# Reconsidering Classes in Procedural Object-Oriented Code

Muhammad Usman BHATTI[1] Stéphane DUCASSE[2] Marianne HUCHARD[3]
[1]*CRI - Univ. Paris 1 Sorbonne, France*
[2]*INRIA - Lille Nord Europe, France*
[3]*LIRMM - Univ. Montpellier 2, France*
*muhammad.bhatti@malix.univ-paris1.fr, stephane.ducasse@inria.fr, marianne.huchard@lirmm.fr*

## Abstract

*Object-oriented software may show signs of procedural thinking because of lack of design or due to design erosion over a period of time. We refer to such a software as procedural object-oriented code. Huge classes, scarce class hierarchies and absence of classes for domain entities are hallmarks of procedural object-oriented code. Due to huge investments in such systems, software restructuring becomes necessary. To support code modularization, it is important to identify useful domain abstractions. In this paper, we present a tool-assisted technique to identify useful abstractions and class hierarchies in procedural object-oriented code. During this task, principal classes (draft classes) are identified. Afterwards, composition and association relationships are inferred for principal classes. Lastly, Formal Concept Analysis (FCA) is used to analyze hierarchical relationships between methods and attributes within principal classes. We validated our approach on several case studies and report our results on an industrial case.*

## 1 Introduction

Software design and software quality are often victims of constrained budgetary resources. The use of state of the art languages such as C# and Java cannot replace the need for an upfront modular design. Moreover, software design erodes with evolution and shows signs of procedural thinking [9]. We coined the term *procedural object-oriented* code (POC) for the code which shows signs of absence or erosion of an overall object-oriented design but nonetheless has been developed using state of the art object-oriented languages. Procedural object-oriented code consists of *half-baked* objects: objects encompassing other objects. The hallmark of POC is the presence of huge classes and the absence of class hierarchies. In addition, certain domain entities are not represented in precise classes but scattered in other classes. Software reengineering and restructuring is beneficial for such code because of the domain knowledge represented by the code and costs of its development.

Formal Concept Analysis (FCA) is a mathematical technique to discover useful hierarchical groupings of *objects* having similar *attributes* [14]. This technique has been successfully applied to obtain useful groupings of functions and global variables in procedural code to place them in same object-oriented classes [23, 24]. This technique has also been applied to analyze and restructure class hierarchies [10, 26, 2]. However, the application of FCA for procedural object-oriented code may need further refinements because of the following reasons: First, methods attached to current classes may be misplaced. Second, huge lattices obtained from procedural object-oriented code may obstruct the analyzer to find useful abstractions amongst the existing classes. In case of reduction of context to find meaningful information, it is necessary to find the pertinent methods within the existing classes that operate upon particular attributes. Third, this type of code may contain interesting traces of object-oriented language constructs, which may be employed to enhance the information to generate class hierarchy. Hence it is not enough to assign procedures to types to get classes as in traditional object identification from procedural code [23, 24, 27].

In this paper, we present a semi-automatic, tool-assisted approach for restructuring object-oriented software showing signs of absence of object-oriented design. We start our discussion by the description of some of the design problems or "code smells" related to POC for quick discovery of restructuring opportunities. Afterwards, we define our approach in four steps:

1. To decompose large classes into smaller cohesive pieces, methods present in the code and user-defined types they operate upon are grouped in *principal classes* following certain rules.

2. An architectural abstraction for principal classes is obtained to understand the interaction and composition of principal classes amongst themselves.

3. Hierarchical abstractions for the methods and attributes of each of the principal classes are obtained by analyzing their accesses to the individual elements of user-defined types.

4. Scattered code related to global enumerated types is identified and refactored into new methods. These methods are then added to the user-specified principal class.

This paper is organized as follows: Section 2 presents some of the commonly occurring design problems in POC. Section 3 describes our restructuring approach for POC. In Section 4, we present an evaluation of our approach on an industrial software system. Section 5 discusses the results of our approach and its limitations. Section 6 work presents the state of the art on the presented work and Section 7 concludes the paper and presents perspectives of the future work.

## 2 Procedural Object-Oriented Code Design Problems

Object-oriented legacy systems often face similar problem as procedural legacy systems [9]. They contain duplicated code and logic, misplaced logic, and incomplete abstractions. To make a clear distinction between good object-oriented code and code presenting such defects we coined the term procedural object-oriented code (POC).

Code smells often indicate the presence of design problems. Code smells can be used to apply possible remedy (refactoring) to the problem and help developer ascertain information related to problematic code [12]. We provide a list of code smells related to procedural object-oriented code and revealing object-oriented design absence. Although, some of the code smells that we present are related to those presented in [12], the code smells presented here are provided to describe them in the context of lack of design.

### 2.1 Half-Baked Objects

Object-oriented design principle advocates the decomposition of various objects into types and sub-types relationship. These types are then composed through structural and behavioral composition to define the business logic. In POC, however, such principles are not consistently applied leading to lack of inheritance hierarchies and lack of object-oriented decomposition.

