementations of functional features, the idioms (Steps 1 and 2) in Section 6.1, algorithms (Steps 3) in Section 6.2, and results (Steps 4 and 5) in Section 6.3.

6.1 Steps 1 and 2: Idiom Definition

Figure 6 shows an example of the ad-hoc idiom and the incidence relations that we use to discover related candidate functional features.

TABLE 2
Numbers and precision of the candidate features composed through delegation.

Metric \ Systems	JHotDraw v5.2		JavaWebMail v0.7		JReversePro v1.4		FreeMind v1.4.1		Lucene v0.7.1	
#Cand. with Coverage $= 100\%$	8	38%	2	50%	2	0%	3	0%	32	47%
#Cand. with $50\% \leq$ Coverage $< 100\%$	7	43%	1	100%	0	N/A	0	N/A	3	33%
#Cand. with Coverage $< 50\%$	26	N/A	13	N/A	2	N/A	20	N/A	43	N/A
Total Cand. with Coverage $\geq 50\%$	15	40%	3	66%	2	0%	3	0%	35	46%

Figure 6a shows an example of program elements occurring together in multiple locations. The *emerging* functional feature (*intension*) is to describe *production resources*: a production resource has `capabilities`, i.e., what it can do, and a `schedule`, when it should do it. It divides into assembly-line resources (both machine tools and people) and transportation resources (rolling stocks and drivers).

Figure 6b shows an example of the idiom that we want to discover: (1) a hierarchy fragment, i.e., a class X and its two subclasses Y and Z and (2) a *slice* of the hierarchy fragment, i.e., a subset of its program elements[5]. A class/interface hierarchy is a directed graph G whose vertices V are classes/interfaces and edges $E = V \times V$ represent inheritance/realisation relations between vertices. Let *PE(v)* be the non-empty set of program elements of a vertex v and *P(v)* the path starting from v, i.e., the set of vertices reachable from v via its edges, transitively. A slice of the program elements rooted in $v \in V$ is $S(v) = \{PE(v')|v' \in P(v)\}$.

To find occurrences of this ad-hoc idiom, we use FCA, as explained in Section 2.2, by defining a *context* in which entities are hierarchy fragments $P(v), v \in V$ and properties are program elements, $PE(v)$. This context is computationally costly. First, given a non-leaf class `ClassX`, the number of hierarchy fragments under `ClassX` is exponential in the number of classes under `ClassX`. Second, for a given hierarchy fragment $P(v), v \in V$, the number of possible slices S(v) is exponential in the number of program elements in each class. Indeed, if `l`, `m`, and `n` are the numbers of program elements in `ClassX`, `SubClassY`, and `SubClassZ`, respectively, then there are $2^l - 1$ non-empty subsets for `ClassX`. Thus, there are $(2^l - 1) \times (2^m - 1) \times (2^n - 1)$ possible sets. Finally, the matching of slices with one another is a graph-matching problem and, thus, NP-complete.

However, this encoding is also needlessly restrictive: it requires the exact same slices to occur at different locations. We relax the *design* of *both* the hierarchy under a class `ClassX` in Figure 6b and the program elements in the slices. Figure 6c shows the *reverse inheritance* incidence relationship described in Section 2.3. It associates a root class `ClassX` with the *union of the program elements of its sub-classes*.

With this new incidence relationship, concepts consist of pairs (V, PE) in which, for each class $v \in V$, the sub-hierarchy rooted at v has all of the program elements in $PE(v)$, plus those in $PE(v')$ for any v' subclass of v, and both V and PE are maximal in the sense that: (1) there is no class outside of V that has all of the elements in PE and there is no element outside of PE that is common to all classes in V.

5. The figure shows *attributes*, but the algorithm also uses *methods* by considering their signatures.

At first, we could state that any concept (V, PE), such that V contains more than one class, suggests that PE represents a feature. However, considering the example in Figure 6a, this statement would yield a concept whose extent consists of the classes {`Resource`, `Machinery`, `Personnel`, `ShopFloorStaff`} and whose intent consists of the program elements {`assemblyLine`, `capabilities`, `licenseClass`, `schedule`}. Yet, the intent does *not* occur in four *independent* hierarchy fragments, because three of those occurrences correspond to *nested* sub-hierarchies, which are shown in Figure 8. We conclude that not every concept with an extension containing more than one class represents a functional feature: To be valid, the classes that are in the extent must *not* be hierarchically related.

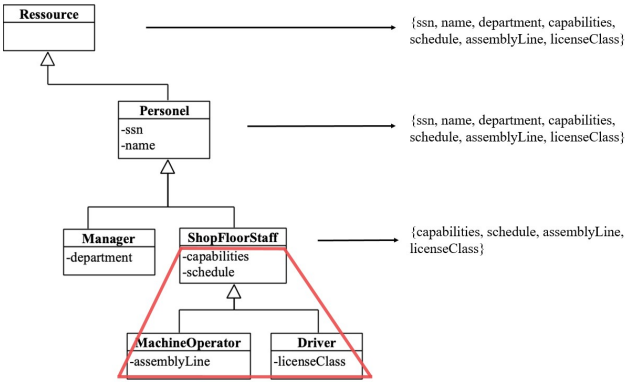


Fig. 8. The three occurrences of {`capabilities`, `schedule`, `assemblyLine`, `licenseClass`} are not independent.

Thus, before deeming the intent PE of a concept (V, PE) a potential functional feature, we order the classes in V using the class–subclass relationship and keep only the set of minimal classes , e.g., in Figure 6, the original extent {`Resource`, `Machinery`, `Personnel`, `ShopFloorStaff`} is reduced to {`Machinery`, `ShopFloorStaff`}.

Conversely, we consider only the intent PE of a concept (V, PE) that has the largest number of program elements w.r.t. the classes in V and their subclasses. Indeed, if a set of program elements has two or more *independent* occurrences, which makes it a candidate feature, then *a-fortiori* any subset thereof has *at least* as many *independent* occurrences. Consequently, some of the subsets *could* qualify as features of their own: The criterion here is whether there are further independent occurrences, i.e., ones that are not subsets of the respective occurrences of the super-feature.

Figure 9 illustrates a candidate feature and its sub-features: the set of elements {`assemblyLine`, `capabilities`, `licenseClass`, `schedule`} is deemed a functional feature, because it has two independent

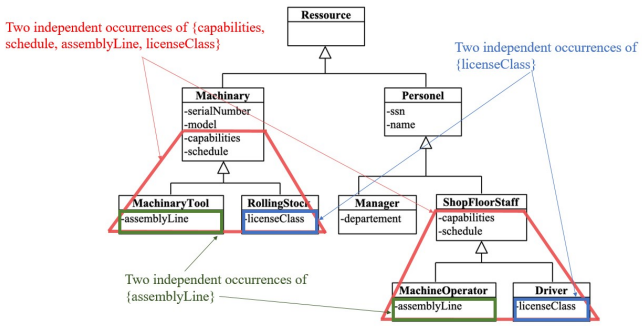


Fig. 9. A candidate feature induces several candidate sub-features.

occurrences under `Machinery` and `ShopFloorStaff`, and so do the subsets {`assemblyLine`} and {`licenseClass`}, which have two *independent* occurrences each. These two subsets {`assemblyLine`} and {`licenseClass`} are actually sub-features of the more interesting feature {`assemblyLine`, `capabilities`, `licenseClass`, `schedule`}. To avoid redundancies, we only keep the features of maximal size: If a candidate feature has a super-feature with the same number of occurrences, it is ignored.

6.2 Step 3: Idiom Detection

We identify program elements appearing together at multiple locations, such as the ones illustrated in Figure 6a, to discover functional features, by clustering these elements. We use FCA because it supports the construction of *concepts* from collections of classes and their program elements.

Following our encoding of program elements in concept lattices explained in Section 2.3, we have a context $\mathcal{K} = (C, M, I)$, where C is the set of classes, M is the set of their methods and attributes, and I the incidence relationship *reverse inheritance*. A context $\mathcal{K}$ associates to each sub-hierarchy rooted at c all the elements (methods and attributes) that occur anywhere in the sub-hierarchy.

We extract candidate functional features in three steps: (1) we build the concept lattice $\mathcal{L}$ of $\mathcal{K}$ using an incremental lattice-construction algorithm [54], (2) we traverse the resulting lattice to identify candidate features, and (3) we categorise candidate features by computing two measures for each one of them.

6.2.1 Generating Candidate Features

Algorithm 3 shows how we obtain candidate features. It takes a concept lattice $\mathcal{L}$ as input and outputs a list of candidate functional features, *FeatureList*. For each concept (X, Y), starting from the level below the lattice top ($\top_{\mathcal{L}}$), the algorithm computes $\min(X)$, which removes non-minimal classes w.r.t. the class–subclass relationship. This removal leaves only the root classes of the *independent occurrences* of Y. If there is more than one such occurrence, $|\min(X)| > 1$, then it adds the concept to the list of candidate features. It then looks at each child (X', Y') of (X, Y) in $\mathcal{L}$ and adds it to the list of concepts. If its minimal extent, $\min(X')$, has the same size as the parent's, $\min(X)$, then it removes (X, Y) from the candidate feature list.

Input: concept lattice $\mathcal{L}$
Output: feature candidates *FeatureList*
ListConcept $\leftarrow$ children($\top_{\mathcal{L}}$)
FeatureList $\leftarrow \emptyset$
while *ListConcept* $\neq \emptyset$ **do**
  $(X, Y) \leftarrow$ extract($ListConcept$)
  if $|\mathbf{min}(X)| > 1$ **then**
    add $((X, Y)$, *FeatureList*)
    foreach $(X', Y') \in$ *children*$((X, Y))$ **do**
      add $((X', Y')$, *ListConcept*)
      if $(|\mathbf{min}(X')| = |\mathbf{min}(X)|)$ **then**
        remove $((X, Y)$, *FeatureList*)
    end
  end
end

Algorithm 3: Feature discovery with lattice mining.

We use *FeatureList* as input for categorisation and measurement, described in the following subsections.

6.2.2 Categorising Candidate Features

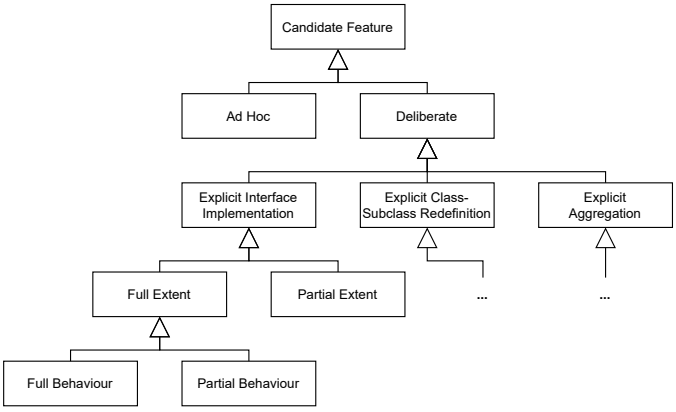


Fig. 10. Categories of candidate functional features.

Once *FeatureList* is output by Algorithm 3, we analyse it manually and categorise the candidate features based on the identified behaviour (the *intent* component). Figure 10 summarises the categories, which we explain in the following.

First, we divide candidate features in two categories:

1) *ad-hoc features* correspond to cases where developers *missed* that a set of methods (the intent) represented a cohesive, reusable functionality worth embodying in a class or interface and, thus, (partly) cloned these methods in different classes. *No relationship exists between the elements (classes or interfaces) of the extent.*
2) *deliberate features* correspond to cases where developers *recognised* that a set of methods represented a reusable functionality, and codified it into a class or interface; that class/interface is then somehow 'reused' in several places, e.g., using some of the design/coding patterns uncovered by Algorithm 2. Each candidate feature contains one anchor type, which is the class/interface that embodies the common behaviour.

If a candidate functional feature belongs to the *Ad Hoc* category, all of its descendants in the lattice $\mathcal{L}$ will *also* belong to this category. Indeed, because the extents of the descendants are *subsets* of the extent of their parent, then if no relationship existed between the elements of the extent of the parent, *a fortiori* none will be found in a subset thereof. The converse is not true: A candidate feature may be deliberate and have descendants that are *Ad Hoc*. Indeed, as we travel down the lattice, classes/types are taken out of extents while methods are added to the intents; if we remove an *anchor type* from the extent, then *typically*[6] the remaining classes/interfaces have no relationship among one another and, thus, form an *Ad Hoc* feature.

Although the value of Algorithm 3 lies mainly in identifying *ad-hoc features*, which correspond to refactoring opportunities, a finer analysis of the deliberate features provides additional insights on the design of the analysed systems: instances of *suboptimal refactoring*.

Second, we divide deliberate features into three categories, based on the relationship between the *anchor type* and the other elements of the extent, which we illustrate with examples from JHotDraw. The three categories correspond to three different *reuse idioms*: *type reuse*, *implementation reuse*, and *aggregation*.

