STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIX**, Number 2, 2014

FAOS - A FRAMEWORK FOR ANALYZING OBJECT-ORIENTED SOFTWARE SYSTEMS

ZSUZSANNA MARIAN, GABRIELA CZIBULA, AND ISTVAN GERGELY CZIBULA

Abstract. In this paper we are presenting a software framework we have developed for supporting several machine learning-based techniques which were introduced for solving some software engineering problems. Search-based software engineering is a research and practice domain which is based on the idea of reformulating all software engineering problems as search problems, and on applying metaheuristic search techniques for solving these problems. We have previously introduced in the search-based software engineering literature several machine learning-based solutions for solving problems of major importance within software engineering, namely: software remodularization both at the class and the package level and software design defect detection. The *FAOS* (*Framework for Analyzing Object-oriented Software systems*) framework was used for developing all the above mentioned techniques. A comparison of the *FAOS* software with similar existing approaches in the literature is also provided, emphasizing its characteristics and advantages.

1. Introduction

Search-based software engineering [10] is a research and practice domain appeared in the field of software engineering, and is based on the idea of reformulating all software engineering problems as search problems. Researches within this field are conducted towards developing metaheuristic search techniques for solving different software engineering problems such as testing, module clustering, cost estimation, requirements analysis, systems integration, software maintenance and evolution of legacy systems.

Harman and Jones claim in [9] that Software Engineering is ideal for the application of metaheuristic search techniques, such as genetic algorithms, simulated annealing and tabu search. But these techniques are not the only

Received by the editors: September 15, 2014.

2010 *Mathematics Subject Classification.* 68N30, 68N19.

1998 *CR Categories and Descriptors.* D.2.8 [**Software Engineering**]:Metrics – *Product metrics*; D.2.10 [**Software Engineering**]: Design – *Methodologies*; D.2.13 [**Software Engineering**]: Reusable Software – *Reusable libraries*;

Key words and phrases. software framework, object-oriented software system, software development.

computational intelligence techniques applicable to software engineering tasks. Machine learning algorithms or fuzzy approaches can be used as well. These techniques could offer solutions to difficult problems within software engineering and may provide ways of finding acceptable solutions in situations where perfect solutions are impossible or practically infeasible. Even if the relationship between searching and learning is not obvious, from the perspective of human reasoning modelling, machine learning techniques can be connected to intelligent searching techniques. Essentially, an artificial learning system (irrespective of the learning strategy used) has to search (or discover) an optimal sequence of actions for solving a particular problem. Thus, machine learning based solutions for solving software engineering problem are associated to the search based software engineering topic.

We have previously introduced several machine learning-based techniques for solving problems of great importance during software maintenance and evolution. The approached problems were: software remodularization at the class [16] and the package level [18] and software design defect detection [3].

The main contribution of this paper is to introduce the *FAOS* framework (*Framework for Analyzing Object-oriented Software systems*) which we have used for the development of all the previously mentioned techniques. *FAOS* was designed to be generic enough to offer a support for analyzing an object-oriented software system and to easily extract from it relevant information, as well as to provide reference implementation for several software metrics which are useful to measure software quality [11].

The rest of this paper is organized as follows. Section 2 presents the related work as well as the techniques already introduced in the literature which were developed using the *FAOS* framework. Our software framework proposal is introduced in Section 3. Section 4 emphasizes the advantages of the framework proposed in this paper and provides a comparison with existing related approaches. Conclusions of the paper as well as directions to further improve and extend the *FAOS* framework are outlined in Section 5.

2. Background

This section presents a literature review on existing related software frameworks, as well as a review on the machine learning-based approaches which we have previously introduced in the search-based software engineering literature and were developed using the *FAOS* framework.

2.1. **Literature review.** In search-based and machine learning-based software engineering researchers often implement their approaches in the form of different tools or frameworks that can be used both by researchers for the comparison of results, but also by software developers to get help with their

everyday tasks. While there is no other framework with the exact same functionalities as *FAOS*, there are some frameworks in the literature which are similar to parts of it. In this section we will present those frameworks that we have met in the literature and we consider that are similar to the *FAOS*.

One such framework is *JDeodorant*, which is actually an Eclipse plugin, available at [13], developed for identifying bad smells [8] in Java code. *JDeodorant* provides a specialized view in Eclipse and is able to identify four types of bad smells: *God Class*, *Long Method*, *Feature Envy* and *Type Checking*. Each of them opens a new view in Eclipse, and these views have an *Identify Bad Smells* button. Clicking this button the corresponding bad smells are identified in the selected element, which can be a whole project, a package, a class and for some smells even just a method. A big advantage of *JDeodorant* is that it not only suggests refactorings to be applied to remove the identified bad smell, but it can also give a preview of how the system would look like after the refactoring and perform the refactoring.