**Absence of Class Hierarchies.** A very typical POC symptom code is the absence or scarcity of class hierarchies in the software. This naturally origins from the omission of analysis process whereby types and their subtypes for domain entities are identified. Consequently, the classes represent huge structures encapsulating all types and subtypes, which are present in a single class. In such scenarios when no entity abstracts common functionality, duplicate logic for common tasks appears in methods.

**Lack of Decomposition - Missing Abstractions.** If the design decomposition is not refined to a certain level, it results in the absence of important types from the system, resulting in their scattered manifestation in other half-baked objects. Code associated to these missing abstractions makes the overall reuse, change and evolution of the system cumbersome.

### 2.2 Global Enumerated Types and States

In absence of encapsulation and abstraction provided by classes in a classical object-oriented system, global enumerated types are used to hold the i) states, and ii) types of objects. These global enumerated types are used in a large number of methods according to the functional program flow, hence leading to their scattering and tangling throughout the code. This scattering happens mostly in the form of conditionals to test the states of program entities represented by the enumerated types, and switch statements to perform related operations according to the state and type information. Enumerated types, when used with conditionals, provide the code smell for applying Transform Conditional to Polymorphism, as mentioned in [12, 9]. Note that in our context this is more difficult since the code is spread over multiple classes.

### 2.3 Non-abstracted Template Behavior

A lack of hierarchical structure may actually lead to a lack of code reuse, a leverage obtained by abstracting the common code in superclass of the hierarchy. This leads to duplication of ”template code”. Template code manifests itself either in the form of common calls or method duplication.

**Common Calls.** Absence of a parent class for similar types results in the presence of non-abstracted, common calls in methods that perform common processes. Common calls pattern is manifested by invocation of similar methods from their client methods and is depicted in Figure 1. In this case, methods B, D, F invoke methods A and C. These cloned calls show the presence of non-abstracted template behavior that can be refactored into a common superclass. Hence, such pattern is identified to create a common superclass for the methods implementing common calls. The client methods may be present in same classes or different classes.

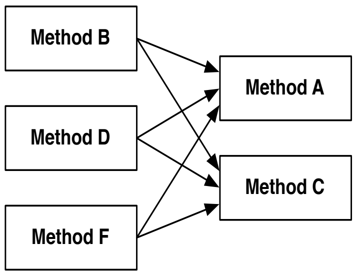

**Figure 1. Cloned Calls - Missing Template Behavior**

**Method Duplication.** Method duplication refers to different methods where each of them implements some common functionality in addition to their respective specialization. This form is different from common calls because the common logic is in the form of a set of common instructions working on input variables. These methods are commonly implemented using similar names. This may be overlooked due to the resemblance to method overloading.

## 3 Object Identification

In this section, we describe our object identification process which supports the discovery of useful abstractions in procedural object-oriented code. As described earlier, based on type/class usage we identify *principal classes*. In the second phase, we identify composition relationships between principal classes based on *common creation* pattern, *i.e.,* types that are created together. The last phase consists of finding hierarchy of attributes and methods assigned to principal class. We also perform an analysis of enumerated types to associate each of them with their associated principal class. The overall approach is presented in Figure 2. However, before we move further to describe the internals of our approach, we describe basic definitions of Formal Concept Analysis (FCA).

**FCA Basic Definitions.** Concept Analysis provides a way to identify sensible grouping of *objects* that have common *attributes* [14]. A context is a triple $C = (O, A, R)$, where $O$ and $A$ are finite sets (the objects and attributes, respectively), and $R$ is a binary relation between $O$ and $A$. A concept is a pair of sets: a set of covered objects (the extent) and a set of shared attributes (the intent) $(X, Y)$ with $X \subseteq O$, $Y \subseteq A$ and $X = \{o \in O | \forall y \in Y, (o, y) \in R\}$, $Y = \{a \in A | \forall x \in X, (x, a) \in R\}$.

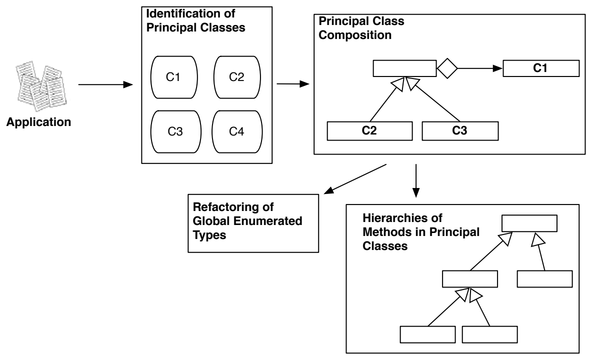
Application
Identification of Principal Classes
C1
C2
C3
C4
Principal Class Composition
C1
C2
C3
Refactoring of Global Enumerated Types
Hierarchies of Methods in Principal Classes

**Figure 2. Overall Approach**

### 3.1 Identification of Principal Classes

The first step consists of the identification of a cohesive set of methods grouped by type usage. Types are the user-defined classes defining a set of atomic (primitive) attributes. Read or write access to an atomic attribute of a type is considered as the read or write access to the type defining the attribute. We term the group of methods and the type that these methods access and modify as a *principal class*. Note that we only consider end-user types as potential target for principal classes, primitive types are not considered [23]. The following rules define method groupings as illustrated in Figure 3 using principle of class cohesion [11].