1) The case where the common behaviour is embodied in a Java interface, and the extent consists of that interface, along with the classes that implement it. We call this category *Explicit Interface Implementation*. An example of such an extent corresponds to the red box in Figure 11, which contains the interface `Figure` with classes that implement it (`AbstractFigure`, `DecoratorFigure`, etc.)[7].
2) The case where the common behaviour is embodied in a Java class, and the extent consists of that class, often abstract, and its subclasses, which override some of its methods. We refer to this category as *Explicit Class Subclass Redefinition*. An example of such an extent is the blue box in Figure 11, which contains `AbstractFigure` and its subclasses.
3) The case where the common behaviour consists of a class `A`, which is then reused as a *component* in other classes that expose some of its behaviour. This category is called *Explicit Aggregation*. An example of such an extent is the green box in Figure 11, which contains `Figure` (the component) and the class `DecoratorFigure`, which has an attribute of type `Figure` that it exposes through its API.

Third, we further divide the three previous categories based on whether the *full extent* consists of interfaces/classes satisfying the category definition or only a *subset of the extent* satisfies the definition. The latter case occurs when the candidate feature includes the *anchor* type and its *related interfaces/classes* as well as *unrelated classes* that also provide the same subsets of methods. This case can happen, deliberately (through cloning) or not (repeated maintenance without proper refactoring to factor common methods). Thus, we divide each of the three previous categories into two categories: *Full Extent* and *Partial Extent*.

The example for *Explicit Interface Implementation* in Figure 11 corresponds to a *full extent*: The extent consists *solely* of the interface `Figure` and its implementation classes. For JHotDraw 5.0.2, we found many instances where the extent consisted of `AbstractFigure`, some of its subclasses, and other classes that are *not* descendants of `AbstractFigure`. Such candidate features belong to the *Partial Extent* category of *Explicit Class Subclass Redefinition*. In this case, the classes that are descendants of `AbstractFigure` are called *related types* (to the anchor type). Thus, the extent of a candidate feature that is of a *Partial Extent* category will consist of (1) an *anchor type*, (2) *related types* to the *anchor type*, and (3) *unrelated types*.

Another example from JHotDraw, from the interface implementation category, involves the concept of *animation*. Our tool returned a candidate feature whose extent consisted of the types {`Animatable`, `AnimationDecorator`, `BouncingDrawing`} and whose intent is the single method {`void animationStep()`}, which is the single method of interface `Animatable`. Only class `BouncingDrawing` implements the interface `Animatable`. Class `AnimationDecorator`, which is a subclass of `Decorator`, does not implement this interface either directly or indirectly. However, the method `void animationStep()` is present in `AnimationDecorator`, which is clearly an oversight.

Fourth, we further divide each one of the three *Partial Extent* categories into two categories corresponding to *Full Behaviour* or *Partial Behaviour* because the previous categories considered only the extents of the candidate features. Yet, the *intents* of the candidate features belonging to a *Partial Extent* category may or may not include *all* the methods shared between the *anchor type* and its *related types*.

6.2.3 Measuring Candidate Features

With the previous categories, the same candidate feature can satisfy the conditions for *several categories* as illustrated by the example of Figure 11, where the same extent, shown in a black box, matches three categories highlighted by the red, blue, and green boxes. In addition to being *Full Extent*, *Full Behaviour Explicit Interface Implementation*, this candidate feature is *also Partial Extent Explicit Class Subclass Redefinition*, because the extent contains the interface `Figure`, which is not a subclass of `AbstractFigure`. Idem for *Partial Extent Explicit Aggregation*. This is no accident: several of the design idioms used to package and reuse functional features *do* combine aggregation and inheritance; see Sections 3.1 and 3.2.

By definition, the shared behaviour among the *anchor type*, e.g., `AbstractFigure`, and its *related types* is a *superset* of the intent because, together, these classes are a *subset* of the extent. We define $BehaviourCoverage_{Anchor\ Related\ Types}$ to measure the relative size between that shared behaviour and the intent:

$$BehaviourCoverage_{Anchor\ Related\ Types} = \frac{|Intent|}{|Shared\ behavior\ among\ anchor\ and\ related\ types|} \quad (2)$$

6. Not *necessarily*: The extent of a candidate feature may include several overlapping *partial categories*, with their own *anchor types*, as illustrated in Figure 11

7. For this example and the next two, the reader can ignore the names of the coloured bounding boxes.

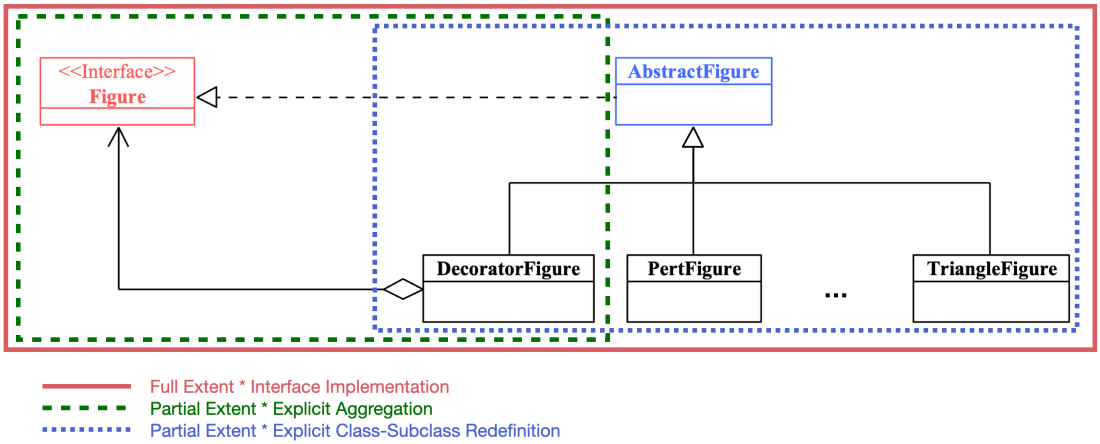


Fig. 11. Example of a candidate feature belonging to the category *Full Extent* of *Explicit Interface Implementation* (red bounding box and red anchor type), whose elements, taken in subsets, belong to the categories *Partial Extent* of *Explicit Class–Subclass Redefinition* (in green) and of *Explicit Aggregation* (in blue).

The $BehaviourCoverage_{Anchor\ Related\ Types}$ of the candidate feature in blue in Figure 11 is 0.8, which reflects that the four methods in the intent represent 80 per cent (four-fifths) of the shared behaviour between `AbstractFigure` and its subclasses in the extent.

An example of a candidate feature belonging to the *Full Behaviour* category of the *Full Extent* of *Explicit Class Subclass Redefinition* contains the abstract class `AbstractFigure` and (some of) its (concrete) subclasses. Its intent consists of only four methods: `connectionInsets()`, `handles()`, `read(StorableInput)`, and `write(StorableOutput)` while `AbstractFigure` declares 33 methods. The subclasses must only redefine these four *abstract* methods because `AbstractFigure` provides a default, working implementations for the other 29. Thus, even when a candidate feature has a full extent and full behaviour, its intent may consist of only a subset of the behaviour of the anchor type.

We define $BehaviourCoverage_{Anchor\ Domain\ Interface}$ to measure the coverage of the behaviour in the intent relative to the behaviour of the anchor type in the extent as follows:

$$BehaviourCoverage_{Anchor\ Domain\ Interface} = \frac{|Intent|}{|Anchor\ type\ interface|} \quad (3)$$

A candidate feature node can include several partial categories. For example, in JHotDraw, a candidate feature node with extent {`PointConstrainer`, `GridConstrainer`, `StandardDrawingView`} and with intent the single method {`constrainPoint(Point)`}, divides into:

- A *Partial Behaviour* of *Full Extent* of *Explicit Interface Implementation* with anchor type `PointConstrainer` and related type `GridConstrainer`.
- A *Full Behaviour* of *Partial Extent* of *Explicit Aggregation* with anchor type the component `PointConstrainer` and related type `StandardDrawingView`.

The $BehaviourCoverage_{Anchor\ Related\ Types}$ is 0.33 in the first partial category because its intent represents one-third of the shared behaviour between the anchor `PointConstrainer` and related type `GridConstrainer`. For the second partial category, it is 1 because the shared methods coincide with the intent methods. For both partial categories, the $BehaviourCoverage_{Anchor\ Domain\ Interface}$ is 0.33, i.e., the intent `Point constrainPoint(Point)` represents one-third of the full interface of `PointConstrainer`.

6.3 Steps 4 and 5: Detection Results

We present some measures of the lattices produced by our algorithm for the five systems presented in Section 2.4. Then, we describe the categorisation of the candidate functional features. We discuss detailed results for JHotDraw and JReversePro in Section 9.

We distinguish between (1) concepts in the lattices, (2) candidate features, and (3) candidate features with two or more methods in their intents, whose numbers are shown in Table 3. The systems have different designs: there is no correlation between the numbers of concepts and of candidate features. Table 4 summarises the categories of the candidate features. The last five columns represent disjoint sets. Table 5 shows the distributions of the different categories.

TABLE 3
Numbers of candidate features for the subject systems.

Systems	#Concepts	#CF	#CF >= 2
FreeMind	251	69	42
JavaWebMail	162	50	29
JHotDraw	394	154	123
JReversePro	105	26	18
Lucene	276	91	64

If we associate a high percentage of *Ad Hoc* candidate features with low code factorisation/maturity, then:

1) JHotDraw is the most mature, with only 35% *Ad Hoc* candidate features, 38.32% *Full Behaviour* of *Full Extent*, and 26.62% *Partial Extent*.
2) JReversePro is the least mature, with 92.3% *Ad Hoc* candidate features and only 7.6% *Full Behaviour* of *Full Extent* candidate features.

TABLE 4
Candidature feature categories.

Systems	#CF	#AD	#INT	#SUB	#AGR	#PART
FreeMind	69	32	1	6	6	24
JavaWebMail	50	23	0	5	7	15
JHotDraw	154	54	41	18	0	41
JReversePro	26	24	1	1	0	0
Lucene	91	47	4	23	4	13

CF: candidate functional features
AD: *Ad Hoc* features
INT: *Full Behavior* of *Full Extent* of *Explicit Interface Implementation*
SUB: *Full Behavior* of *Full Extent* of *Explicit Class Subclass Redefinition*
AGR: *Full Behavior* of *Full Extent* of *Explicit Aggregation*
PART: *Partial Extent* not included in previous categories

TABLE 5
Distribution of candidate feature categories.

Systems	%AD	%INT	%SUB	%AGR	%PART
FreeMind	46.38	1.45	8.7	8.7	34.78
JavaWebMail	46	0	10	14	30
JHotDraw	35.06	26.62	11.69	0	26.62
JReversePro	92.3	3.8	3.8	0	0
Lucene	51.65	4.4	25.27	4.4	14.3

3) FreeMind, JavaWebMail, and Lucene are in between. Lucene has both the second highest percentages of *Ad Hoc* (51.65%) and of *Full Behaviour* of *Full Extent* (34.05%) candidate features.

On one end of the spectrum, JHotDraw was developed as a *case study in design patterns* and the studied version, v5.1.2, shows that its design had gone through several iterations and its developers had opportunities to recognise functional features and to factor them as such, using interfaces, abstract classes, and delegations. On the other end, JReversePro was developed by two researchers, with a very specific/narrow focus: reverse-engineering Java bytecode. FreeMind and JavaWebMail are the work of several developers, which explains the better factorisation. They are relatively complex, multi-modal systems, which justifies poorer factorisation due to complexity. The previous observations depend on the "quality" of the candidate features in the *Ad Hoc* category and, to a lesser extent, in the *Partial Extent* category.

When considering candidate features belonging to the *Partial Behaviour* category, two cases can happen. The intent may contain less or the same methods as declared by the anchor type: developers designed adequately the hierarchy. The intent could also contain more/different methods: these *extra* methods are not declared by the anchor type but are still implemented by *all* its implementations/subclasses. Developers should study such cases to decide whether they missed factorisation opportunities or implemented a cross-cutting concern necessary to all implementations but not belonging to the anchor type.

We discuss in Section 7.4 threats to the validity of these results after performing a qualitative evaluation of some of these results in the next Section 7.

7 Quantitative and Qualitative Evaluations

We now present a quantitative evaluation of the tool that implements our approach. We called the tool F^3Miner for FCA-based Functional Feature Miner. We also present a qualitative evaluation of some candidate features proposed by F^3Miner as well a comparison with a previous work.

7.1 Implementation and Performance

We implemented in F^3Miner all the algorithms presented in the previous sections as an Eclipse plug-in, using the JDT plug-in for Java source-code parsing and analysis. This plug-in uses the *Visitor* design pattern to perform various tasks, including feature discovery and filtering, feature categorisation, and the computation of various metrics. It outputs the categories by decreasing the size of the related types set, and thus *Full Extent* categories are always output first. F^3Miner is released online[4].

