Archer: An Architectural Monitoring Tool

Vitor C. Alves, Rafael H.S. Rocha, Rodrigo de B. Paes, Evandro de B. Costa, Leandro Dias da Silva
Universidade Federal de Alagoas
Maceió, Brazil
{vitorcorreia.ufal, rafaelrocha.ufal, leandrodds, ebcosta}@gmail.com, rodrigo@ic.ufal.br

Gustavo R. De Carvalho
Pontífica Universidade Católica do Rio de Janeiro
Rio de Janeiro, Brazil
guga@les.inf.puc-rio.br

Abstract - Software Maintenance is a continuous process in software development that begins when the software is first released and does not end while the software is being used. This characteristic makes it one of the most expensive processes in software development. Software engineering has identified some factors that increase software maintenance costs and presented good practices to face these problems. Good software architectures make a software easier to maintain and to evolve. Several reference architectures have been defined. Nowadays, there are software tools that provide architectural discovery and documentation tools, but they do not effectively protect the architecture from being compromised. This paper presents a software architecture monitoring tool called Archer, which was implemented as an eclipse plug-in. This tool aids the programmers with respect to software architecture through identifying architectural flaws introduced when coding. Also, Archer supports discovering existing architecture from a software project by using reverse engineering techniques, providing the architect with information to improve, or do not compromise, the software architecture in existing software.

Keywords - Software Engineering; Software Architecture; Architectural Enforcement, Maintenance.

I. INTRODUCTION

Software maintenance is an activity that begins when the software is released and users start to use it. It corresponds by up to 80% of total software costs [1]. Software documentation is an important practice to maintain a software. It aids programmers in the understanding of how the software was designed and how changes can be made without compromise its structure. However, only the software documentation is not enough to guarantee protection to its logical structure, sometimes programmers do not obey, either deliberately or unintentionally, the software architecture and break it. This problem normally appears when the programming team changes, and no further explanation about the software structure and architecture is passed to the new employees.

The problem stated above suggests that it would be desirable to have a solution that helps the programmers in the understanding of legacy software. The solution should also enforce that architectural decisions will not be broken by the programmers, at least unintentionally. The software tool presented in this paper aims to fulfill both requirements: (i) to help in the understanding of already developed applications and (ii) to specify and enforce architectural styles [16]. The remaining of this paper is organized as follows. Section 2 describes some related work and compares them to Archer. Section 3 shows in details how Archer works. Section 4 illustrates the Archer though a case study. Finally, in Section 5, we conclude the paper discussing the contributions, limitations and further improvements of the current proposal.

II. RELATED WORK

ARCHJAVA [2] is a tool to recover software architecture on legated systems written in Java. Their goal is to be able to recover architectures documented in the literature, such as MVC (Model-View-Controller) [3] and Layers [3] by defining architectures as domain-independent rules. These rules are based on static [17] and dynamic [17,18] analysis. Static analysis enables the verification of software structure and dynamic analysis verifies the objects behavior. However, in contrast to Archer, ARCHJAVA is intended to be used only in java based software.

A hybrid computer aided approach for close monitoring source code by using this same static and dynamic analysis methods is presented in [15]. On this approach, the verification process analyses design-implementation congruence: concrete rules such as coding guidelines, architectural components, such as design patterns [10] or connectors [14], and design principles such as low coupling and high cohesion.

In Harris et al. [19], a language to request parsed information to analysis is described: the source code query language. This language allows programmers to recover information from an abstract syntax tree. The idea is very similar to the Archer architectural analyzer (Section 3.3), but since this query language interacts directly with the source code, the entire program will have to be rewritten to support a new source-code language. Archer, on the other hand, is prepared to support new languages without this kind of effort. This is possible because its architectural analyzer interacts only with the object oriented model (Section 3.2) which is language independent.