- All methods that exclusively write to a particular variable of a given type are associated to the principal class for this type. In Figure 3 method m1 is associated with principal class 1 because it writes to variable e1 of type T1.
- Similarly all methods that exclusively read from a variable of a given type are associated to the principal class of that type. Class 5 in Figure 3 reads from the variable e6 of Type T6.
- In case a method writes to two types, the method is marked as a candidate for decomposition using slicing [13]. Slicing helps segregate instructions working on different variables. Method 4 in Figure 3 is a candidate for slicing. In this case, the method is decomposed into two and each new method is assigned to its corresponding principal class.
- In case a method reads to two types, it is associated to the type with most read number.
- When a method $m$ doesn't read or write a type but calls another method $n$ in principal class PC1, then the method $m$ is associated to PC1.

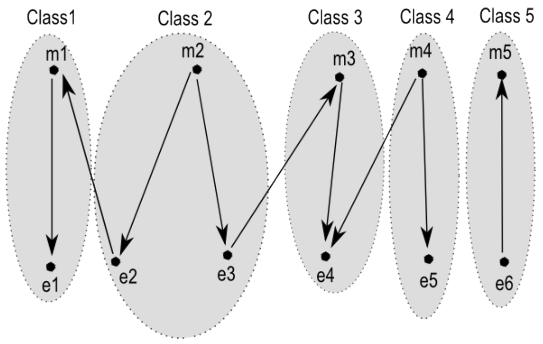

**Figure 3. Principal Class Identification**

This step identifies groups of cohesive entities and methods that make up candidates for principal classes.

During this step, we keep track of the dependencies amongst the identified principal classes. This is achieved by keeping track of the read information of other principal classes in all methods of a principal class. For example, if methods of a principal class PC1 access variable of type t2 and t3, which are associated to PC2 and PC3, this information is kept. Such information is later used to identify degree of associations amongst principal classes.

## 3.2 Principal Class Compositions

Once principal classes are identified, we identify composition relationships among principal classes. For this purpose, we identify *Create-Create* pattern in code.

The pattern searches for all methods belonging to a principal class that create a variable of its own type as well as a variable of another type. For example, a method belonging to principal class PC1 initializes a variable of its associated type and calls another method which creates instances of another type PC2. Thus, a composition link is created such that PC1 is composed of PC2 as depicted on the left of Figure 4. This pattern is also searched in methods marked for slicing because this may provide useful information about principal class composition.

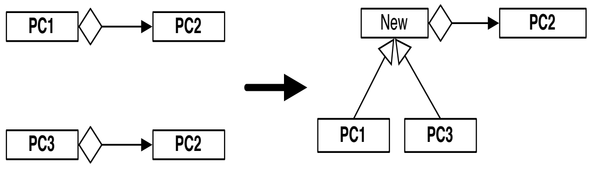

**Figure 4. Principal Class Compositions**

If two principal classes PC1 and PC3 are found to be composed of a third principal class PC2, a new parent class is created which composes itself with PC2. PC1 and PC3 then inherit from the newly created class as demonstrated in Figure 4. So for finding common compositions, concept lattices are created for composition relationship. For this purpose, we define the FCA context as follows:

- O = All Principal classes
- A = All Principal classes
- R = $pc1$ creates $pc2$

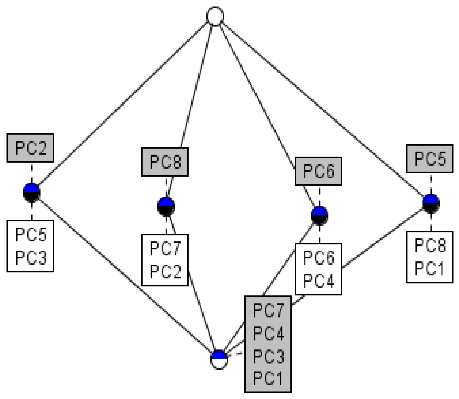

**Figure 5. Common Compositions**

Figure 5 depicts an example of lattice displaying common compositions (attributes are in grey background, PC represents a principal class). For example, the figure shows that principal classes 5 and 3 commonly create class 2. Moreover, principal classes 1, 3, 4, 7 are not created by any other principal class. So, four superclasses are created containing variables of types of principal classes 2, 8, 6, 5.