Theoretically, assuming a bounded number of methods per class, the algorithms for Scenarios 1 and 2 have a complexity of $n \times log(n)$, where n is the number of classes and $log(n)$ represents the depth of the class hierarchy. (Real class hierarchies tend to be much flatter.) The computation for Scenario 3 is more involved. The computation of the incidence relationship has a similar complexity to Scenarios 1 and 2, i.e., $n \times log(n)$. The theoretical complexity of the incremental lattice construction algorithm that we implemented is $\mathcal{O}(m \times n^2 \times p)$, where m is the number of concepts in the resulting lattice, n is the number of classes, and p is the number of program elements par class [55]. The number m of concepts is unknown and has an upper bound of 2^n, corresponding to the powerset of the set of classes. Because the lattice generation algorithm falls into the category of *enumeration algorithms*—it *enumerates/generates* concepts—it is customary to talk in terms of *unit costs*, i.e., the cost of generating one lattice node, which is $\mathcal{O}(n^2 \times p)$. Finally, using the Visitor design pattern, the filtering and the categorisation of the lattice nodes is also $\mathcal{O}(m \times n \times log(n))$, where $n \times log(n)$ accounts for the detection of hierarchical relationships between the classes of an extent.

In practice, the lattice sizes are fairly small, compared to the theoretical limit, and the algorithms are fairly efficient. Table 6 shows lattice sizes, the numbers of candidate features, and execution times for seven different systems. Execution times were obtained on a MacBook Air, with 8 GB of RAM, and an Intel Core i5. The last column shows that the execution time grows slower than the theoretical complexity.

7.2 Analysis of Some Candidate Features

We focus on the *ad-hoc* candidate features (Scenario 3), which correspond to situations where developers/designers inadvertently implemented the same set of methods (functionality) in several places in the class hierarchy, without realising their cohesion or potential for reuse. While F^3Miner can *also* uncover functional features that have been recognised as such, and that have been packaged as interfaces or classes (Scenarios 1 and 2, see the discussion in Section 9.1), its *main value* is in identifying *ad-hoc* features, which are opportunities for refactorings to increase reuse.

Of the tested five subject systems (JHotDraw, JReversePro, JavaWebMail, Lucene, and FreeMind; see Section 2.4), we chose JHotDraw and JReversePro for detailed qualitative analysis. Our algorithm (Algorithm 3) has for objective

TABLE 6
Execution performance

Systems	LOCs	Types (n)	Methods	Concepts (m)	Cand. Feat.	Exec. Times		
						Total	Per Concepts	Over $m \times n^2 \times p$
FreeMind 0.7.1	65,490	198	4,785	251	69	00:45	0.179	1.89 E-07
JavaWebMail 0.7	10,707	115	1,079	162	50	00:38	0.234	1.89 E-06
JFreeChart 1.5	135,238	1,157	9,894	2,129	1,084	19:05	0.537	4.70 E-08
JHotDraw 5.2	9,419	171	1,229	394	154	01:54	0.289	1.38 E-06
JReversePro 1.4.1	9,656	87	663	105	26	00:16	0.152	2.64 E-06
Lucene 1.4	15,480	197	1,270	276	91	00:52	0.188	7.53 E-07
PMD 6.36.0	65,447	1,262	5,368	1,133	417	09:17	0.491	7.26 E-08

to find ad-hoc implementations of functional features, i.e., occurrences of the same sets of program elements in different parts of a system implemented by developers without recognising their similarity and encapsulating it in a class or interface. In addition, our algorithm, through its incidence relationship can also discover candidate functional features when developers *did recognise them* and encapsulated it in a class or interface reused through inheritance or delegation. Hence, the output of Algorithm 3 can complement that of Algorithms 1 and 2.

We now discuss its output using the two systems at both ends of the maturity spectrum of our systems: JHotDraw and JReversePro. We also discuss in Section 7.4 the threats to the validity of the results presented in the previous Section 6.3 and of this detailed qualitative evaluation.

7.2.1 JHotDraw

In Table 4, out of 154 candidate functional features, JHotDraw had 54 *Ad Hoc* (AD, no relationships between the classes of the extent), 41 consisting of an interface and a subset of its implementing classes, covering the full extent (INT), 18 consisting of a class and some of its subclasses, again covering the full extent (SUB), and 41 candidate features where the extent *contained* one of the previous two categories, along with unrelated classes (partial extent).

7.2.1.1 *Explicit Interface Implementation* Candidate Features: The 41 candidate functional features correspond to the following interfaces:

- `Figure`: 18 nodes of the lattice in all, each one consisting of `Figure`, along with *different* subsets of its implementing classes, and whose intents are different sets of methods.
- `Tool`: 10 nodes, consisting of `Tool` with different implementing classes and different intents.
- `Handle`: 8 nodes, similarly to `Figure` and `Tool`.
- `Connector`, `DrawingEditor`, `FigureEnumeration`, `Locator`, and `Painter`, with one node each.

For one given interface, referred to as the *anchor type*, the nodes of the lattice are hierarchically linked in relatively deep hierarchies, where the children of a node, 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 the implementing classes and a larger number of methods.

For example, Figure 12 shows an excerpt of the JHotDraw lattice and the nodes containing the interface `Figure`. It shows 18 interface-implementation nodes (yellow ellipses), 12 class–subclass nodes (orange rectangles), and

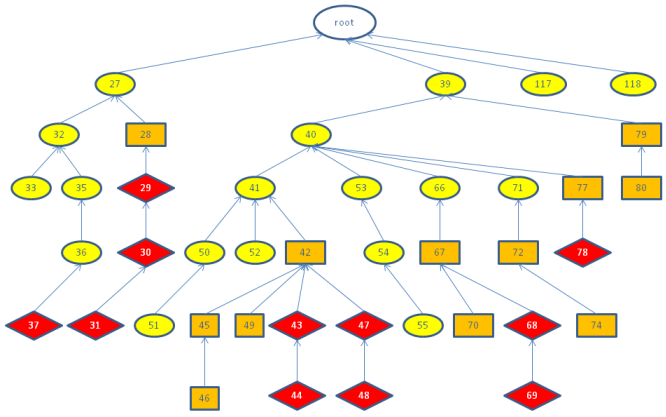


Fig. 12. Lattice of *some* candidate features *induced* by the interface `Figure`. Yellow ellipses represent interface implementations, orange squares class–subclass nodes, and red lozenges are *Ad Hoc* nodes.

11 *Ad Hoc* nodes (red lozenges). (It does not show partial nodes, which are descendants of these full nodes.) Its analysis shows that the extent of Node 40 (in the middle) contains the interface `Figure`, the class `AbstractFigure`, which implements `Figure`, and a set of descendants of `AbstractFigure`. Children of Node 40 (Nodes 41, 50 through 55, 66, and 71) have extents that are subsets of the extent of Node 40.

7.2.1.2 *Explicit Class Subclass Redefinition* Candidate Features: The 18 nodes include the following five classes as *anchor* types:

- `AbstractFigure`: 13 nodes, whose extents are 13 different sets of descendants of `AbstractFigure`.
- `Command`: 2 nodes.
- `ChopBoxConnector`, `DrawApplet`, and `DrawApplication`, one node each.

Many of these nodes are direct or indirect descendants of interface-implementation nodes. Figure 12 shows 12 class–subclass nodes (orange squares) out of the 13 nodes whose anchor type is `AbstractFigure`; all these 12 nodes are descendants of interface-implementation nodes. This observation is not surprising: as shown by Figure 11, any subset of the extent of a candidate feature that *does not include* `Figure` is a class–subclass node.

7.2.1.3 *Ad Hoc* Candidate Features: The usefulness of our algorithm and its results resides in the usefulness

of the *Ad Hoc* (54) and *Partial Extent* (41) candidate functional features because they reveal behavioural commonalities (features) that were not recognised by the developers. We specifically examine the *Ad Hoc* candidate features to understand and discuss the results of our algorithm.

Many *Ad Hoc* nodes are children of deliberate (partial or full) nodes, as shown in Figure 12, in which *Ad Hoc* nodes (lozenges) descend from deliberate nodes (ellipses or squares) and, as explained in Section 6.2.2, all the descendants of an *Ad Hoc* node can only be *Ad Hoc* nodes. JHotDraw has 54 *Ad Hoc* nodes, which are:

- 9 nodes that represent truly fortuitous co-occurrences of program elements in JHotDraw.
- 39 nodes that are descendants of full/partial nodes, such as the lozenges in Figure 12.
- 6 nodes that are "false positives" because the multiple occurrences of the program elements are, in reality, the same occurrence, but arrived at from different paths in the class hierarchy.

Figure 13 shows an example of such "false positives". The interface `ConnectionFigure` extends both `Figure` and `FigureChangeListener`, which are *not* related hierarchically and do *not* share any methods. Thus, our incidence relationship counts the domain interface of `ConnectionFigure` (and its implementing classes) as *two independent occurrences* because it comes from both `Figure` and `FigureChangeListener`. Therefore, Algorithm 3 returns a *Ad Hoc* candidate functional feature whose extent consists of the interfaces {`Figure`, `FigureChangeListener`} and whose intent includes the union of the methods of the two interfaces.

A similar situation occurs between any interface I and any class C if one of its subclasses, D, implements I, which yields a candidate functional feature whose extent includes the pair $\{I, C\}$ and whose intent includes the methods in the sub-hierarchy of D. The intent would include *more than* the union of the methods in $DOMINT(I)$ and $DOMINT(C)$, because the classes under the D sub-hierarchy would provide additional methods, specific to them.

7.2.1.4 Usefulness to Developers: The candidate functional features that are *potentially interesting* are some of the *Ad Hoc* nodes. We disregard the nodes that are descendants of nodes representing deliberate features, e.g., the lozenges in Figure 12, because they reflect commonalities between classes that are known to extend the same class or implement the same interface. We also disregard the nodes with classes/interfaces with multiple ancestors because they are false positives, as explained above. We focus on 7 nodes out of 9, which reveal interesting features that were not identified by the developers of JHotDraw 5.2:

- A figure is an indexed set of points, which appears in sibling classes `PolylineFigure` and `PolygonFigure`.
- The concept of *connection*, as distinct from `Connector`.
- The concept of *indexable, storable objects*.
- The similarity/redundancy between sibling classes `DrawingChangeEvent` and `FigureChangeEvent`.
- The similarity between `DrawApplet` and `DrawApplication`, beyond their common interface `DrawingEditor`, as shown in Figure 14 (left).

Our algorithm identified commonalities between `DrawApplet` and `DrawApplication`, suggesting that an abstraction `DrawingApplicationInterface` could be useful (Figure 14 (right)).

7.2.2 JReversePro

Table 4 shows that JReversePro had 26 candidate features, including one *Explicit Interface Implementation*, one *Explicit Class Subclass Redefinition*, and 24 *Ad Hoc* ones. We now discuss these candidates.

The *Explicit Interface Implementation* candidate feature is anchored by the interface `jreversepro.revengine.JReverseEngineer`, with implementing classes `JDecompiler` and `JDisAssembler`. The classes `JDecompiler` and `JDisassembler` represent the entry points to the two main functionalities of JReversePro, disassembling and decompiling bytecode. `JReverseEngineer` consists of a single method `void genCode()`, implemented in each class.

The *Explicit Class Subclass Redefinition* candidate feature has for anchor the class `JBlockObject`, and subclasses such as `JCaseBlock`, `JCatchBlock`, `JSwitchBlock`, `JDoWhileBlock`, etc., and intent {`getEntryCode()`, `String getExitCode()`}, which return the beginning and ending string of each block type. The extent includes all classes in the package `jreversepro.reflect.method`. This candidate feature has a $BehaviourCoverage_{Anchor\ Domain\ Interface}$ of 0.17. The relatively low coverage has two explanations:

- A *good* factorisation: most of the behavior (83%) of the subclasses of `JBlockObject` was factored in `JBlockObject`, leaving little to be redefined.
- A *poor* factorisation: there are few common methods (17%) between `JBlockObject` and its subclasses.

We study the domain interfaces of the subclasses of `JBlockObject`: if they define only the two methods of the intent, then it is a *good* factorisation. If each subclass of `JBlockObject` implements a different, larger set of methods, sharing only two common methods, then it is a *poor* factorisation. We observe that *ten out of twelve* subclasses of `JBlockObject` implement *only* the methods of the intent, thus a good factorisation. The subclasses `JForBlock` and `JDoWhileBlock` define more methods than the intent because these classes have more complex structures.