Another tool similar to Archer on its objectives is Dali [4], a workbench that aids the analyst to manipulate and interpret recovered architectural information. There are works such as LSME [5] and RMTool [6], but their scope is very similar to Dali. The main difference between these

 ISBN: 978-1-61208-105-2

works and Archer lies on the fact that they only work with legated systems, in other words, their concern is about recover software architecture to support the user in defining software architecture, but no further action is taken. Archer aims to use recovered information to protect existing architecture through the enforcement of architectural rules.

III. ARCHER

Archer is a plug-in that works integrated with Eclipse IDE [7]. It provides support for the software architecture enforcement and documentation. It is able to recognize architectural patterns in code and verify if a software is in agreement with a pre-established architecture. Its structure is divided in three parts: a parser, an object-oriented model and the architectural analyzer. Figure 1 illustrate the process of analysis.

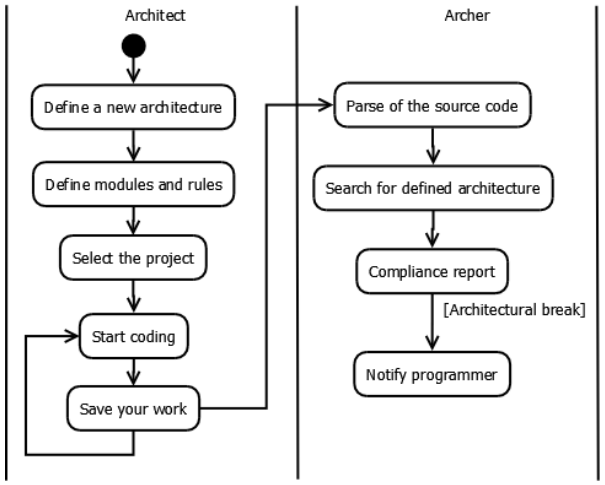

Figure 1. Archer process activity diagram

3.1 Parser

Through source-code analysis, all relevant information about the software is gathered by the parser. Examples of such pieces of information are classes, packages and relations between classes. This information is then organized on the Archer's object-oriented model. Although archer is designed to support the analysis of source code written in different programming languages, the current implementation works only with java. The parsing of the source code is made using the Eclipse/JDT Java Model [8] (Java Development Tools). Archer was designed for extension, then there are hotspots that may be extended for supporting parsers of other programming languages. Since all the information is stored into the archer's object-oriented model, archer can be used to analyze source code written in other programming languages, with no need of changes in the architectural analyzer. In this case it is necessary to change the parser. In other words, a new architecture analyzer (section 3.3) is not required to verify existing architecture patterns, just a new parser for other languages.

3.2 Object-Oriented Model

From the Archer point of view, the lowest abstraction level of a software's structure is its implementation (source code). A model represents this structure in a language-independent manner. It is composed of a set of elements which are present in object-oriented languages. Figure 2 shows the Archer meta-model. It is based on UML Meta-Model [9]. The main goal of this model is to represent the code structure. The model represents this throw in a set of objects which can be manipulated without language-dependent issues. The elements are defined in two main groups: relationships and named elements. Relationships represent connections between concrete elements. There are two relationships represented in the model: interface realization and inheritance ("Generalization").

The named elements are elements that have an identifier. They are defined in four main types: "Packageable Element", "Namespace", "Redefinable Element", and "Typed Element". The PackageableElement contains elements with a visibility type, e.g a class can be private, private is the visibility of the class. Namespace can contain other elements with names and can exclude equivalent elements within it. RedefinableElements are elements that can receive different values from other RedefinableElements that are equivalent or more specialized. TypedElement are elements that contains a "Type", i.e a primitive type or a class type. The "Project" is a "Package", since it can contain other packages as well as packageableElements. "Class" and "Interface" are "Classifiers". However, only "Class" is a type due to the reason that interfaces cannot be instantiated.

The Variable element is defined as a typed element only. Therefore, it cannot be analyzed in a redefinition context as a Property. However, in most cases it was not necessary to evaluate a constraint of an architecture. The Variable can be treated as the operations and properties since they are both "Typed Elements".

Figure 3 shows how a class could be graphically represented in the model in a simplified approach. It contains a generalization relationship, generalization contains oval and drawable component. Generalization is a directed relationship. Therefore, it has a source component and a target component.