## 3.3 Hierarchical Method-Attribute Relationship

Now that principal classes and their composition links are inferred, we identify hierarchical abstractions present *within* a principal class. For this purpose, information regarding the types and methods associated to each principal class is studied. For this, access patterns are observed for methods accessing the attributes of the types in their principal classes. This offers a possible decomposition of principal class into subclasses, with common attributes appearing in the parent class.

We extract using FCA three different views (named fundamental, interaction and associations) to support the possible modularization within principal classes. These views help us to simplify the structure of the resulting concept lattices for principal classes. The views can be combined (*i.e.,* a single lattice is generated) for smaller principal classes but for larger ones, extraction of useful modularization in one combined view becomes cumbersome.

**Fundamental View.** For generating fundamental view lattices, we consider individual (class) attributes of the user-defined types and methods accessing these attributes. When this information is fed into FCA and lattices are generated, the lattices provide hierarchy of methods using the attributes in principal classes. For the fundamental view, we define the FCA context as follows:

- O = All Methods within principal class
- A = Attributes of the user-defined type associated to principal class
- R = Method $m$ reads or modifies an attribute of its associated type

This context has generally been used to search for object-oriented abstractions in procedural code [23, 24, 27]. In our particular case, it helps finding possible attributes and methods of subclasses. In our approach, the context is only the concerning attributes and methods, hence the results are less complicated.

Figure 6 presents an example of a fundamental view of a principal class containing five methods and five attributes (attributes are in grey background denoted by $a$, methods are denoted by $m$). It presents two disjoint sets of classes to reconstitute methods of the principal class represented in the view: The right-hand side concepts can be restructured as hierarchy of classes with a superclass a1:m1 and a subclass a5:m5), and the left-hand side concepts propose a hierarchy with a superclass (a2:m2) and two subclasses (a3:m3,a4:m4).

**Common Interactions View.** The common interaction view helps understanding all of the method invocation of the methods in a principal class. This helps understanding the interaction of various methods present within the principal classes for their possible categorization as interface methods or functionality providers [1]. In addition, this supports the identification of template behavior regarding common method calls. That is, methods that call common methods are clustered together.

For this view, we define the FCA context as follows:

- O = All Methods within principal class
- A = Method invocations
- R = method $m$ calls method $n$

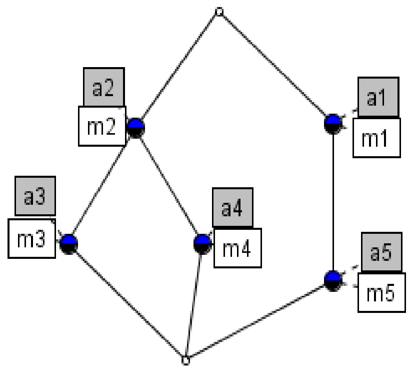

**Figure 6. Fundamental View of Principal Classes**

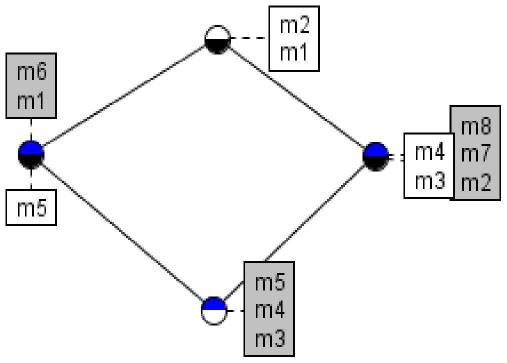

**Figure 7. Common Calls View of Principal Classes**

Figure 7 presents the interaction view for the principal class presented in Figure 6. The figure demonstrates that method m5 calls m1 and m6. Moreover, the concept containing methods m3 and m4 in its extent indicates the existence of common call pattern. It can also be inferred that methods m1 and m2 within this principal class do not invoke any methods inside the principal class.

**Associations View.** In this view, we use the information retrieved while assigning methods to principal classes. This information, which we term as *associations*, represents accesses to other principal classes from the methods of the current principal class. For this view, we define the FCA context as follows:

- O = All methods within a current principal class
- A = Attributes of the type, and other principal classes (except those linked by composition)
- R = $m$ accesses an attribute of its type or accesses principal class $pc$

This view shows the degree of usage of a principal class within methods of the current principal and hence its place within the hierarchy of the principal class. Principal classes for which current principal class already has a composition link are excluded from this view. For example, the current principal class PC1 has a composition link to PC2, then association links to PC2 are excluded from the association view lattice.

In addition, this view helps to segregate the functionality implemented by methods of a principal class. For example, a principal class may have two sets of methods: methods solely working on its attributes and methods which interact with other principal classes. In some cases, these two may represent new "candidate" classes. We illustrate this by an example: Consider the lattice presented in Figure 8. This lattice augments the fundamental view presented in Figure 6. It demonstrates the usage pattern of two principal classes PC1 and PC2 within the methods of the current principal class. It is interesting to see that PC2 is commonly used in all the methods of the principal class, while PC1 is used by another subset. The reengineer can interpret this usage pattern according to her understanding. A new class may be created to include PC2 from which the two disjoint classes created for the current principal class can be inherited.