The study of the 24 `Ad Hoc` candidates reveals that:

- 3 are interesting abstractions *clearly* not recognised by the developers.
- 6 are mildly interesting features that maybe did not warrant explicit factorisation.
- 6 are abstractions that were *visibly* recognised by the developers but *not implemented to exploit commonalities*. They have extents consisting of pairs of graphical classes, with different names, in different packages, but offering similar functionalities with Java AWT or Java Swing. Surely there are other ways of handling such similar implementations, without

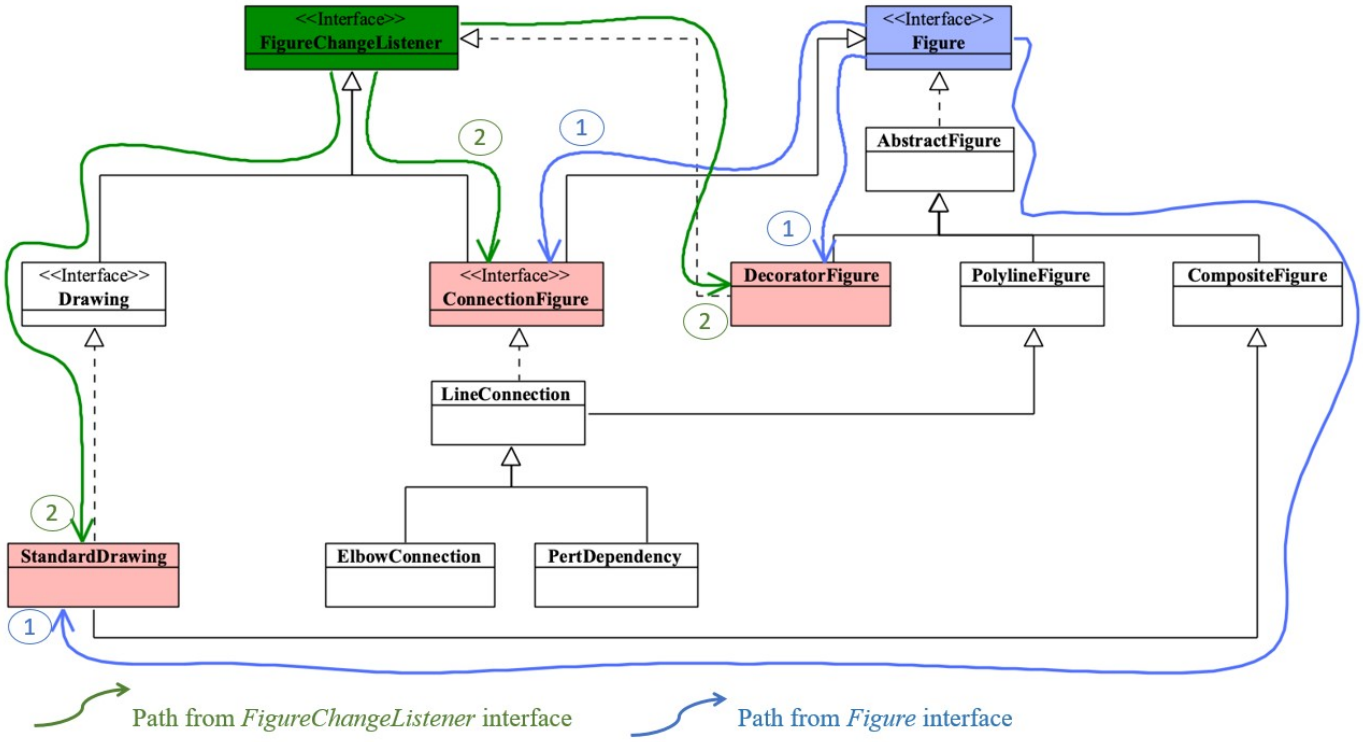


Fig. 13. Starting from the interfaces `Figure` and `FigureChangeListener`, if there are two *paths* to the same type (interface or class), the methods of that type (and those of its descendants) will be *erroneously* treated by our algorithms as independent occurrences of those methods and will be included in the intent of the candidate node whose extent includes/consist of `Figure` and `FigureChangeListener`.

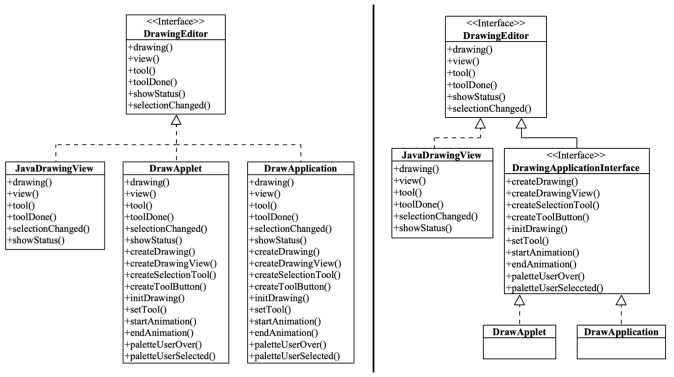


Fig. 14. Commonalities between the three classes `DrawApplet`, `DrawApplication`, `JavaDrawingView` using the interface `DrawingEditor`. The novel abstraction `DrawingApplicationInterface` could be useful to factor methods in `DrawApplet` and `DrawApplication`.

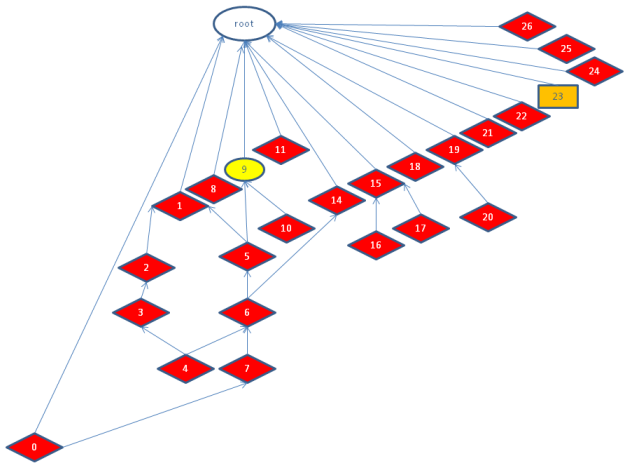


Fig. 15. Lattice of *some* candidate features found in JReversePro. Yellow ellipses represent interface implementations, orange squares class–subclass nodes, and red lozenges are *Ad Hoc* nodes. (Some of the transitive edges between nodes are omitted for clarity).

code duplication, e.g., using the Bridge and–or Factory design patterns.

- 9 have extents consisting of two or more interfaces and intents consisting of the union of the domain interfaces of the classes implementing these interfaces. These are false positives: we had similar false positives with JHotDraw—six of them.

7.3 Comparison with Previous Work

We now compare our approach to a previous, existing approach. We reported in Section 2 that there are only a few works that attempt *to discover* features in existing software systems. Table 7 summarises the works most similar to ours. It shows that these works span more than 20 years but that most do not provide public implementation, systems, and results to allow comparisons. We highlight two works that are most closely related to ours and that provide some public information that could be used for comparison: [25] and [20]. These two works are quite recent and provide

TABLE 7
Detailed comparison of approaches related to feature discovery. Highlighted rows show works for which most information for replication is public.

Approaches (Years)	Main Analysis Techniques and Tools	Implementation (Public?)	Datasets (Public?)	Results (Public?)	Comparisons with Others	Comments
[21] (2005)	Manual (static) systems analyses	N/A	Sonar systems (Unavailable)	No	No	
[22] (2005)	Static analysis (Rational Rose), manual analyses	N/A	Home service robot applications, Samsung Home Robot (Unavailable)	No	No	
[23] (2007)	DaRT, Understand for C++, Tau, Together, Rhapsody, DOORS, then manual analyses	N/A	G Platform (Unavailable)	No	No	
[24] (1997)	Clustering techniques	No	None	No	No	
[25] (2017)	Static analyses, clustering	Yes$^{\alpha}$	12 open-source systems	No	No	Microservices identification
[20] (2021)	FoSCI: dynamic analysis, search-based functional atom grouping	Yes$^{\beta}$	JPetStore + 6 open-source Web applications	Table 2, else not public	LIMBO, WCA, and MEM, metrics only	Service identification
[27] (2006)	Slicing, manual analysis	No	Yes (COBOL)	Table 1	No	
[28] (2014)	Static analysis and clustering	No	No	Figures 14-18	No	Architecture deficiencies
[29] (2020)	Static and dynamic analyses (ServiceCutter), manual analyses	No	in|FOCUS (Proprietary)	No	No	

$^{\alpha}$: https://github.com/gmazlami/ ; $^{\beta}$: https://github.com/wj86/FoSCI

results on available open-source systems. However, they do not provide complete results, except for some in [20].

Consequently, we chose to compare F^3Miner to the work by Jin et al. [20] by applying F^3Miner on JPetStore[8] and using the Tables 2 and 4 provided in their work. We chose this previous work because it provides some results on an available open-source system, JPetStore. Also, it provides services that, we argue, are similar enough to the concept of features.

Figure 16 shows the features discovered by F^3Miner in JPetStore. F^3Miner discovered 25 candidate features, which we report for replication in an online document[4]. Among these 25 candidate features, we report in Table 8 the features discovered by F^3Miner and their corresponding services identified by the approach of Jin et al. The table shows that F^3Miner could discover features that match the four service candidates reported by Jin et al. It highlights that both approaches find common classes that, combined with others, are candidate features/services.

In addition, F^3Miner discovered 21 other candidate features. We cannot describe all these features here for lack of space but, among these, 10 are ADHOC candidate features, which represent opportunities to refactor code or create a new abstraction missed by developers.

For example, the candidate feature #1 is *Ad Hoc* and has for extent the classes `org.mybatis.jpetstore.domain.Category`, `org.mybatis.jpetstore.domain.Product`, and `org.mybatis.jpetstore.domain.Sequence` and for intent the methods `String getName()` and `void setName(String)`. It shows that the main domain abstractions all provide the same set of two methods. This candidate feature is an opportunity to introduce a new abstraction, e.g., `NamedEntity` that could be inherited by the domain classes.

As another example, the candidate feature #58 is also *Ad Hoc* and has for extent the classes `org.mybatis.jpetstore.domain.CartItem` and `org.mybatis.jpetstore.domain.LineItem` and for intent the six methods `BigDecimal getTotal()`, `Item getItem()`, `int getQuantity()`, `void calculateTotal()`, `void setItem(Item)`, `void setQuantity(int)`. On the other hand, the delegation candidate feature #27 has as an extent the classes `CartItem`, `LineItem` and `Item`, and as intent the methods `int getQuantity()`, and `void setQuantity(int quantity)`. This indicates that both `CartItem` and `LineItem` wrap an `Item` and delegate a common set of methods to it. In view of these results and the fact that `Item`, `CartItem`, and `LineItem` do not share a common super-type, we conclude that there is a refactoring opportunity missed here by the developer.

Thus, with this comparison, we show that F^3Miner compares favourably with the closest related work, which provides results on open-source systems. We also show F^3Miner discover other candidate features, which are opportunities for refactoring the system by introducing new abstractions or merging similar abstractions.

7.4 Threats to Validity of the Evaluations

There are three main threats to the validity of the results reported in these quantitative and qualitative evaluations: (1) what constitutes an interesting or useful functional feature and how to compare features (*construct validity*), (2)

8. https://github.com/mybatis/jpetstore-6

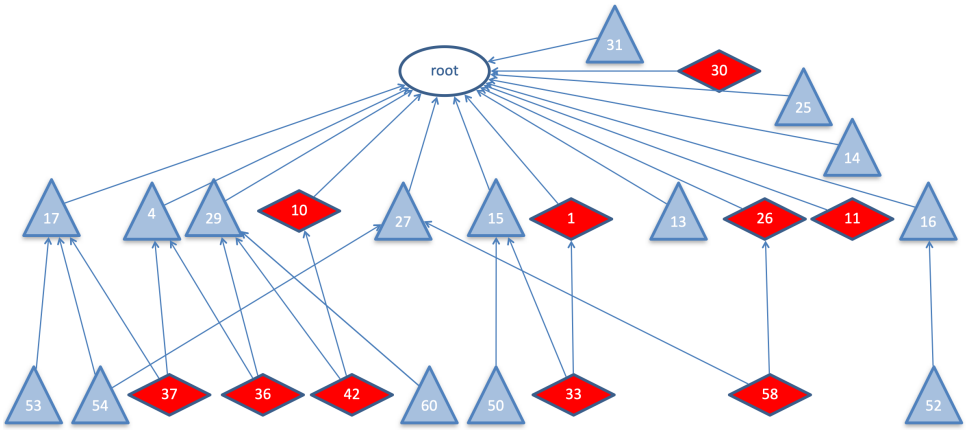


Fig. 16. Lattice of candidate features discovered by F^3Miner in JPetStore (red lozenges: *Ad Hoc* nodes, blue triangles: *delegation* nodes).