Another tool which is similar to *FAOS* is *iPlasma*, available at [12], which is an integrated environment designed for the quality analysis of object-oriented software systems. Its code analysis is implemented for two programming languages, Java and C++, their internal representation being the same. There are multiple possible analysis methods implemented in *iPlasma*, it can compute the values of a high number of software metrics, it implements the Detection Strategies presented in [19], and it also has a module for detecting duplicate code.

The *Bunch* tool is similar to the *packageRestructuring* module of *FAOS* (presented in Section 3.2.3), it uses different search algorithms (Steepest Ascent Hill Climbing, Next Ascent Hill Climbing and Genetic Algorithms) for finding a good partitioning of the software system, using the *MQ* measure as objective function [14, 21].

There are also other tools, which can compute the value of different software metrics for a system or part of a system, some freely available, others commercial. For example, for Eclipse we have found two different plugins, both called *Metrics*. The first, available at [20] can compute the value of 23 different software metrics. It allows the setting of different threshold values for the metrics, and when a class or a method has a higher value than the threshold, it is marked in the editor. The other Eclipse plugin that computes the values of software metrics is available at [7]. It has implemented the value of seven metrics computed for a method, and four metrics computed for a class.

2.2. **Machine learning based software engineering.** In [17] we have approached the problem of improving the quality of a software system design, an important issue during the evolution of object-oriented software systems.

Starting from the fact that software metrics are essential in measuring the software quality [2], a metric-based high-dimensional representation of the entities from a software system was introduced. Using this representation, we have defined a distance semi-metric between the entities of the software system. An experimental validation of the distance semi-metric on two case studies was provided, illustrating that the distance function introduced in [17] may be successfully used for improving the internal structure of software systems.

In [15] we have worked on the problem of improving the quality of a software system design as well, but from a different point of view. Instead of using a high-dimensional representation of the entities, we decided to aggregate the values of different software metrics into a single score, which was used to guide a hierarchical clustering process. For this approach we have identified 16 software metrics that measure the cohesion, coupling and size in a software system and used their values in the aggregated score. Experiments on small software systems showed that our approach is capable of identifying a good structure of the software system.

In the paper [16] we have investigated through a case study, whether the use of unsupervised learning methods can be beneficial in the process of automatic refactoring [8] identification. The results of two algorithms (one that uses hierarchical clustering and one that does not) are compared for a case study, and show that the algorithm that uses hierarchical clustering is capable of identifying refactorings which are not found by the non-clustering based algorithm.

The problem of software refactoring at the package level [22] was approached in [18] using hierarchical clustering. Two approaches are proposed in order to help software developers in designing well-structured software packages. The first approach takes an existing software system and re-modularizes it at the package level using hierarchical clustering, in order to obtain better-structured packages. The second method we propose considers a certain structure of packages for a software system and suggests the developer the appropriate package for a newly added application class. The proposed methods are based on computing the value of several features, aggregated into a single score, which was used as a distance measure during the clustering process. Computational experiments are performed on two open-source frameworks and the algorithms introduced in [18] have proven to perform well in comparison to existing similar approaches.

Another problem within the software engineering domain that we have approached from a machine learning perspective is the problem of automatically detecting application classes having design defects, presented in [3]. We have proposed a method based on relational association rule mining for detecting faulty application classes in existing software systems. The proposed method

is based on mining of relational association rules for identifying design defects [19] in software. Experiments on open-source software are conducted in order to detect defective application classes in object-oriented software systems.

3. The *FAOS* software framework

In this section we introduce an application programming interface (API) that is realized in JDK 1.6 and is intended to analyze software systems written in Java in order to extract from them relevant information such as attributes, methods, application classes and relationships between all these entities. The *FAOS* framework was used for developing the machine learning-based software engineering techniques presented in Subsection 2.2.

3.1. **Theoretical considerations.** First, we will start with presenting some theoretical aspects on which the software framework is based on.
Let $S = \{s_1, s_2, ..., s_n\}$ be a software system, where $s_i, 1 \leq i \leq n$ is an *entity* (an application class, a method from a class or an attribute from a class).

Let us consider the following. (1) $Class(S) = \{C_1, C_2, \ldots, C_l\}$, $Class(S) \subset S$, is the set of applications classes from the software system S. Each application class C_i $(1 \leq i \leq l)$ is a set of methods and attributes, i.e., $C_i = \{m_{i1}, m_{i2}, \ldots, m_{ip_i}, a_{i1}, a_{i2}, \ldots, a_{ir_i}\}$, where m_{ij} $(\forall j,\ 1 \leq j \leq p_i)$ are methods and a_{ik} $(\forall k,\ 1 \leq k \leq r_i)$ are attributes from C_i. (2) $Meth(S) = \bigcup_{i=1}^{l} \bigcup_{j=1}^{p_i} m_{ij}$, $Meth(S) \subset S$, is the set of methods from all the application classes of the software system S. (3) $Attr(S) = \bigcup_{i=1}^{l} \bigcup_{j=1}^{r_i} a_{ij}$, $Attr(S) \subset S$, is the set of attributes from the application classes of the software system S.