## 3.4 The case of Enumerated Types

In some occasions, enumerated types are used in conjunction with conditional statements to replace object-oriented abstractions [12, 9]. When such a practice occurs, the reengineer is requested to apply either Transform conditional to polymorphism [9] or Rewrite type code with state/strategy [12]. However, these solutions hypothesize that these enumerated types are encapsulated in methods. In POC, such enumerated types are used as global variables in different methods and the logic associated to a particular enumerated type is scattered in these methods. Hence, the first step is to decern the location of global enumerated types and refactor these enumerated types in methods so that the proposed refactorings can be applied [12, 9]. The refactored methods are then associated to the principal class specified by reengineer. For this purpose, a simple tool is developed that inspects usage of enumerated types along with conditional statements for guiding the identification of such patterns. The idea is to ease location identification of places where these enumerated types are used and refactor their code in a new method. This method is then associated to one of identified principal classes.

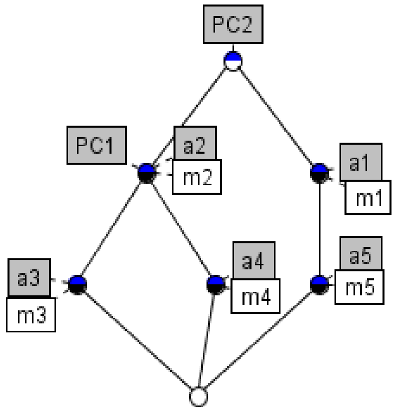

**Figure 8. Association View of Principal Class**

Now that we have presented our approach, we evaluate it on an industrial system. The next section presents the results of the evaluation of our approach.

## 4 Application of the Approach: Case Study Software

We evaluate our approach on an industrial software system. The software system drives blood disease analyses. Typically a machine is loaded with a sample of patient plasma and reagents (products) for chemical reactions. The machine performs the analysis and raw results are calculated and converted to interpreted results for their easier interpretation by doctors and medical staff. For the sake of precision and clarity, we only describe the software subsystem that manages the functional entities and processes, and operates with the database layer to manage the relevant data. Certain core functionalities, such as blood analysis data, reagents used by the machines, raw and interpreted results, patient data, and quality control, are the key features implemented at this layer.

Table 1 below shows some of the software quality metrics for our case study software [15].

**Table 1. Case Study Metrics**

| Class Name | LOC | NOM | NOA | DIT | LCOM |
|---|---|---|---|---|---|
| CPatient | 11,462 | 260 | 9 | 1 | 0.85 |
| CTest | 2792 | 81 | 13 | 1 | 0.72 |
| CProduct | 2552 | 77 | 6 | 1 | 0.72 |
| CResults | 1652 | 52 | 13 | 1 | 0.85 |
| CPersistency | 1325 | 67 | 29 | 2 | 0.97 |
| CGlossary | 1010 | 121 | 5 | 1 | 0.80 |

## 4.1 Design Metrics

Table 1 demonstrates some facts about the business entity layer: There is a clear lack of hierarchical structure and presence of huge service classes lacking cohesion, with large number of methods. Certain domain entities such as patient tube do not have associated classes which could have encapsulated the state and behavior related to these entities in a single class. Therefore, it offers a good case study to evaluate our approach since the software is developed in C# and is a mixture of object-oriented code and procedural one expressed within the same paradigm.

## 4.2 Identification of Principal Classes

We proceed with the identification of principal classes within our case study software system. Following are the results of the application of the first step of our approach.

**Table 2. Identification of Principal Classes**

| | |
|---|---|
| Total Principal Classes | 41 |
| Methods Associated Correctly | 403 |
| Methods Marked for slicing | 21 |
| False positives | 7 |

**Table 3. Some Principal Classes**

| ClassName | Method Count | ClassName | Method Count |
|---|---|---|---|
| PatientRecord | 40 | TestParams | 21 |
| RawResults | 57 | Calibration | 41 |
| InterpretedResults | 9 | Paramproduit | 3 |
| CalibrationData | 19 | PatientTube | 7 |
| QualityTest | 14 | BloodTest | 10 |
| TransParam | 3 | TransmitData | 8 |
| Product | 14 | Lot | 11 |
| DeviceHistory | 6 | Maintenance | 9 |

Table 2 describes results for the first step. We identified 41 principal classes, one for each user-defined types, and 403 methods were associated to these principal classes. List of some of the principal classes along with their method count is presented in Table 3.

Of all the methods observed, 7 methods were determined to be false positives, that is they were associated to the class to which they didn't belong. This happened for two reasons: Firstly, when methods were reading types, the number of reads for the actual type were lesser than other types. For example, the method verifying if the patient information is being used in a test. Now, this information can be placed in one principal class or another depending on the proximity of data and reengineer may be required to manually fix the position of such a method. The second reason is more complicated as it involved the cases where a method called another method to perform operations on its type while other principal classes are accessed directly. For example, a method for repeating test on patient data called another method to recreate test information while result information was directly created in the method.