TABLE 8
Features discovered by F^3Miner that correspond to services identified by the approach of Jin et al. (We replaced the package names `org.mybatis.jpetstore` with ... for the sake of clarity.

#	Type	Extent and Intent	#	Entities
16	*Partial Extent* *Full Behaviour* *Explicit Aggregation*	....mapper.CategoryMapper **....service.CatalogService** **....web.actions.CatalogActionBean** getCategoryList()	SC0	....domain.Category **....service.CatalogService** **....web.actions.CatalogActionBean** ....domain.Product ....domain.Item ....domain.Sequence
25	*Full Extent* *Full Behaviour* *Explicit Aggregation*	....mapper.OrderMapper **....service.OrderService** getOrdersByUsername(String) getOrder(int) insertOrder(Order)	SC1	....domain.LineItem ....web.actions.OrderActionBean **....service.OrderService** ....domain.Order
			SC2	....domain.Cart ....domain.CartItem ....web.actions.CartActionBean
31	*Full Extent* *Full Behaviour* *Explicit Aggregation*	....mapper.AccountMapper **....service.AccountService** insertAccount(Account) updateAccount(Account)	SC3	**....service.AccountService** ....web.actions.AccountActionBean ....domain.Account

whether we are reliable judges of this usefulness (*internal validity*), and (3) whether the results can be generalised to other software systems (*external validity*).

With respect to construct validity, we argued elsewhere that *reusability* is a combination of *usefulness* and *usability* [56] where *usefulness* refers to the extent to which a program element is *often* used while *usability* refers to the ease with which it can be used.

Algorithm 2 identifies program elements that are *de-facto* (re)usable: classes and interfaces. Thus, we focused on *usefulness*. We were careful, both in our algorithms (e.g., the concept of *domain interface*) and in our analyses, to focus on domain semantics, and we made sure to distinguish between features that were central to the application domains of the studied systems, from the ones that were secondary (e.g., GUI features for JReversePro and FreeMind). We confirmed the *usefulness* of many of such features by their multiple occurrences in the code because many of the same features were also uncovered by Algorithm 3. The *usefulness* of *Ad Hoc* candidate features is *de-facto* because all occur multiple times by definition. However, we only performed a manual comparison of the features and did not assess some objective measures, for example using cohesion [57] or coupling [58], which we plan in future works.

We conducted a separate experiment, described in Section 8.3, where we asked participants to assess the research hypotheses that underlie our work.

With respect to internal validity, the authors have, at one time or another, working with the systems studied for other reasons (e.g., we *used* both JHotDraw and JReversePro in our previous work on design patterns and architecture recovery [59, 60]), and we have a comprehensive knowledge of these systems and their implementations. Further, we conducted a separate experiment described in Section 8.4, where we asked participants to assess the same candidate features, although they had less time to become familiar with the software systems under study.

With regard to external validity, we want to perform more experiments. However, we were careful in choosing the systems to cover a broad spectrum of (1) sizes (10 vs. 70 KLOC), (2) maturity (JHotDraw vs. JReversePro), 3) framework (JHotDraw) vs. monolithic system (JReversePro), (4) mostly graphic (JHtoDraw, FreeMind) vs. mostly command line (JReversePro), and (5) system (JReversePro) vs. end-user

applications (JavaWebMail).

The scenarios, categories, and resulting features are general and do not depend on the programming language. Although we used Java to illustrate and apply our work, it should generalise to any programming language where: (1) there is some form of explicit inheritance and association relationships and (2) a similarity in method signatures suggests a similarity in function. These assumptions should apply to any statically-typed programming language, such as C++, C♯, Scala, or TypeScript.

8 User Validation

We now present an end-user validation, performed with developers, on three aspects of F^3Miner:

1) The *research hypotheses about ad-hoc features*, their imprint in actual code, the benefits of identifying them (Section 8.3).
2) The *quality of the ad-hoc candidate features* proposed by F^3Miner (Section 8.4).
3) The *usability and usefulness of* F^3Miner as a design and re-engineering tool (Section 8.5).

We start by describing the experimental protocol (Section 8.1) and the participants (Section 8.2) before presenting the questions and the participants' answers in Sections 8.3, 8.4, and 8.5. We discuss threats to the validity of this user validation in Section 8.6.

8.1 Experimental Protocol

We conducted the experiment online for the participants' environmental health and safety as well as their convenience.

Prior to the experiments, the participants received information about installing and using F^3Miner. All of the material for the experiment is available online[4].

Each experiment session lasted about two and a half hours and consisted of five steps:

1) Delivering a brief tutorial, explaining the research problem and providing enough elements about the research methodology to allow participants to answer the subsequent questions.
2) Filling out an online questionnaire about the participants' background, knowledge, and experience in object-oriented design and programming in general, and Java in particular.
3) Evaluating the research hypotheses and methodology, through an online questionnaire.
4) Evaluating the quality of twelve ad-hoc candidate features, among those discussed in Sections 7.2.1 and 7.2.2, through an online questionnaire.
5) Evaluating the usability and utility of the tool, also through an online questionnaire.

We designed Steps 3, 4, and 5 using the *Goal Question Metric* framework to determine the tasks to be performed and the elements to be measured for performance. We discuss any issues and trade-offs in the corresponding sections.

8.2 Participants and Experimental Sessions

We recruited twelve participants among the members of the authors' research labs, which host about 120 graduate students and postdocs. We sent a general invitation to the members of the labs. All participants were unpaid volunteers. All participants could withdraw from the experiment at any time, for any (or no) reason, without any consequence. All answers are anonymous and anonymised.

Table 9 gives some descriptive statistics on the participants. All participants, who are all software engineering students, are mostly moderately experienced Java developers. They were divided into two almost equal halves regarding their knowledge of the Prototype and Singleton patterns.

One of the authors remained connected throughout the experimental sessions to answer the participants' questions.

8.3 Research Hypotheses about Ad-Hoc Features

Our work rests on three hypotheses:

H_1: Classes in legacy OOP tend to combine distinct functional features;
H_2: Discovering such functional features is useful for understanding, debugging, and evolution;
H_3: Functional features are implemented using specific idioms, discoverable in the source code.

The purpose of this part of the experiment is to validate these hypotheses, focusing on Ad-hoc functional features (Scenario 3) for the hypothesis H_3.

The questionnaire was designed with three objectives. First, it helps evaluate our hypotheses. Second, it leads participants to take a *critical look* at the research methodology, which will help them answer precise questions about the quality of the candidate features (next section). Finally, it allows interpreting the results of the participants' evaluation of the candidate features.

The questionnaire consisted of four yes/no questions pertaining to H_1, H_2 and H_3, each with an open-ended question to justify a negative answer. For example, for H_1, the question was:

- We observed, and thus hypothesised, that domain classes in legacy OOP systems often implement several functional features, regardless of how they are composed (multiple inheritance, aggregation, aspect weaving, or ad-hoc). Do you agree?
- If not, why not?

We asked a more precise question for H_3:

- We argued, with a toy example, that class members pertaining to a functional feature may be distributed among many classes, and not be factored in a single class. For example, the functionality may itself be specialised into several flavours. Do you agree?
- If not, why do you think that it is unlikely that class members related to a functional feature be distributed amongst many classes?

Table 10 summarises the participants' answers. The actual questionnaire and individual answers are available online[4].

TABLE 9
Participants' Descriptive Statistics

Question	#Answers	Answers							
Experience in software development	12	0 - 6 months	0	6 - 12 months	1	1 - 3 years	3	3+ years	8
Experience with Java	12	0 - 6 months	1	6 - 12 months	2	1 - 3 years	3	3+ years	6
Knowledge of the feature concept	12	None	6	Theoretical	2	Practical	4		
Knowledge of design patterns	12	None	1	Theoretical	7	Practical	4		
What is the implementation difference between Prototype and Singleton patterns?	12	Unfamiliarity with the patterns			5	Familiarity with the patterns			7

TABLE 10
Evaluating the research methodology (hypotheses). For the last question, 'yes' stands for 'reasonable' and 'no' stands for "too permissive'.

Question #	Question (summary)	Hypothesis	#Answers	Yes	No	% agreement
Q_1	Classes implement multiple features in legacy OOP systems	H_1	12	10	2	83.33 %
Q_2	It is worth identifying such features in legacy OOP systems	H_2	12	12	0	100 %
$Q_{3.a}$	Class members pertaining to a feature may be distributed among many classes	H_3	12	10	2	83.33 %
$Q_{3.b}$	When counting occurrences of sets of class members within sub-hierarchies, we should ignore structure (topology)	H_3	12	11	1	91.66 %

Participants agreed, from 80% to 100%, with our research hypotheses. Considering Q_1 (two negative answers), one participant said that what we describe corresponds to the Blob anti-pattern and felt that we may be over-generalising. The other negative participant wrote: "If you [...] observed something happening, you don't need to hypothesise about its occurrence".

With regard to question $Q_{3.a}$, one negative participant said that this was contrary to recommended practice. The second negative participant argued that some class members may participate in several functional features, with which we agree: we do not see a contradiction between the two statements.

With respect to question $Q_{3.b}$, a single participant thought that we were being too permissive by not taking into account the structure/topology, i.e., where exactly in the sub-hierarchy do the various elements of the intent appear; s/he also mentioned overloading as a counter-example.

8.4 Quality of the Ad-Hoc Candidate Features

We apply the Goal Question Metric (GQM) framework. The goal of F^3Miner is to discover non-factored, or partially factored, reusable functional features in legacy object-oriented source code. Given this goal, we can ask several questions to assess whether F^3Miner achieves its goal:

1) Given a system S, with known non-factored functional features $F_1, F_2, ..., F_n$, does F^3Miner discover these features?
2) Given a system S, with known non-factored functional features $F_1, F_2, ..., F_n$, does F^3Miner discover them *faster* than a manual inspection?
3) Given a system S , how good are the candidate features $CF_1, ..., CF_m$ discovered by F^3Miner?

The first two questions require a *ground truth* against which to compare the output of F^3Miner, which could be obtained by having experienced designers study the subject software systems and identify non-factored (ad-hoc nodes) or partially-factored (partial-extent nodes, see Section 6.2.2) functional features that may be worth (re)factoring.

However, an *exhaustive* discovery of reusable functional features, and more generally refactoring opportunities, is neither practical—time-consuming—nor necessary. Indeed, with reuse and refactoring, there is a cut-off point below which, packaging the intent of an ad-hoc node into a class may not be worth the refactoring effort.

Similarly, the second question is neither practical nor fair to experiment participants. It is impractical because identifying reusable functional features in a non-familiar system of 60 or 70 KLOCs would take hours. It is unfair because F^3Miner completes its computations in under two minutes, even for the largest of the five systems, thanks to an efficient incremental algorithm for building Galois lattices.

Therefore, we chose to focus on the third question and have participants assess the candidate features discovered by the tool, similarly to what we did in Section 7.2, but as an independent validation.

We asked the participants to assess the *quality* of ad-hoc candidate functional features because those correspond to reusable features that developers of the software systems did *not* factor and duplicated in the class hierarchies.

We defined the quality of an ad-hoc candidate feature using three distinct characteristics of the discovered candidate features that we formulate as hypotheses to be verified on the individual candidates:

H_4: The multiple occurrences of the same set of methods (the *intent* of a candidate, see Section 6.2) is meaningful/non-fortuitous;
H_5: The methods forming the *intent* of a candidate are *functionally cohesive*, i.e., they work towards the same functionality;
H_6: The identified feature (the *intent*) represents a *useful* domain abstraction.

Hypothesis H_4 embodies a *key* methodological choice made in our approach: *we focus only on method signatures*,

disregarding method bodies. We assume that two methods that have the same signature and that appear in different parts of a class hierarchy should have the same objective and, therefore, the same behaviour. The other two hypotheses pertain to the intrinsic quality of the candidates.

We associated each of the hypotheses with a yes/no question Q_4, Q_5, Q_6. For H_6/Q_6, if the answer is negative, we asked participants additional questions about the candidate feature:

$Q_{6.a}$: Do you see a design anomaly that could have explained how/why this multiple occurrences of a set of methods occurred?

$Q_{6.b}$: Do you see a rule that we could use to eliminate these candidate nodes (false positives)?

Thus, essentially, we asked the participants to perform the same kind of analyses that we performed in Section 7.2.

Our tool identified 54 ad-hoc candidate functional features in JHotDraw and 24 in JReversePro.

We chose a subset of these ad-hoc features: 12 features, six from JHotDraw and JReversePro, which we chose randomly. The nodes used for the experiment are available online[4].