Based on the above notations, the software system S may be defined as $S = Class(S) \bigcup Meth(S) \bigcup Attr(S)$ [4].

3.2. **The framework design.** The *FAOS* framework is currently composed of three main modules, *Analyzer*, *Metrics* and *PackageRestructuring*. The following subsections will present in detail these three modules.

3.2.1. *The Analyzer module.* The first module of the *FAOS* framework is the *Analyzer*, which performs the actual analysis of the software system and builds an internal representation of it, which can be further used for other tasks. The current implementation of the *Analyzer* module can only analyze software systems written in Java, and it requires either the compiled *.class* files or a *jar* archive for the analysis.

A simplified class diagram of the *Analyzer* module is presented in Figure 1. This diagram (and the diagrams for the rest of the modules) presents only the

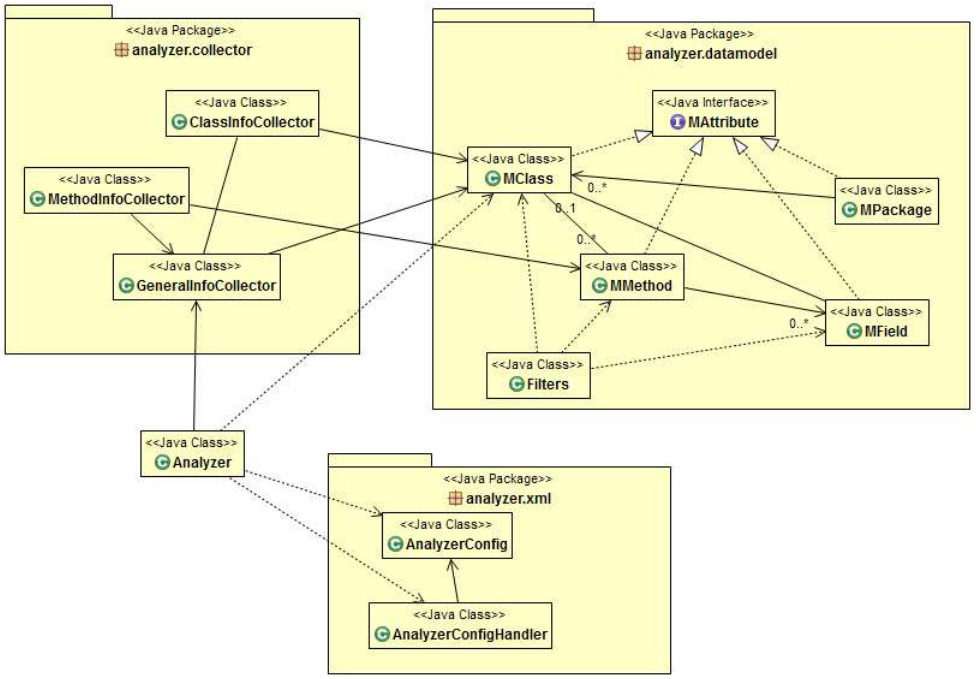

FIGURE 1. Simplified class diagram for the *Analyzer* module.

packages of the module and the classes from each package, without including methods and attributes of these classes. In order to avoid overcrowding the diagram, only the most important relations between the classes are marked. In the following we will present in detail the components of Figure 1.

Datamodel for representing software systems. The *Analyzer* module of the *FAOS* framework provides an internal representation of the analyzed software system. This subsection will present the datamodel used, i.e., how we represented elements of the software system. The datamodel is a very important component of the *FAOS* framework, because the rest of the modules use the datamodel as well. This part is presented on the *datamodel* package from Figure 1.

An important base for our datamodel is the *MAttribute* interface, which is implemented by the classes representing entities of a software system: packages, classes, methods and fields. By using a common interface for all these entities the definition of different approaches that treat these entities equally is a lot easier. Besides a method to return the name of the entity, the *MAttribute* interface has another method which returns the set of relevant properties for the given entity.

The *MClass* class represents an application class from a software system. It is a complex class, having many attributes, the most important of them being: the name of the class, the name of the superclass, an array with the names of the implemented interfaces, a list of methods and a list of fields. Methods of the *MClass* class include getters and setters for the fields (combined with some filtering possibilities in case of the lists of methods and fields), methods that verify if the class is an interface, abstract and so on.