## 4.3 Principal Class Compositions

The creation pattern provided useful information regarding the composition of principal classes. All in all 15 composition relationships were found of which three composition relationships were identified as common to 9 principal classes. Hence, three superclasses were identified with such a pattern. Moreover, useful domain information was also deduced using this pattern such as a patient has a tube, results have validation thresholds parameters, etc.

## 4.4 Hierarchies in Principal Classes

Once the task of identification of principal classes and their composition is achieved, we proceed to identify hierarchical information present within the principal classes. There are two sets of principal classes identified for our application.

**Simple Principal Classes.** Simple principal classes are those for which a single concept lattice merging all the views is generated for understanding the internals of the class. This helps decerning restructuring information from a single lattice and the reengineer doesn't need to produce lattices for different views. Figure 9 demonstrates concept lattice for QualityTest class in Table 3, whereby a single lattice is generated to understand the hierarchies of attribute accesses as well as method calls and type usage. The resulting concept lattice provides a clear decomposition of methods. The reengineer can further explore lattice information to create useful classes.

**Complex Principal Classes.** Complex principal classes are likely to consist of large number of methods. Concept lattice for such class showing all the views contains huge number of concepts and it is near to impossible to infer any

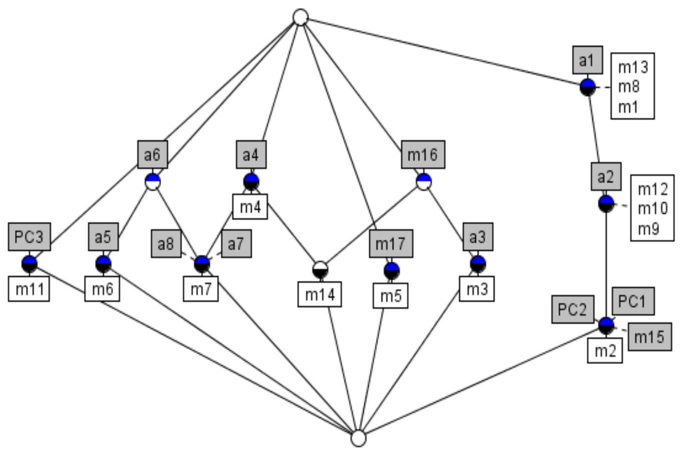

**Figure 9. A Simple Principal Class**

useful abstractions. Thus, for such classes subsequent views are generated to understand the hierarchies for their methods.

We consider class marked Calibration in Table 3. First, a fundamental view is generated to get the methods and attributes.

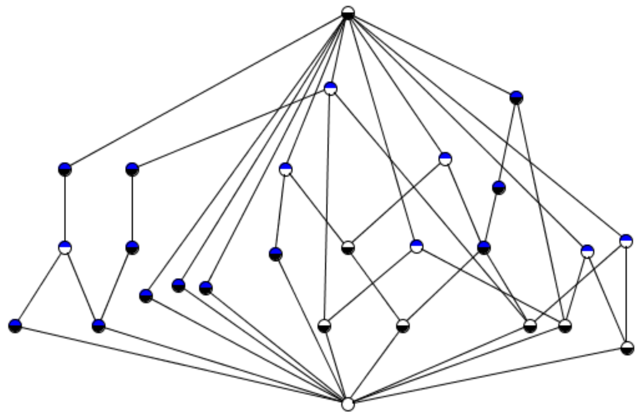

**Figure 10. Fundamental View of a complex Class**

Figure 10 doesn't reveal any particular decomposition apart from a few methods that are disjoint and can be placed in a separate class (attribute and method names are omitted for clarity). There are strong connections amongst methods and attributes. However, the common calls view presented in Figure 11 does indicate presence of common calls where methods m14, m15 and m17 invoke 6 common methods lying outside their principal class (their method numbers lay outside the method count of the principal class). These methods implement complex logic for the process of creating a calibration test, and the three methods actually implement the three types of tests. Hence a superclass is created to contain template behavior and three subclasses are then created for each process.