Table 11 presents the aggregated results for questions Q_4, Q_5, Q_6. The last column shows our own evaluation of the same nodes, which was the basis for the results presented in Sections 7.2.1 and 7.2.2. By and large, the results suggest that the assessments of the experiment participants are consistent with ours. Table 12 aggregates those results, per 'type of node' (*Positive*, *Positive but 'Uninteresting'*, and *False positive*). It shows that ad-hoc nodes that we deemed as positive *and* interesting, scored consistently higher than false positive features: 68% versus 61% on Q_4, an average cohesion of 3.75, out of 5, versus 3, and a score of 79 %, versus 50 %, for question Q_6. These results should be put into the context of (1) the small number of ad-hoc nodes overall (12), and of each category (8, 1, and 3), (2) the fact that the participants had zero familiarity with JHotDraw and JReversePro prior to the beginning of the experiment, and little time to assess each ad-hoc node–about 5 minutes each, and (3) the relative diversity of their level of expertise in Java and OO design (see Table 9).

Looking at the answers to questions $Q_{6.a}$ ("if it is a false positive, what design anomaly could explain it") and $Q_{6.b}$ ("what rule can you think of that would filter out this false positive"), the participants had similar answers to ours regarding the 'why' and 'how to fix' of the false positive (`JReversePro N1`) and uninteresting (`JReversePro N3`) nodes of JReversePro, respectively. For the JHotDraw nodes, the participants who agreed with our assessment regarding the two false positives (nodes `JHotDraw N5` and `JHotDraw N6` in Table 11) also had similar to answers for questions $Q_{6.a}$ and $Q_{6.b}$[9].

8.5 Usability and Usefulness of F^3Miner

We now evaluate the usability and utility of the tool, also following the GQM framework and an online questionnaire.

Given the tasks carried out by the participants during the previous parts of the experiment, we focused on the evaluation of the following aspects:

- How easy is F^3Miner to install and use by the participants? Although our tool is a research prototype, we assessed the participants' perception of the tool's overall usability with this general question.
- Is the graph generated by F^3Miner easy to understand? We wanted to know if the forms and colours of the nodes presented by F^3Miner make the graph easy to read.
- Does the graph generated by F^3Miner support understanding the design of the analysed system? We wanted to know if the participants could navigate within the graph and identify features in a system.
- Does the generated graph support evaluating the design quality of the analysed system? Design quality cannot be evaluated absolutely by the participants because it requires a deep knowledge of the system so we chose a relative evaluation.

We associated the first three aspects with questions Q8, Q9, and Q10, whose answers were on a 5-point Likert scale, from "Very easy", to "Very difficult". For the last aspect, we provided the participants with graphs generated by F^3Miner for four systems and, in Q11, asked participants to assign them a relative design quality score on a 4-point Likert scale, from "having the highest number of refactoring opportunities" to "having the smallest number of refactoring opportunities".

We used two of the systems from our qualitative evaluation, JHotDraw and JReversePro. They are the most and the least mature in terms of the discovered *Ad Hoc* candidate features, see Section 6.3. We added two actively maintained OO systems: JFreeChart and PMD. JFreeChart is a framework that supports developing charts. PMD is a static source code analyser that uncovers programming flaws through the use of built-in rules. Table 13 reports some descriptive statistics for these four systems, including the size of the graph generated by F^3Miner, in the last column.

Figure 17 displays the results for questions Q8, Q9 and Q10. For Q8, most of the participants found that the tool is Easy or Very Easy to use (10 out of 12). However, for Q9, some participants found it difficult to understand the graphs (5 out of 12). We believe that this is due to the too-short amount of time assigned during the tutorial for presenting the graphs (forms and colours). For Q10, participants found it Easy to Average (10 out of 12) to navigate the graphs and identify system features.

Figure 18 summarises the results for Q11: 11 out of 12 participants answered the question. It shows that most participants (10 out of 11) recognised JFreeChart as being the system with the highest number of refactoring opportunities, which matches findings in previous works that analysed JFreeChart (e.g., [61, 62]). Most of the participants assigned a good score to JHotDraw: 6 out of 11 gave a 3 or 4 and none a score of 1, which conforms with our analysis in Section 6.3 and the fact that JHotDraw was designed with reuse (and design patterns) in mind.

Regarding PMD, we expected results similar to those of JHotDraw because PMD is a framework/tool enforcing

9. For `JReversePro N1`, `JHotDraw N5` and `JHotDraw N6`, the classes of the extent had a `void main(String[] args)` method, which 1) were inappropriate, from a class design point of view ($Q_{6.a}$), and 2) should not have been included in the *domain interfaces* of the classes at hand.

TABLE 11
Aggregated results for 12 ad-hoc candidate features. The number between parentheses in the node ID refers to F^3Miner generated IDs.

Ad-hoc node ID	Question Q_4			Question Q_5		Question Q_6			Our Assessment
	Yes	No	%Yes	Avg. Cohesion	Stdv. Cohesion	Yes	No	%Yes	
JRevPro N1 (11)	6	6	50.00	3.08	1.88	3	9	25.00	False positive
JRevPro N2 (12)	7	5	58.33	3.42	1.08	10	2	83.33	Positive
JRevPro N3 (14)	8	4	66.67	4.08	1.68	7	5	58.33	Uninteresting
JRevPro N4 (15)	8	4	66.67	3.83	1.53	8	4	66.67	Positive
JRevPro N5 (24)	8	4	66.67	3.16	1.40	7	5	58.33	Positive
JRevPro N6 (20)	7	5	58.33	3.83	1.75	9	3	75.00	Positive
JHotDraw N1 (5)	6	6	50.00	3.5	1.31	9	3	75.00	Positive
JHotDraw N2 (20)	10	2	83.33	4.42	1.00	12	0	100	Positive
JHotDraw N3 (21)	8	4	66.67	4.16	1.53	10	2	83.33	Positive
JHotDraw N4 (110)	11	1	91.67	3.67	1.37	11	1	91.67	Positive
JHotDraw N5 (128)	8	4	66.67	2.33	1.50	6	6	50.00	False positive
JHotDraw N6 (135)	8	4	66.67	3.67	1.72	9	3	75.00	False Positive

TABLE 12
Answers to Q_4, Q_5, and Q_6, aggregated by qualitative evaluation per Sections 7.2.1 and 7.2.2.

Category per Sections 7.2.1 and 7.2.2	Average % Yes for Q_4	Average for Q_5	Average % Yes for Q_6
Positive	67.71	3.75	79.17
Positive but 'uninteresting'	66.67	4.08	58.33
False positive	61.11	3.03	50.00

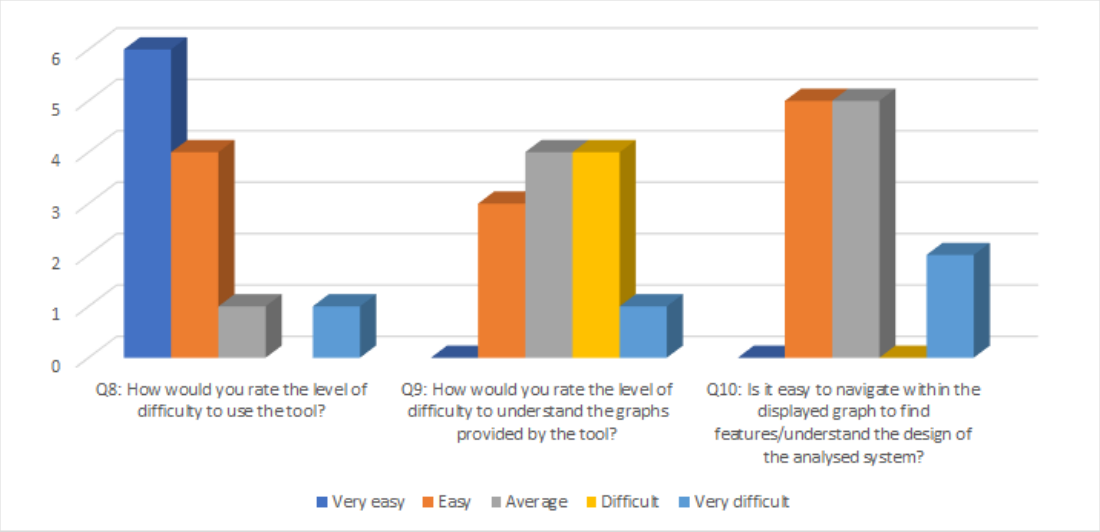


Fig. 17. Results of Q8, Q9 and Q10.

TABLE 13
Some metrics for the OO systems used for Q11.

System	#LOCs	#Types	#Methods	#Graph-Size
JHotDraw 5.2	9,419	171	1,229	154
JReversePro 1.4.1	9,656	87	663	27
JFreeChart 1.5	135,238	1,157	9,894	1,084
PMD 6.36.0	65,447	1,262	5,368	417

best practices in programming. However, the results were inconclusive: most participants gave a score of 2 or 3 (9 out of 11). We may explain these results by the size of PMD, which is much bigger than JHotDraw's in terms of LOCs, types, and methods, as shown in Table 13. Thus, the graph generated by F^3Miner for PMD is much bigger in the number of nodes and candidate features than JHotDraw's. We believe that some participants based their negative score on the high number of candidate features.

Regarding JReversePro, the results are mixed. Six participants gave a low score (1 or 2), which matches the results of our analysis in Section 6.3, while five participants gave a positive score (3 or 4). We explain these mixed results by the fact that JReversePro is small and has one main feature (reverse engineering compiled Java code). Thus, the size of the generated graph was small and some participants may have based their positive assessment on the low number of candidate features.

8.6 Threats to Validity of the User Validation

There are three main threats to the validity of the results of this user validation: threats to (1) their construct validity, pertaining to the design of the validation and its participants, (2) their internal validity, concerning the causes of the results, and (3) their external validity and generalisation.

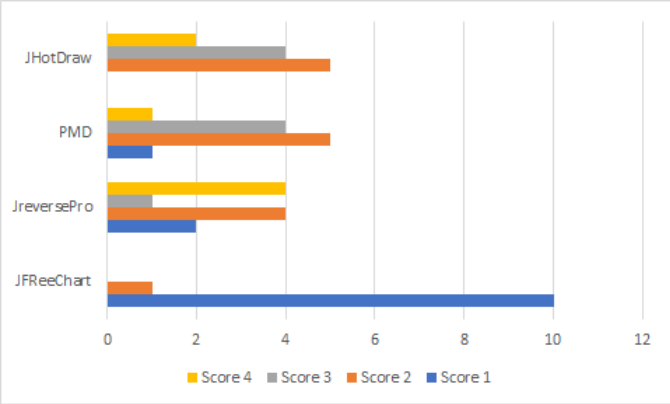


Fig. 18. Results of Q11 for 4 systems. Score 1 indicates the system has the highest number of refactoring opportunities, while score 4 indicates the system has the smallest number of refactoring opportunities.

The design of and participants to this user validation are threats to its construct validity in different aspects. First, the questions and questionnaires may have led participants to give certain answers. We mitigated this aspect by defining the questions as simply as possible. We also used the usual Likert scales, familiar to most participants. Second, we chose systems of manageable sizes with which participants had no experience. Thus, we attempted to put all the participants on an equal footing so that their answers are valid and comparable. Third, we used a convenient sampling of participants who are not representative of all developers. However, we argue that graduate students will be the next generation of developers and, as such, are somewhat representative of junior developers in the field.

The choices of the systems, of the graphs, and of the questions and questionnaires impact our results and form a threat to their internal validity. Indeed, it is possible that our results are not due to these choices but due to, for example, the intrinsic complexity of the chosen systems and the complexity of working with unfamiliar graphs. We mitigated this threat by asking 12 participants, allowing for some control over the variability between participants. We also mitigated this threat by using systems known for their good and bad design and others with expected average design quality. Finally, we asked participants to explain their answers to avoid random answers.

Finally, we are aware that our results may not generalise to other participants and other systems and that more experiments would be necessary to confirm their generalisability. However, our qualitative and user validations showed that our algorithms and tooling did help in discovering candidate functional features and creating refactoring opportunities. Hence, they gave us confidence that our algorithms did indeed help our purpose, which is discovering reusable functional features in legacy object-oriented systems. They did not allow computing the extent to which they help, which is future work.

9 Discussion

We now discuss various aspects of our work.

9.1 Scenarios

Results from Algorithms 1, 2, and 3 show that our techniques are interesting in two ways:

- They can help uncover those functional features that *have been codified* in the program, using multiple inheritance or delegation because this information provides developers with a *cartography* of the system and its various functional features, which could be combined differently to build new systems.
- They can also uncover abstractions that *have not been* identified by developers, such as *Ad Hoc* candidate functional features discussed in Section 6 to guide future refactorings of the system.

The algorithms also produce intersecting but different sets of features:

- The algorithm pertaining to multiple inheritance identifies features when a class has two ancestors by marking those ancestors as potential features to be (re)used, whereas *Explicit Interface Implementation* and *Explicit Class Subclass Redefinition* candidate features come from a class/interface with two different sub-hierarchies that implement the same behaviour, and propose those interfaces or classes (i.e., the anchors) as features.
- The algorithm for detecting instances of delegation identifies cases of aggregation, even if the component appears *once* whereas *Explicit Aggregation* candidate features happen when a component appears in *several* aggregates. Again, there is an overlap, but the features discovered by each algorithm are different.