A method from an application class is represented by the *MMethod* class. This is a complex class as well, its most important fields being: the name of the method, the *MClass* object of the application class to which this method belongs, the set of fields that are used by the method, the set of methods called by the method, the set of local variables created and used in the method, the set of parameters of the method. Methods of the *MAttribute* class contain getters and setters for most fields, methods that verify if the method is public, private, abstract, static, constructor, a setter or a getter.

An attribute, or field, of an application class is represented by the *MField* class. This is a simple class, it only has attributes for the name of the field, the *MClass* object for the owner class of the field and its type. Besides getters and setters it has methods that verify if it is private, public, static, constant or the *"this"* field.

There is a fourth class that implements the *MAttribute* interface, the *MPackage* class. This class represents a software package and it simply has a list of objects of type *MClass* and a name.

The *ASM* bytecode manipulation framework. The main role of the *Analyzer* module is to take a set of compiled Java classes and extract from them a set of *MClass* objects (together with the corresponding list of *MMethods* and *MFields*) that can later be used for different other tasks. For the analysis of the compiled code we used the *ASM* Java bytecode manipulation framework [1], version 4.0. This framework provides the users with a set of interfaces and abstract classes which can be easily implemented or extended to perform specific tasks. One of the base classes from the *ASM* API is the *ClassVisitor* abstract class, which, as its name suggests, implements the *Visitor* design pattern. This class has a series of methods, corresponding to the content of a compiled Java class file, each being automatically invoked by the *ClassReader* object from *ASM*, when the corresponding element is visited.

Two very important methods of the *ClassVisitor* class are the *visitField* and *visitMethod* methods, which are called for every attribute and method of the analyzed class. These methods return other *Visitor* objects, a *FieldVisitor* or a *MethodVisitor*. The *MethodVisitor* abstract class is used to visit elements of a method from an application class, there is a separate method for each

bytecode instruction category. The *FieldVisitor* class was not used it was not needed for obtaining the information for our representation of a field.

Since the *ClassVisitor* and the *MethodVisitor* classes are abstract, we extended them and implemented our classes that perform the analysis and build the corresponding *MClass*, *MMethod* and *MField* objects. This part of the *Analyzer* module is presented in the *collector* package from Figure 1. The *ClassInfoCollector* class extends the *ClassVisitor* class and builds an *MClass* object corresponding to the analyzed class. We have defined the *MethodInfoCollector* class as well, which extends the *MethodVisitor* class and builds an *MMethod* object for the visited method. The third class from the *collector* package, the *GeneralInfoCollector*, was written to "coordinate" the other two classes. This class maintains the list of *MClass* objects that are created one-by-one by the *ClassInfoCollector* object as the classes are analyzed.

Configuring the *Analyzer*. The third package of the *Analyzer* module, called *xml* contains the two classes needed for configuring the analysis. For the configuration we have chosen to use *xml* files because of their flexibility.

The *AnalyzerConfig* class is a simple class which contains all the information read from the configuration file, while the *AnalyzerConfigHandler* class is the class which reads the *xml* file and builds the corresponding *AnalyzerConfig* object.

Finally, the last class in the *Analyzer* module is the *Analyzer* class. This class needs the name of the configuration file, uses the *AnalyzerConfigHandler* to get the corresponding *AnalyzerConfig* object and, in the *analyzeProject* method, creates a *GeneralInfoCollector* object and starts the analysis. This method returns the list of *MClass* objects extracted from the analyzed software system, a list which can later be used to all sorts of further tasks and analyses.

3.2.2. *The Metrics module.* The second module of the *FAOS* framework is the *Metrics* module, which contains the implementation of different software metrics using the datamodel presented in Subsection 3.2.1. This module is divided into two main packages, the *classLevel* package contains the implementation of different software metrics computed for a class, while the *packageLevel* package contains the implementation of software metrics and measures computed for a software package. The simplified class diagrams for these two packages are presented on Figures 2 and 3.

From Figure 2 it can be seen that the central class in the *classLevel* package is the *ClassMetric* abstract class, which is extended by every software metric in this package. While currently these metrics can only be computed for an application class, we would like to extend them to measure other entities as well, for example methods and fields. Thus the most important method of the *ClassMetric* class is the *measure* method, which receives as parameter an object of type *MAttribute*. This method, based on the exact type of the