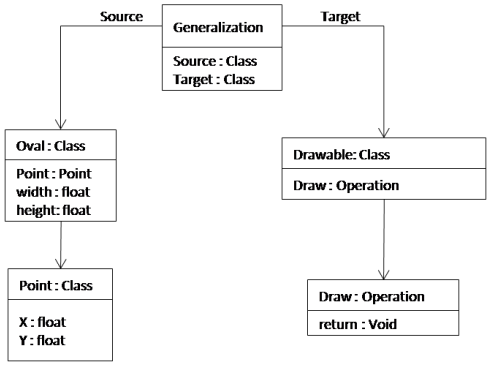

Figure 3. A Simple Class Representation in the Model

In the illustrated case, the oval class is the source component and drawable, the target. This means that oval inherits drawable component. Drawable has a draw operation, the oval component has a point and two float attributes and the point has two float attributes.

Usually, a software architecture is separated in modules. These modules can often be separated in smaller units, such as design patterns. These units are abstractions that commonly represent implementation constraints. The architectural analyzer contains modules composed of rules for representing these smaller units. The rules search for code samples implementing the constraints, e.g., Singletons [10], have private constructors. Therefore, a rule should search in classes for constructor methods with private visibility.

The tool use the rule constraints implemented in terms of the model components. The Analyzer retrieves classes from the code analyzed and tests them for each rule. Since the code was parsed to be represented in components of the model, comparison is possible. A matching percentage is given as evaluation result for a module. Each rule counts a point and the sum is divided by the total of rules from this module.

3.3 Architectural Analyzer

Having all the information required, the analysis process may start. Archer works with two important concepts for its operation: code rules and architectural patterns.

3.3.1 Code Rules

Code rules are built from the information contained in the object oriented model database. These rules are patterns found in source code. An example of a code rule is the verification of all classes that implements an observer. Code rules are used to search for architectural patterns. On Archer, these rules are implemented as a class using the Java language.

In the current version, Archer contains the patterns: Command, Singleton, Observer, Abstract Factory, Bridge. Frequently, these patterns are key elements for defining an architecture. Archer has also more practical rules, such as the detection of graphic objects, database access and event controllers (listeners). The problem of verifying the existence of patterns in source code was already studied in [21].

3.3.2 Architectural Patterns

Architectural patterns are known solutions that work efficiently to solve an architectural problem. It defines how classes interacts and sets how the software information flows in runtime [11]. In practice, it also defines some characteristics that classes should have to be part of an architectural structure such as a layer, for example. On Archer, we define an architectural pattern as a group of these characteristics code rules.

Archer analysis process starts everytime programmers save their work, by searching every class of the project for code rules. When the process is finished a list of found code rules is obtained, and then a process of comparison is made to check if the classes have any similarity with one of the architectural patterns contained in the database. This process will give a percentage, indicating the chance of a class to belong to a specific architectural structure from an architectural pattern. The higher the percentage is, better are the chances of the class to belong to that structure.

An important feature of Archer is that the architect can make his own architecture from existing code rules. This is possible because of its generic analyzer. When the information is loaded from the object-oriented model, it organizes the information on a matrix. The information contained in that matrix informs which classes contain which code rules. Then a process of finding similar classes on the matrix is performed in order to find possible members of the same architectural modules. Finally, an analysis is made to identify the architecture itself, based on the characteristics found on these architectural modules.

After the analysis process is done, the results are saved and programmers will do their work as usual, but anytime the programmers team would otherwise compromise the architecture, Archer will send an alert warning about unwanted changes.

The process of enforcement is completed after the analysis process. If a programmer tries to break the architecture, Archer will analyze its database and check if the change is harmful to the architecture. If it is, Archer will discourage the programmer to continue with his changes. That way, the architecture will be safe. For example, if a programmer tries to put graphic objects on the model abstraction of the MVC architectural pattern, Archer will warn about unwanted code rules on this abstraction (model does not implement graphical objects). This way, an act that would otherwise compromise the architecture will be prevented.