9.2 Variants

A common limitation to existing work on codifying idioms and identifying occurrences of these idioms is the handling of variants of the idioms. Indeed, like natural languages, programming languages allow developers to express similar idioms in a great variety of ways.

The handling of variants is a limitation in many software-engineering problems. For example, the identification of occurrences of design motifs—the solutions parts of design patterns—suffers from this limitation. While solutions have been proposed, for example, the use of explanation-based constraint programming to relax constraints modelling design motifs [63], they may still miss occurrences that do not satisfy the constraints.

This limitation is also conceptual. The handling of variants requires answering "how different an occurrence must be from a given idiom before it ceases to be legitimately considered an occurrence of that idiom?" Answering this question often depends on the purpose and knowledge of the developers who want to identify such occurrences.

In this work, we designed the idioms and their detection so that they can handle variants that we considered legitimate, and whose accuracy we validated qualitatively in Section 7.2 and through a user validation in Section 8.

For delegation, in Section 5, the detection algorithm handles variants through three mechanisms: 1) by using the concept of *domain interface*, 2) by not requiring that

the aggregate (delegator) and the component (delegate) implement the same interface, and 3) by using the concept of *coverage ratio*. For ad-hoc functional features, introduced in Section 6, the variance is embodied in the incidence relationship we used to build the initial Galois lattice of candidate features.

The handling of variants also relates to the possible combination of idioms, for example, inheritance and delegation. Our algorithms and tool naturally handle such combinations, which would be reported as different candidate functional features. The model of Figure 11 is an actual example from JHotDraw where the same set of Java types contains three overlapping categories: that of an interface (`Figure`) with classes that implement it (`AbstractFigure`. `DecoratorFigure`, `PertFigure`, etc.), of a class (`AbstractFigure`) with its subclasses (`DecoratorFigure`, `PertFigure`, etc.), and an aggregate class (`DecoratorFigure`) with its component (`Figure`).

9.3 Categories and Factorisation

The categorisation of nodes, and the accompanying metrics, provide insights into the levels of factorisation within the systems under study and may help compare them in terms of design maturity. From the categorisation of candidate nodes and the results for JHotDraw and JReversePro, we observe:

- The hierarchical representations of candidate features, in Figures 12 and 15, provide insights into the factorisation landscape of a system, to focus on features that are most interesting, i.e., *Ad Hoc* features that are *not* descendants of deliberate nodes.
- Notwithstanding false positive *Ad Hoc* candidate functional features and `ADHOC` nodes that descend from deliberate nodes (which are readily identifiable), *most* of the remaining `ADHOC` nodes (7 out of 9 for JHotDraw and 9 out of 15 for JReversePro) are interesting abstractions missed by developers, which may warrant refactorings into well-packaged features.
- Some counterintuitive observations about the *quality* of factorisation lead us to argue that, in the lower levels of a class hierarchy, a *good* factorisation is a combination of (1) a small fraction of the behaviour of the root of the sub-hierarchy redefined by its subclasses but (2) subclasses define *little* additional behaviour to the root one. If there is a *too small* coverage, i.e., no/little additional behaviour in the subclasses, then we question the need for separate subclasses. If there is a *too big* coverage, i.e., extra behaviour in the subclasses, then the subclasses could better be in their own sub-hierarchies.

9.4 Categories and Design Patterns

Design patterns, by their very definitions, use delegation and inheritance abundantly. Our categories and algorithms, unsurprisingly, show that JHotDraw, the GUI development framework meant as a *case study in design patterns*, is the most mature of the five systems.

On the contrary, JReversePro, a research prototype for reverse engineering Java bytecode, developed by two researchers working in language design, is the least mature of the studied systems.

Our categories and metrics reflect the presence or absence of design patterns. However, contrary to work that only attempts to identify occurrences of some design patterns in systems, e.g., [63], our approach distinguishes between "good" and "bad" designs and, therefore, under/over uses of design patterns.

Indeed, while a moderate and targeted use of design patterns improves design quality, using *no* design patterns or using design patterns *unnecessarily* decreases design quality. Our algorithms and metrics will identify such cases by reporting features that could be factorised (missing design patterns) and showing overlapping features (unnecessary design patterns).

9.5 Functional Features and Galois Lattices

We defined a functional feature as a set of program elements distributed among a number of classes, which together implement a particular function, as shown in Figure 6a. Notwithstanding the complexity and computational intractability of a structure-sensitive incidence relationship, we proposed a simplified incidence relationship, which associates each class with the set of methods and attributes defined in the class and its descendants, which we call the *reverse inheritance* incidence relationship.

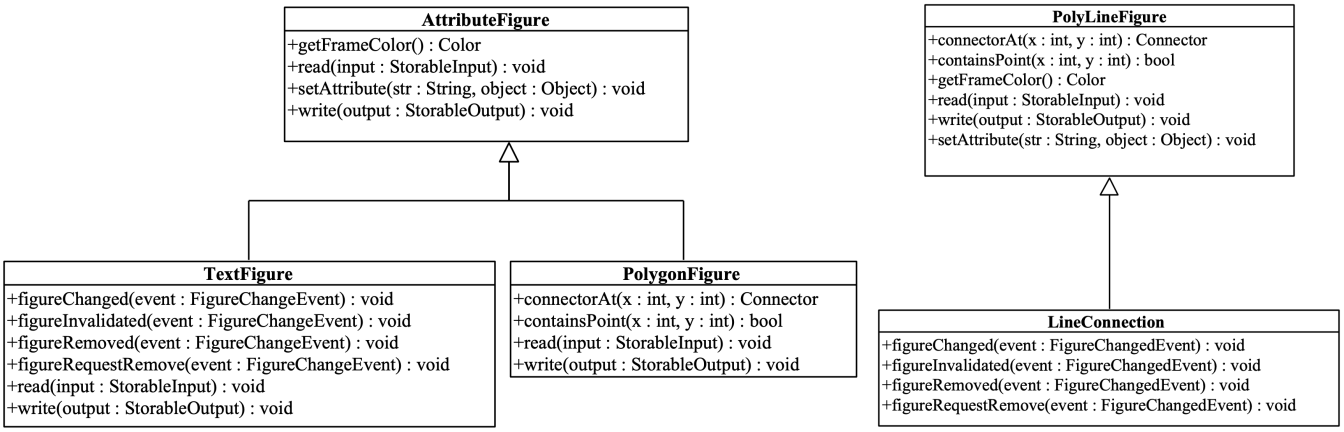
In earlier work, we used Galois lattices on class interfaces to identify optimal factorisations of class hierarchies [54]. The example of Figure 14 shows a situation where our current algorithm discovers a candidate feature consisting of methods occurring directly in the classes of the extent (`DrawApplet` and `DrawApplication`), i.e., without resorting to reverse inheritance.

Yet, we *do* need reverse inheritance because the program elements that constitute a legitimate feature will often be spread over several classes. Figure 19 illustrates the situation, showing methods corresponding to a *subset* of the intent, which is a feature implementing a concept of a *connection*, as the locus of linkage between figures that reacts to figure change events. On the right, Figure 19 shows the connection-point feature from `PolyLineFigure` and the changeability feature from `LineConnection`. On the left, the same methods are spread over three classes, two of which are *not* hierarch vically related. We need the reverse-inheritance incidence relationship because our algorithm only considers inherited methods if they belong to different class sub-hierarchies.

9.6 Functional Features and Code Clones

We chose to define the functional features of interest as similar to *subjects* [1] rather than *aspects* [2] (see Section 1) because we are interested in domain-related features, which is also the reason for not analysing method bodies.

However, aspects, either explicitly using some technology like AspectJ or implicitly, often in the form of code clones, could help improve the quality of the detected functional features or find new ones. For example, Peng et al. [64] used code clones to detect features in source code. Yet, their



(a) One occurrence of the intent of Node 5 (JHotDraw). (b) A second occurrence.

Fig. 19. The concept of *connection* is spread over several classes in JHotDraw

features of interest are implementation-related, not domain-related, e.g., `BankOpAuth` for operator authentication.

We are interested in domain-related functional features, for example, the feature concerned with moving figures in JHotDraw. Therefore, we did not consider using code clones but will explore in future work combining our idioms and algorithms with code-clone detection to assess whether this combination helps in improving the discovered functional features and–or detecting new functional features.

9.7 Functional Features Understanding

Algorithms 2 and 3 in this article (1) provide a cartography of the various functional features of a system and (2) identify additional opportunities for factorisation, refactoring, and reuse—the *Ad Hoc* candidate functional feature.

- Those that are descendants of deliberate (candidate) features are *most likely not* interesting, the result of the combinatorics of associating sets of classes to their shared set of methods/attributes; inherent to FCA.
- Those that are *not* descendants of deliberate (candidate) features embody interesting abstractions that have been missed by developers.

Thus, our algorithms could contribute to software development and evolution by:

- Identifying and displaying deliberate, recognised classes and interfaces that embody reusable functional features, which are identified by Algorithm 2, and which are identified as *anchor types* by Algorithm 3.
- Discovering and displaying independent *Ad Hoc* candidate functional features, which are novel features unrecognised by developers.
- Displaying the candidate features, i.e., highlighting the program elements composing it, when the classes or interfaces that participate in it are selected, as shown in Figure 11, along with metrics.
- Displaying the hierarchy of deliberate candidate features induced by a class or interface upon its selection, similar to the hierarchies (e.g., in Figures 12 and 15), but hiding *Ad Hoc* nodes.
- Similarly, the program elements composing *Ad Hoc* candidate functional features upon the selection of one of these elements, similar to Figure 14.

Such a tool would have several advantages:

- Providing a synthesized *indicator*—if not a metric—of the quality of factorisation in a system.
- Suggesting novel features through the *Ad Hoc* candidates, embodying interesting abstractions that may warrant refactoring.
- Requiring no developers' input thanks to our algorithms relying on *explicit* programming language semantics and formal concept analysis; as opposed to *implicit* semantics, as embodied in variable names and comments.

10 Conclusion and Future Work

We presented algorithms to discover candidate functional features in legacy OO systems. We explored three hypotheses about how developers, without aspect-oriented programming abstractions, would implement functional features in a system.

We developed algorithms that rely *solely* on types (classes and interfaces) and method signatures to discover deliberate and ad-hoc candidate functional features and applied them on five open-source systems and observed that (1) multiple inheritance and delegation were overwhelmingly used to compose recognised functional features; (2) our algorithms discovered ad-hoc functional features as well as some deliberate features not following the multiple inheritance/delegation idioms; (3) our algorithms are complementary to cartography systems and discover opportunities for refactoring and reuse; and, (4) the outputs of our algorithms provide *indicators* of the design maturity of the systems under study.

In future works, we want to apply our approach to more, diverse legacy systems to study their generalisability. We also want to use objective measures to assess and compare the discovered candidate features. We will validate ad-hoc candidate features *a-posteriori* by studying subsequent

versions of some systems. We will also study the use of code clones to improve the quality of the functional features and–or detect new functional features. We will study the evolution of features and their relationships with changes and faults. We will also consider evaluating the applicability of our approach for dynamically-typed programming languages, with their implicit dependencies [65], and non-object-oriented programming languages, e.g., typed functional programming languages like TypeScript.

ACKNOWLEDGMENTS

This work was partly funded by Canada's NSERC Discovery Grant and Research Chair programs.

REFERENCES

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

[2] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda, and A. Mendhekar, "Aspect-oriented programming," in *Proc. of ECOOP'97*. Springer, 1997, pp. 220–242.

[3] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *Proceedings - IEEE Workshop on Program Comprehension*, vol. 2000-January, 2000, pp. 241–249.

[4] J. Rubin and M. Chechik, "A Survey of Feature Location Techniques," in *Domain Engineering: Product Lines, Languages, and Conceptual Models*. Springer-Verlag, 2013.

[5] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code : a taxonomy and survey," *JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS*, vol. 25, no. November 2011, pp. 53–95, 2013.

[6] G. Szőke, C. Nagy, R. Ferenc, and T. Gyimóthy, "A case study of refactoring large-scale industrial systems to efficiently improve source code quality," in *Computational Science and Its Applications – ICCSA 2014*, B. Murgante, S. Misra, A. M. A. C. Rocha, C. Torre, J. G. Rocha, M. I. Falcão, D. Taniar, B. O. Apduhan, and O. Gervasi, Eds. Cham: Springer International Publishing, 2014, pp. 524–540.

[7] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[8] Denys Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *Transactions on Software Engineering (TSE)*, vol. 33, no. 6, pp. 420–432, June 2007, 14 pages.

[9] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program understanding and the concept assignment problem," *Commun. ACM*, vol. 37, no. 5, pp. 72–82, 1994.