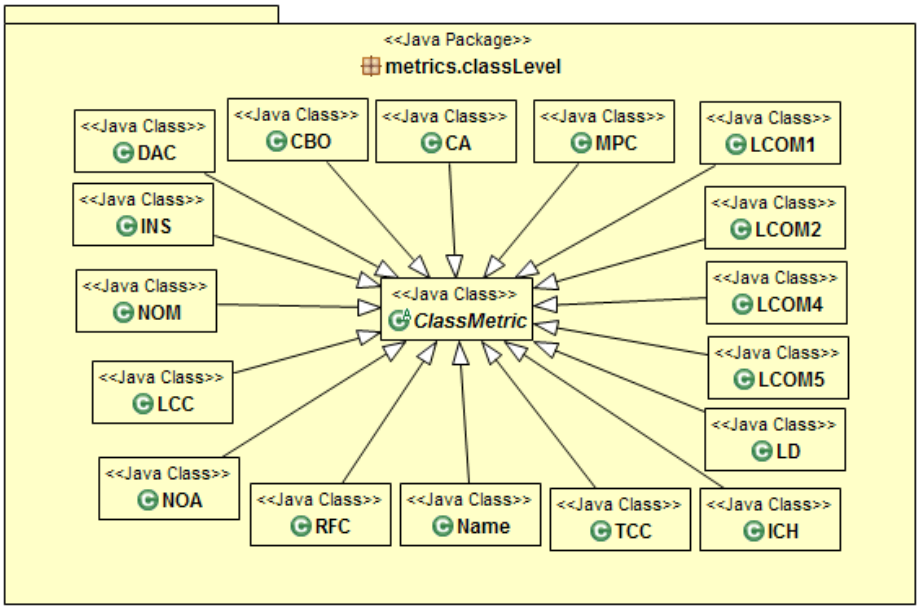

FIGURE 2. Simplified class diagram for the class-level metrics from the *Metrics* module.

received parameter, will call one of the following three methods: *measureClass*, *measureMethod* and *measureField*.

As it can be seen from Figure 2 currently there are 17 different class-level metrics implemented in the *Metrics* module. 16 of them are the ones used for the aggregated metrics-based class-level restructuring and for design defect detection (both presented in Section 2.2). The 17^{th} measure, called *Name* is a measure that we have defined and used for the 4^{th} feature of our package-level restructuring approach.

The second part of the *Metrics* module contains the implementation of different metrics and measures for packages of a software system. The simplified class diagram of this package is presented on Figure 3. It has the abstract class *PackageMetric* to have a common superclass for all metrics for packages, but there are actually three different types of package metrics, as suggested by three of the four subpackages inside the *packageLevel* package. These types are the following:

- Metrics which are computed for one package from the partitioning of the software system. Such metrics and measures are placed in the *onepackage* subpackage and extend the *OnePackageMetric* abstract

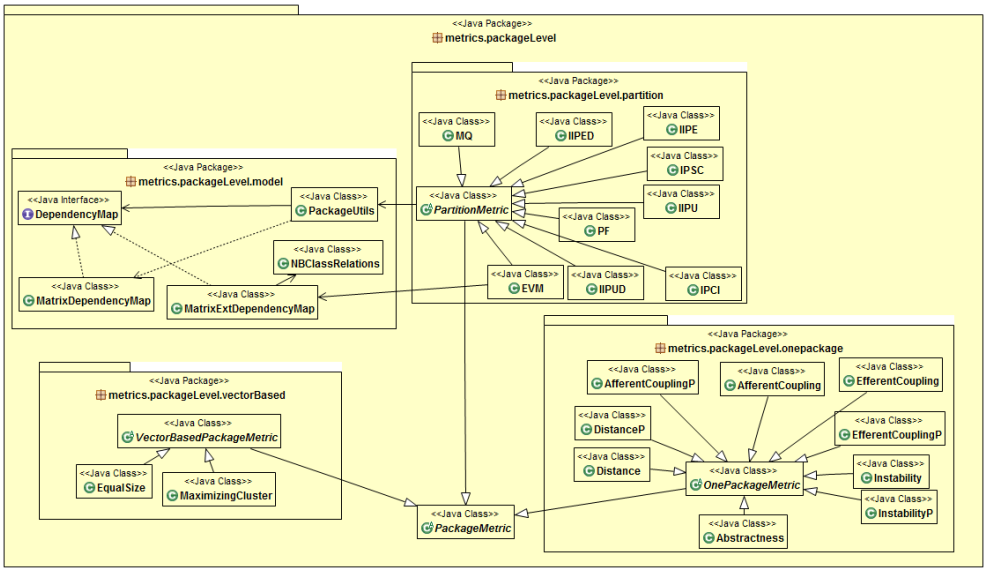

FIGURE 3. Simplified class diagram for the package-level metrics from the *Metrics* module.

class. The implemented metrics are well-known in the literature for evaluating software packages [6].