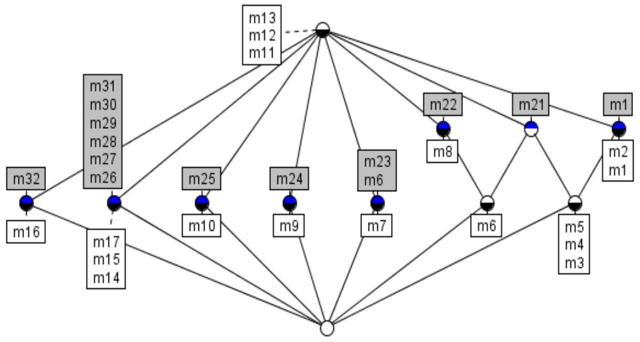

**Figure 11. Common Interaction View of a complex Class**

## 5 Discussion

The approach helps identifying objects missing in procedural object-oriented code. User-defined types and methods are used to extract class hierarchies from POC. This is done by analyzing the usage of atomic attributes in a principal class by the methods of that principal class. Moreover, method invocations and the degree of principal classes association help us detect common interactions and identification of association degree of various principal classes. The presence of user-defined types, groupings of primitive types for the representation of domain entities, is essential for the approach to identify principal classes. In addition methods are assumed to be crisp in their functionality in that they implement functionality pertaining to a particular task and they are not huge. In the absence of these two elements, the restructuring approach would fail because of failure to identify principal classes and failure to correctly assign methods to principal classes. If methods are huge providing complex functionality, slicing [13] and refactoring [12] should be applied first to split chunks of related instructions into methods.

Various views are generated to understand the internals of principal classes. We believe that these views do not automate the task of object identification. They act as a tool to guide the restructuring activity. Reengineer needs to interpret different views to combine them in an intelligent manner as the three views are not completely orthogonal. First, she needs to understand the compositions. Once compositions are inferred, hierarchies of methods and attributes in principal classes are identified with the fundamental view. Later, new classes obtained from common interaction view and associations view should be integrated to the overall class model of the principal class in question. Common interaction view should also be considered to understand the collaborations amongst methods of a principal class. Heuristics and guidelines can be developed for the refinement of identified objects, as the one described in [22]. However, such heuristics are application dependent. We provide general views from which the reengineer can infer class hierarchy information and refine it with domain knowledge.

## 6 Related Work

Several research works aim to specify patterns of bad design in code. A catalog of code smells and steps for their possible refactoring to improve the software design is presented in seminal work of Fowler et al. [12]. The code smells are a consequence of constant maintenance efforts resulting in small design problems. But the overall hierarchy doesn't suffer from the design flaws and absence of classes for domain objects. Demeyer et al. [9] provide a catalog of design flaws in their work. Possible flaws along with the possible detection techniques and solutions are presented from a reengineering perspective. Anti-patterns describe solutions that degenerate into negative consequences when applied to a problem, and higher-level, architectural Anti-Patterns have been presented in [3]. Globally, the aforementioned half-baked classes in POC in a respect do represent an example of the Blob anti-pattern, and the God class (behavioral form) presented in [21]. The code smells that we presented in a way describe anti-pattern manifestation in code. An effort to formalize the copious design defects (code smells, anti patterns, and design pattern defects) presented in the literature has been introduced in [18] to remove ambiguities in their interpretation. This work actually takes different design defects as input. This work bases itself on the existing code smells and tries to formalize them, while our work concretely contributes the code smells related to procedural object-oriented code to the existing code smells. An automated approach for suggesting defect-correcting refactoring using relational concept analysis is proposed in [19]. The approach aims and suggest to make classes more cohesive and less coupled by removing Blob anti-pattern and suggest refactoring in this regard. An approach to detect design defects by applying heuristics on design metrics has been presented in [17]. A pattern-like description of design flaws and a systematic description of detection metrics is developed for each design flaw.

Several work attempted to transform procedural code to object-oriented one [4, 6, 7, 16]. Some approaches were taken to convert procedural code into data flow programs [20]. Newcomb *et al* proposed an Hierarchical Object-Oriented State-Machine Model which is between conventional object-oriented modeling languages, state-based reactive specification systems, and event-driven programming models [20]. COBOL records are mapped to classes and each procedure is mapped to a state machine associated to a method. Several refactorings and transformations are applied to abstract and merge the resulting methods. De Lucia *et al.* [7, 16] describe the Ercole approach for migrating programs written in RPG, the business application programming language, to object-oriented programs. Among the different steps of the approach, one is abstracting an object-oriented model which is centered around the persistent data stores. Subroutines and groups of call-related subroutines are then candidate methods.

Concept formation methods have been applied for object identification in procedural code. Sahraoui et al. in [23] proposed an approach for identifying objects in procedural code. The approach combines metrics calculation with several FCA-based analysis steps in class identification and further graph-based reasoning to detect method associations for newly identified classes. Our approach carries forward their approach in that we generate two more views to refine our class hierarchies. Arie van Deursen proposes to use FCA and to semi-automatically restructure the legacy data structures that can be then used as a starting point to object identification [27]. A comparison of the object identification with clustering and concept analysis techniques is also presented by the authors. An approach to transform a COBOL legacy system to a distributed component system is proposed in [5]. The overall purpose is to reduce the complexity of the lattices through the subgraph identification by the application of an eclectic approach. However, the focus remains the decomposition of programs into meaningful components and no further hierarchical decomposition is mentioned. All these techniques aim to move from procedural code to object-oriented code. However, our goals are bit different than defined in these studies in that we aim to search for the identification of patterns appearing due to absence of design and exploiting them for identifying objects in procedural object-oriented code buried in half-baked classes.