[10] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, March 2003.

[11] M. T. Valente, I. C. Society, V. Borges, and L. Passos, "A Semi-Automatic Approach for Extracting Software Product Lines," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 737–754, 2012.

[12] L. Arcega, J. Font, Ø. Haugen, and C. Cetina, *Feature Location Through the Combination of Run-Time Architecture Models and Information Retrieval*, J. Grabowski and S. Herbold, Eds. Springer, 2016, vol. 2.

[13] P. W. McBurney, C. Liu, and C. McMillan, "Automated feature discovery via sentence selection and source code summarization," *J. Softw. Evol. Process*, vol. 28, no. 2, pp. 120–145, Feb. 2016.

[14] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "Sniafl: Towards a static noninteractive approach to feature location," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 2, pp. 195–226, Apr. 2006.

[15] S. Breu and J. Krinke, "Aspect mining using event traces," in *Proceedings of the 2004 Conference on Automated Software Engineering*, 2004.

[16] P. Tonella and M. Ceccato, "Aspect mining through the formal concept analysis of execution traces," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, Nov. 2004, pp. 112–121.

[17] C. Nagy, L. Meurice, and A. Cleve, "Where Was This SQL Query Executed ? A Static Concept Location Approach," in *SANER*. Montreal, Canada: IEEE Computer Society Press, 2015, pp. 580–584.

[18] A. Shatnawi, H. Mili, G. El-Boussaidi, A. Boubaker, Y. Guéhéneuc, N. Moha, J. Privat, and M. Abdellatif, "Analyzing program dependencies in java EE applications," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 64–74.

[19] M. Revelle and D. Poshyvanyk, "An exploratory study on assessing feature location techniques," in *Proceedings of 17th IEEE International Conference on Program Comprehension (ICPC2009)*, Vancouver, BC, Canada, 2009.

[20] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 987–1007, 2021.

[21] S. A. Ajila, "Reusing base product features to develop product line architecture," in *16th International Conference on Information Reuse and Integration*. IEEE, 2005, pp. 288–293.

[22] K. C. Kang, M. Kim, J. Lee, and B. Kim, "Feature-oriented re-engineering of legacy systems into product line assets – a case study," in *9th International Conference on Software Product Lines*, 2005, pp. 45–56.

[23] K. Kim, H. Kim, and W. Kim, "Building software product line from the legacy systems "experience in the digital audio and video domain"," in *11th International Conference on Software Product Lines*, 2007, pp. 171–180.

[24] T. A. Wiggerts, "Using clustering algorithms in legacy systems remodularization," in *4th Working Conference on Reverse Engineering*, 1997, pp. 33–43.

[25] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," in *24th International Conference on Web Services*, 2017, pp. 524–531.

[26] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage, "How do professionals perceive legacy systems and software modernization?" in *36th Interna-*

tional Conference on Software Engineering. Association for Computing Machinery, 2014, pp. 36–47.

[27] C. Chiang and C. Bayrak, "Legacy software modernization," in *9th International Conference on Systems Man, and Cybernetics*. IEEE, 2006, pp. 1304–1309.

[28] M. von Detten, M. C. Platenius, and S. Becker, "Reengineering component-based software systems with Archimetrix," *Software & Systems Modeling*, vol. 13, no. 4, pp. 1239–1268, 2014.

[29] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kröger, "Microservice decomposition via static and dynamic analysis of the monolith," in *17th International Conference on Software Architecture (Companion)*. IEEE, 2020, pp. 9–16.

[30] M. Abdellatif, A. Shatnawi, H. Mili, N. Moha, G. El Boussaidi, G. Hecht, J. Privat, and Y.-G. Guéhéneuc, "A taxonomy of service identification approaches for legacy software systems modernization," *Journal of Systems and Software*, vol. 173, p. 110868, 2021.

[31] B. Dagenais and H. Mili, "Slicing functional aspects out of legacy applications," 2021, arXiv #2109.12076.

[32] A. ElKharraz, H. Mili, and P. Valtchev, "Mining functional aspects from legacy code," in *Proc. of ICTAI'08*. IEEE Comp. Soc., 2008, pp. 403–412.

[33] R. Koschke and J. Quante, "On dynamic feature location," *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering SE - ASE '05*, pp. 86–95, 2005.

[34] Z. Zhang, H. Yang, and W. Chu, "Extracting Reusable Object Oriented Legacy Code Segments for Service Integration," *6th International Conference on Quality Software (QSIC'06)*, pp. 385–392, 2006.

[35] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, June 2007, pp. 37–48.

[36] R. Al-Msie'Deen, A.-D. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "Feature location in a collection of software product variants using formal concept analysis," *Lecture Notes in Computer Science*, vol. 7925 LNCS, pp. 302–307, 2013.

[37] H. Eyal-salman, A.-D. Seriai, and C. Dony, "Feature Location in a Collection of Product Variants : Combining Information Retrieval and Hierarchical Clustering," in *The 26th International Conference on SoftwareSoftware Engineering and Knowledge Engineering (SEKE 2014)*, Vancouver, Canada, 2014, pp. 426–430.

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

[39] R. Godin and P. Valtchev, "Formal concept analysis-based normal forms for class hierarchy design in oo software development," in *FCA: Foundations and Applications*. Springer, 2005, ch. 16, pp. 304–323.

[40] R. Godin, H. Mili, G. Mineau, R. Missaoui, A. Arfi, and T. Chau, "Design of Class Hierarchies Based on Concept (Galois) Lattices," *TAPOS*, vol. 4, no. 2, pp. 117–134, 1998.

[41] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 353–363.

[42] M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S.-C. Cheung, and Z. Su, "Exploring and exploiting the correlations between bug-inducing and bug-fixing commits," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 326–337.

[43] A.-J. Molnar and S. Motogna, "Long-term evaluation of technical debt in open-source software," in *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM '20. New York, NY, USA: Association for Computing Machinery, 2020.

[44] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference*, 2010, pp. 336–345.

[45] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, jan 2002.

[46] J. Brant, "Hotdraw," Master's thesis, University of Illinois, Urbana, 1995, master's Thesis.

[47] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 516–527.

[48] Y. Wu, D. Zou, S. Dou, S. Yang, W. Yang, F. Cheng, H. Liang, and H. Jin, "Scdetector: Software functional clone detection based on semantic tokens analysis," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 821–833.

[49] Y.-G. Guéhéneuc and H. Albin-Amiot, "Recovering binary class relationships: Putting icing on the uml cake," *SIGPLAN Not.*, vol. 39, no. 10, pp. 301–314, oct 2004.

[50] G. Curry, L. Baer, D. Lipkie, and B. Lee, "Traits: An approach to multiple-inheritance subclassing," in *Proceedings of the SIGOA Conference on Office Information Systems*. New York, NY, USA: Association for Computing Machinery, 1982, pp. 1–9.

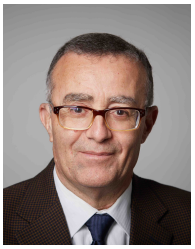
[51] B. Carré and J.-M. Geib, "The point of view notion for multiple inheritance," in *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA/ECOOP '90. New York, NY, USA: Association for Computing Machinery, 1990, pp. 312–321.

[52] Y. Caseau, "Efficient handling of multiple inheritance hierarchies," *SIGPLAN Not.*, vol. 28, no. 10, pp. 271–287, oct 1993.

[53] A. Sabané, Y.-G. Guéhéneuc, V. Arnaoudova, and G. Antoniol, "Fragile base-class problem, problem?" *Empirical Softw. Eng.*, vol. 22, no. 5, pp. 2612–2657, Oct. 2017.

[54] R. Godin and H. Mili, "Building and maintaining analysis-level class hierarchies using Galois lattices," in *Proceedings of OOPSLA '93*. New York, NY, USA:

ACM, 1993, pp. 394–410.
[55] P. Valtchev, R. Missaoui, and R. Godin, "Formal concept analysis for knowledge discovery and data mining: The new challenges," in *Concept Lattices, Second International Conference on Formal Concept Analysis, ICFCA 2004, Sydney, Australia, February 23-26, 2004, Proceedings*, ser. Lecture Notes in Computer Science, P. W. Eklund, Ed., vol. 2961. Springer, 2004, pp. 352–371.
[56] H. Mili, A. Mili, S. Yacoub, and E. Addy, *Reuse-based software engineering: techniques, organizations, and controls*. Wiley, 2001.
[57] L. Briand, J. Daly, and J. Wust, "A unified framework for cohesion measurement in object-oriented systems," in *Proceedings Fourth International Software Metrics Symposium*, 1997, pp. 43–53.
[58] ——, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91–121, 1999.
[59] G. El Boussaidi and H. Mili, "Understanding design patterns, what is the problem?" *Software - Practice and Experience*, vol. 42, pp. 1495–1529, 2012.
[60] A. B. Belle, G. El Boussaidi, C. Desrosiers, S. Kpodjedo, and H. Mili, "The layered architecture recovery as a quadratic assignment problem." in *the 9th European Conference on Software Architecture (ECSA), Dubrovnik, Croatia, LNCS*, 2015, pp. 339–354.
[61] J. Vedurada and V. K. Nandivada, "Identifying refactoring opportunities for replacing type code with subclass and state," *ACM Program. Lang.OOPSLA*, vol. 2, 2018.
[62] E. da Silva Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1044–1062, 2017.
[63] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A multilayered framework for design pattern identification," *Transactions on Software Engineering (TSE)*, vol. 34, no. 5, pp. 667–684, September 2008, 18 pages.
[64] X. Peng, Z. Xing, X. Tan, Y. Yu, and W. Zhao, "Improving feature location using structural similarity and iterative graph mapping," *Journal of Systems and Software*, vol. 86, no. 3, pp. 664–676, 2013.
[65] W. Jin, Y. Cai, R. Kazman, G. Zhang, Q. Zheng, and T. Liu, "Exploring the architectural impact of possible dependencies in python software," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 758–770.



Hafedh Mili is a Professor at Université du Québec à Montréal. He obtained his Ph.D. in Computer Science (Artificial Intelligence) from George Washington University in 1988. His research interests cover many areas of software engineering, from the early phases of software development (business modelling, representation and classification of business processes, generation of software models from business models) to the reengineering of legacy applications to adapt them to modern deployment architectures. He has consulted widely on business rules and co-authored Agile Business Rule Development with J. Boyer (Springer, 2011).

Imen Benzarti received her PhD in computer science from Université du Québec à Montréal, Canada(2020). She is a faculty member at École de Technologie Supérieure, Université du Québec, Canada. She was a postdoctoral fellow at the Université du Québec à Montréal (2021). She is a member of the LATECE research lab at Université du Québec à Montréal. Her research interests include models-driven engineering, human-centric software engineering and empirical software engineering.

Amel Elkharraz is a teaching professor at Collège Bois de Boulogne. She obtained her Ph.D. in computer sciences from the Université de Montréal. She also has a 3rd cycle thesis from Mohammed V University in Rabat and was a professor at the FST Faculty of Science and Technology in Tangier (Morocco).

Ghizlane El Boussaidi received the PhD degree in software engineering from the Université de Montréal, Canada, in 2010. She is a Full Professor of Software Engineering at École de Technologie Supérieure, Montreal, Canada, and a member of the LATECE research lab, at the University of Quebec in Montreal. She participated in and led several research projects funded by Canadian funding agencies and industry partners. Her research interests include software architecture, software re-engineering and modernization, model-driven development and safety-critical systems.

Yann-Gaël Guéhéneuc is full professor at the Department of Computer Science and Software Engineering of Concordia University, where he leads the Ptidej team on evaluating and enhancing the quality of the software systems, focusing on the Internet of Things and researching new theories, methods, and tools to understand, evaluate, and improve the development, release, testing, and security of such systems. He was awarded the NSERC Research Chair Tier I on Empirical Software Engineering for the IoT in 2018. He received a Ph.D. in Software Engineering from the University of Nantes, France, under Professor Pierre Cointe's supervision in 2003. He has published papers in international conferences and journals, including IEEE TSE, Springer EMSE, ACM/IEEE ICSE, IEEE ICSME, and IEEE SANER. He was the program co-chair and general chair of several events, including IEEE ICPC'20 and '19, SANER'15, APSEC'14, and IEEE ICSM'13.

Petko Valtchev received his Ph.D. in Computer Science from J. Fourier University, Grenoble, in 1999 (thesis prepared at INRIA). He is an Associate professor with the Department of Computer Science of Université du Québec à Montréal (UQÀM) and chairs their Center for AI Research. He has contributed several original methods for formal concept analysis and pattern mining and researched their practical application to data from a wide range of fields such as software engineering, telecommunication and networking, manufacturing, agriculture, healthcare, etc. His current research is about knowledge discovery from structured data such as sequences, trees and graphs with rich sources of domain knowledge.