- Metrics that are computed for a whole partition of the software system (they take into consideration all packages). Such measures are placed in the *partition* subpackage and they extend the *PartitionMetric* abstract class and most of them were taken from [6].
- Measures that evaluate a partition, but return a vector with the value of different other measures, instead of returning one single value. These measures are placed in the *vectorBased* subpackage and they extend the *VectorBasedPackageMetric* abstract class. The two measures implemented in this subpackage were taken from [23].

The fourth subpackage in the *packageLevel* package, called *model*, contains classes that help compute the value of the metrics and measures from the other three packages. In order to avoid constantly checking what kind of relations are between different classes (if any), we decided to create a map which is built only once before computing the metrics and memorizes all the dependencies between the classes from the software system.

3.2.3. *The PackageRestructuring module.* The third module of the *FAOS* framework is called *packageRestructuring* and the package-level restructuring approach described in Section 2.2 is implemented in it, so, it is a little less

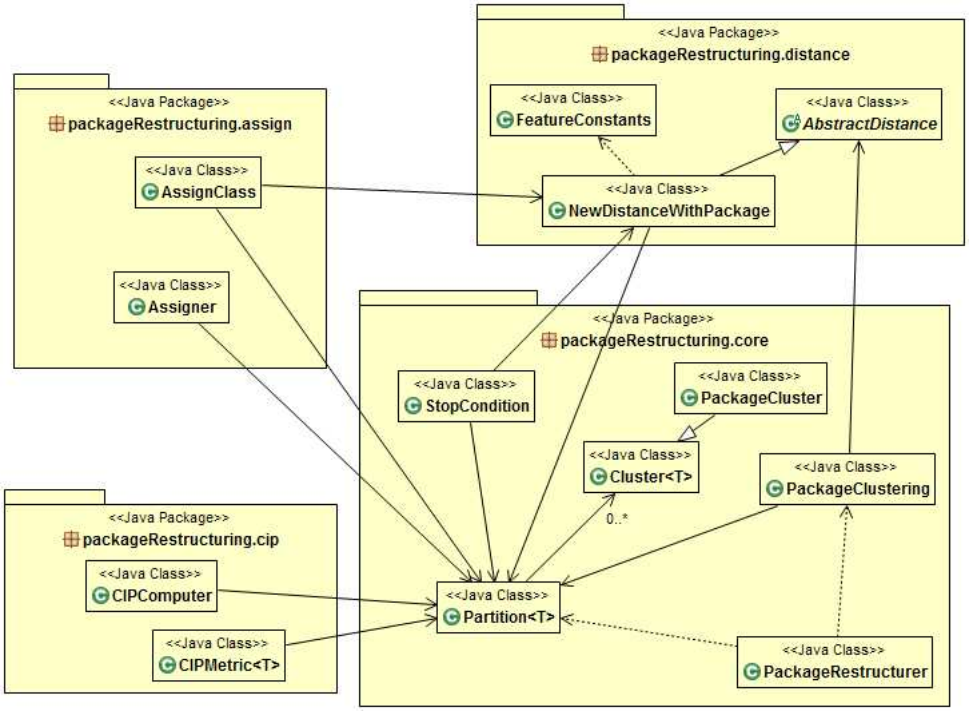

FIGURE 4. Simplified class diagram for the *PackageRestructuring* module.

abstract than the previous two modules. The simplified class diagram of this module is presented on Figure 4.

The abstract and reusable part of this module is in the *core* subpackage, where general classes related to the clustering process can be found, such as: *Cluster* and *Partition*. This subpackage also contains classes that extend them, and implement the tasks specific for restructuring at package level, for example the *PackageCluster* class, which contains a list of *MClass* objects.

The other three subpackages contain the implementation of the algorithms and measures defined for our package-level restructuring and presented in [18].

3.3. **Example of using the *FAOS* framework.** In this section we will present how an analysis can be performed with the *FAOS* framework. As an example we will use an open-source software framework, called *DbUtils* (version 1.5) [5], a library consisting of a small set of classes, which are designed to make working with JDBC easier. It consists of 25 classes, which are divided into three packages. Using the *DbUtils* framework, we will give examples of the results of the analysis of the three modules of the *FAOS* framework.

Class name: *BeanHandler* Package name: *org/apache/commons/dbutils/handlers* Super class name: *java/lang/Object* Interfaces implemented: *org/apache/commons/dbutils/ResultSetHandler* Is abstract: *false* Is interface: *false*
Number of fields: 3 List of fields: *this* of type *org/apache/commons/dbutils/handlers/BeanHandler* *type* of type *java/lang/Class* *convert* of type *org/apache/commons/dbutils/RowProcessor*
Number of methods: 3 List of methods: $<init>$ with parameter of type *java/lang/Class* $<init>$ with parameters of type *java/lang/Class;* *org/apache/commons/dbutils/RowProcessor* *convert* with parameter of type *java/sql/ResultSet*

TABLE 1. Information extracted about the *BeanHandler* class.