FCA is proposed for class hierarchy reengineering by Snelting and Tip [25]. The authors proposed a FCA-based method for adapting a class hierarchy to a specific usage thereof. It comprises a study of the way class members are used in the client code of a set of applications. The study enables the identification of anomalies in the design of class hierarchies, e.g., class members that are redundant or that can be moved into a derived class. FCA-based understanding of class structures is introduced in [2]. The authors identify pattern of views based on FCA to understand access of class attributes and method usage for existing class hierarchies. Dekel uses FCA to visualize the structure of the class in Java and to select an effective order for reading the methods [8]. Method call graphs is superimposed onto the concept lattice to obtain an embedded call-graph, which provides a detailed visualization of the interaction with the class.

## 7 Conclusion

Procedural object-oriented code appears due to absence of software design or due to its erosion over a period of time. This paper describes some of the design problems occurring in procedural object-oriented code. These include huge classes, scarce hierarchical links, global enumerated types and absence of abstractions for domain entities. We presented our approach for the identification of useful abstractions in procedural object-oriented code. For this purpose, principal classes are identified, their composition links are discovered and the hierarchical relationship of their methods is identified through the usage of FCA. Various views are provided with different information to infer hierarchies of methods and attributes. Concept lattices provide us with several modularization proposals for methods and attributes present in principal classes. Our approach bases itself on the presence of user-defined types in procedural object-oriented code. In their absence, a manual task is required to identify them. Our approach also assumes presence of well-focused methods. Our future work includes definition of algorithm for moving from concepts lattices to meaningful classes.

## References

[1] G. Arévalo, S. Ducasse, and O. Nierstrasz. Understanding classes using X-Ray views. In *International Workshop on MASPEGHI 2003 (ASE 2003)*, pages 9–18. CRIM — University of Montreal (Canada), Oct. 2003.

[2] G. Arévalo, S. Ducasse, and O. Nierstrasz. Discovering unanticipated dependency schemas in class hierarchies. In *CSMR'05*, pages 62–71. IEEE Computer Society, Mar. 2005.

[3] W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998.

[4] G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, 24(2):61–70, Feb. 1991.

[5] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca. A Case Study of Applying an Eclectic Approach to Identify Objects in Code. In *IWPC'99*, pages 136–143. IEEE Computer Society, May 1999.

[6] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Softw. Pract. Exper.*, 26(1):25–48, 1996.

[7] A. Cimitile, A. D. Lucia, G. A. D. Lucca, and A. R. Fasolino. Identifying objects in legacy systems using design metrics. *J. Syst. Softw.*, 44(3):199–211, 1999.

[8] U. Dekel and Y. Gil. Revealing class structure with concept lattices. In *WCRE*, pages 353–362. IEEE Press, Nov. 2003.

[9] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

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

[11] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[13] K. B. Gallagher and J. R. Lyle. Using Program Slicing in Software Maintenance. *Transactions on Software Engineering*, 17(18):751–761, Aug. 1991.

[14] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.

[15] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[16] G. A. D. Lucca, A. R. Fasolino, P. Guerra, and S. Petruzzelli. Migrating legacy systems towards object-oriented platforms. In *ICSM '97*, pages 122–129, Washington, DC, USA, 1997. IEEE Computer Society.

[17] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *ICSM'04*, pages 350–359, Los Alamitos CA, 2004. IEEE Computer Society Press.

[18] N. Moha, Y.-G. Gueheneuc, and P. Leduc. Automatic generation of detection algorithms for design defects. In *ASE '06*, pages 297–300, Washington, DC, USA, 2006. IEEE Computer Society.

[19] N. Moha, A. M. R. Hacene, P. Valtchev, and Y.-G. Guéhéneuc. Refactorings of design defects using relational concept analysis. In *ICFCA*, pages 289–304, 2008.

[20] P. Newcomb and G. Kotik. Reengineering procedural into object-oriented systems. In *WCRE'95*, pages 237–250. IEEE CS, 1995.

[21] A. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.

[22] H. A. Sahraoui, H. Lounis, W. Melo, and H. Mili. A concept formation based approach to object identification in procedural code. *Automated Software Engineering Journal*, 6(4):387–410, 1999.

[23] H. A. Sahraoui, W. Melo, H. Lounis, and F. Dumont. Applying Concept Formation Methods to Object Identification in Procedural Code. In *ASE '97*, pages 210–218. IEEE, IEEE Computer Society Press, Nov. 1997.

[24] M. Siff and T. Reps. Identifying modules via concept analysis. *Transactions on Software Engineering*, 25(6):749–768, Nov. 1999.

[25] G. Snelting and F. Tip. Reengineering Class Hierarchies using Concept Analysis. In *ACM Trans. Programming Languages and Systems*, 1998.

[26] M. Streckenbach and G. Snelting. Refactoring class hierarchies with KABA. In *OOPSLA '04*, pages 315–330, New York, NY, USA, 2004. ACM Press.

[27] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *ICSE '99*, pages 246–255. ACM Press, 1999.