Using the *Analyzer* module. Using the *analyzeProject* method of the *Analyzer* class for the analysis of the *DbUtils* framework, the method returns a list of 25 *MClass* objects, each of them corresponding to one of the classes from the analyzed framework. One of these objects corresponds to the *BeanHandler* class, and the information that was extracted from it is presented on Table 1.

The list of fields attribute contains a list of *MField* objects, from which we presented only the name and type of the field. Similarly, the list of methods attribute contains a list of *MMethod* objects, but here we only presented the name and parameters of the methods. The $<init>$ method name represents the constructor of the class. Since *MMethod* objects are more complex than *MField* objects, on Table 2 we will present the information extracted for the *convert* method of the *BeanHandler* class.

From the description attribute of the method it can be seen that it has only one parameter of type *java/sql/ResultSet* and that it returns an object of type *java/lang/Object*. It has one single local variable, which is actually the parameter, because local variables include the parameters of the methods as well. It calls the *toBean* method of this parameter once (in a separate map, whose components are not presented on Table 2 we retain the frequency of these calls as well, because some software metrics need such information), and it uses all the fields of the *BeanHandler* class.

Name: *handle* Owner class: *org/apache/commons/dbutils/handlers/BeanHandler* Description: *(java/sql/ResultSet;)java/lang/Object;* Is abstract: *false* Is static: *false* Is private: *false* Is public: *true* Is constructor: *false* Is setter or getter: *false* Is from superclass: *false*
Local variable: *rs* of type *java/sql/ResultSet*
Called method: *toBean* from class *org/apache/commons/dbutils/RowProcessor*
Used fields: *this* of type *org/apache/commons/dbutils/handlers/BeanHandler* *type* of type *java/lang/Class* *convert* of type *org/apache/commons/dbutils/RowProcessor*

TABLE 2. Information extracted about the *convert* method of the *BeanHandler* class.

EVM	**MQ**	**IIPE**	**IIPED**	**IIPU**	**IIPUD**	**IPCI**	**IPSC**	**PF**
-78	1.16	0.53	1	0.43	1	0.83	1	1

TABLE 3. The value of package-level software metrics from the *Metrics* module computed for the partitioning of the *DbUtils* framework.

Using the *analyzeProjectPackage* method of the *Analyzer* class will return the exact same *MClass* objects, but inside some *MPackage* objects, representing the packages in which these classes can be found. For the *DbUtils* framework, we will have a list of three *MPackage* objects.

Using the *Metrics* module. We have first used the *Metrics* module to compute the value of different class-level software metrics for the application classes of the *DbUtils* framework.

On Table 3 we have presented the value of some package-level software metrics for the *DbUtils* software framework. More exactly, we have computed the value of all implemented package-level software metrics, whose value is computed for a whole partition.

Using the *PackageRestructuring* module. The third module of the *FAOS* framework can be used for the package-level restructuring of a software framework. For the *DbUtils* framework our package-level restructuring approach, implemented in the *PackageRestructuring* module, identifies a partition composed of four packages, instead of the original three. As presented in [18], we consider that this package structure is better than the original one.

4. Discussion and comparison to related work

In the previous section we have presented in detail the three components of the *FAOS* framework. Since it has both abstract components (like the *Metrics* module, which can be easily extended with other metrics) and implementations of different algorithms (like our package-level restructuring implemented in the *PackageRestructuring* module) it is quite different from existing software frameworks, but the frameworks presented in Section 2.1 can be compared to different parts of it.

Both the *JDeodorant* [13] Eclipse plugin and the *iPlasma* [12] tool have internal models for the representation of the analyzed system similar to the datamodel from the *FAOS* framework. *iPlasma*'s internal representation is called *Memoria* and in case of *JDeodorant* the representation is extracted from the Abstract Syntax Tree and is more detailed than our datamodel, it contains the individual statements from a method as well (which can be easily understood, because *JDeodorant* can actually perform different refactorings and for this they need the exact instructions from the code).

Both *JDeodorant* and *iPlasma* are complete tools, ready to be used for different functionalities, which is an advantage over the *FAOS* framework. *iPlasma* can analyze software systems written in the C++ programming language too, a feature which is not present neither in *FAOS* nor in *JDeodorant*. On the other hand, the *FAOS* framework is more flexible than the other two frameworks, the *Analyzer* module returns a list of *MClass* objects which can be used for different tasks, one of them being the identification of refactoring opportunities.

The *Bunch* tool does not have any internal representation or analyzer module for the software system it works with, it needs as input the *Module Dependency Graph* (*MDG*) of the software system. On the other hand this makes it programming language-independent, since tools to generate the *MDG* of a software system can be found for different programming languages.

In conclusion, there are some other tools in the literature which have similar features like the *FAOS* framework, but there is no other tool which can perform the same tasks. *JDeodorant* and *iPlasma* are suitable for situations when somebody wants just to analyze the source code, look for abnormal metric values or bad smells. The *FAOS* framework is suitable when somebody wants to analyze a system, and perform some tasks on the resulting list of entities. Thus the main advantage of *FAOS* can be considered that it is easily extendable: new software metrics can easily be added to it, and the list of entities returned by the *Analyzer* module can be used for further analysis.

We summarize in the following the main advantages of using the framework proposed in this paper. First, it offers a mechanism to model and analyze a software system and to provide the user with different software metrics which

are useful to measure different characteristics of the software. Moreover, when developing the framework we have made an abstraction of the process of extracting from a given software system the entities and the existing relationships between them. The generality of our approach gives the user the possibility to easily define new software metrics by using the information extracted from the analyzed software system. Not least, our framework may offer support for researches in the direction of analyzing object-oriented software systems and developing various machine learning-based software engineering techniques (as shown in Subsection 2.2).

5. Conclusions and further work

In this paper we have introduced a software framework which has been developed for supporting several machine learning-based techniques that were already introduced in machine learning-based software engineering literature. The *FAOS* framework was used for developing several methods for software remodularization at the class and package level and for software design defect detection.

As further work, we would like to apply the *FAOS* framework for developing a machine learning-based solution for solving the *software cost estimation* problem, which is another important software engineering activity.We will focus on developing a plugin for Eclipse that is based on the interface described in this paper. Extensions of the framework in order to handle software systems written in other programming languages (e.g. C++, Python) will be further considered.

References

[1] ObjectWeb: Open Source Middleware. http://asm.objectweb.org/.
[2] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
[3] Gabriela Czibula, Zsuzsanna Marian, and István Gergely Czibula. Detecting software design defects using relational association rule mining. *Knowledge and Information Systems*, published online in January 2014.
[4] Istvn Gergely Czibula and Gabriela erban. Improving systems design using a clustering approach. *International Journal of Computer Science and Network Security (IJCSNS)*, 6(12):40–49, 2006.
[5] Commons dbutils. http://commons.apache.org/proper/commons-dbutils/index.html.
[6] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, and Andre Cavalcante Hora. Software metrics for package remodularization. Technical report, Institut National de Recherche en Informatique et en Automatique, 2011.
[7] Metrics plugin for eclipse. http://eclipse-metrics.sourceforge.net/.
[8] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, 1999.

[9] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, 2001.
[10] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, December 2012.
[11] Rui hua Chang, Xiaodong Mu, and Li Zhang. Software defect prediction using non-negative matrix factorization. *JSW*, 6(11):2114–2120, 2011.
[12] iplasma, 2013. http://loose.upt.ro/reengineering/research/iplasma.
[13] Jdeodorant, 2013. http://www.jdeodorant.com/.
[14] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *In Proceedings of the IEEE International Conference on Software Maintenance*, pages 50–59, 1999.
[15] Zsuzsanna Marian. Aggregated metrics guided software refactoring. In *Proceedings of the 8th IEEE International Conference on Intelligent Computer Communication and Processing*, pages 259–266, 2012.
[16] Zsuzsanna Marian. A study on hierarchical clustering based software restructuring. *Studia Universitatis Babeş-Bolyai Informatica*, LVII(2):20–31, 2012.
[17] Zsuzsanna Marian, Gabriela Czibula, and Istvan Gergely Czibula. Using software metrics for automatic software design improvement. *Studies in Informatics and Control (SIC)*, 21(3):249–258, 2012.
[18] Zsuzsanna Marian, Gabriela Czibula, and Istvan Gergely Czibula. Software packages refactoring using a hierarchical clustering-based approach. *Fundamenta Informaticae*, 2014. Under review.
[19] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Politechnica University Timisoara, Faculty of Automatics and Computer Science, 2002.
[20] Metrics plugin for eclipse. http://metrics.sourceforge.net/.
[21] Brian S. Mitchell and Spiros Mancoridis. On the evaluation of the bunch search-based software modularization algorithm. *Soft Comput.*, 12(1):77–93, 2008.
[22] Wei-Feng Pan, Bo Jiang, and Bing Li. Refactoring software packages via community detection in complex software networks. *International Journal of Automation and Computing*, 10(2):157–166, 2013.
[23] Kata Praditwong, Mark Harman, and Xin Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, 2011.

Department of Computer Science,, Faculty of Mathematics and Computer Science,, Babeş-Bolyai University, Kogălniceanu 1, Cluj-Napoca, 400084, Romania.

E-mail address: {marianzsu, gabis, istvanc}@cs.ubbcluj.ro