IV. CASE STUDY

To illustrate Archer's functionality, a simple calculator and an artificial intelligence simulation software made as a chess game [20] were analyzed under the MVC Architecture [3]. In this section, we present how Archer was used to match existent software architecture to a previously defined architecture.

The calculator is separated in three main packages, the Model is represented by the Calculator class which contains the manipulated data and it is observed by the view's component. The View is represented by the Window class which contains graphical components and observers the model. The Command component is represented by the listener classes, the ActionListener acts as an Observer interface. Figure 4 shows a simplified version of the UML representation of the calculator.

Archer was setup to use the MVC Architecture. It means that it will verify whether the source code of the application is in conformance with the MVC. The architecture was defined in three modules (model, view, control). Each

 ISBN: 978-1-61208-105-2

module implemented it own rules (see Section 3.3). The code parser retrieves the code from the selected project and parses it into objects of the object oriented model (see Section 3.2).

The process of analysis is a sequential search for correspondence with the rules defined in the architecture. It is made with a binary vector that stores the rules needed to define each component of the architecture design. Figure 5 illustrates part of the evaluation process.

The Analyzer verify if a component obey(x) or disobey (-) a rule. In the illustrated case AddListener follows the same rules that define the control component. After the analysis is made, the tool compares the results of each class with the modules of an architecture attributing a percentage of correlation.

Each attribute of a class is already in the model at evaluation time. Therefore, if the tool needs to analyze an attribute of a class before analyze the class itself, it is possible. This feature allows the tool to evaluate components that are defined in terms of others, e.g the classes of view component contain classes from control component.

An architecture is defined in the module "Architectural Composer" each composer contain "Architectural modules" and each module a set of "Rules". A rule is defined as a boolean function which gets a "Class" from the model and evaluate it. The function is defined as a implementation constraints. The complete project structure is available for a rule function. However, it evaluates a class per time. If an attribute of a class must be evaluated before it reaches a conclusive response, the rule pass the attribute to the analyzer to it be evaluated first.

The results of the analysis show the percentage of compatibility of each class with the modules of the architecture as illustrated in Figure 6. Although the results were acceptable, they could be improved.

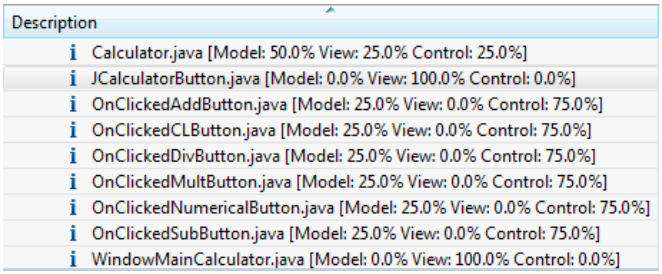

Figure 6. Calculator Evaluation

For the JChess, Archer concluded that the software did not followed the MVC architecture, Figure 7 shows the UML representation of the application.

Archer was used to analyze the architecture of this application according to the MVC Architecture, Figure 8 show the results. Some classes implemented the Model of MVC, e.g. Move and PGN which reaches high compatibility with this module, these classes contains data information that is accessed by the view module, JChessBoard contains graphical interfaces and modifies the data of model classes directly so it is unevenly distributed over view and control. Most of the classes are not well defined, it shows the inconsistency of the application with this architectural pattern.

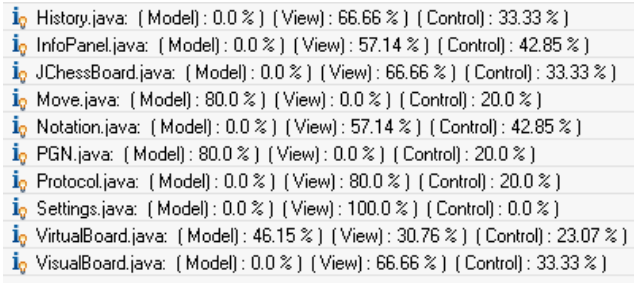

Figure 8. JChess Evaluation

V. CONCLUSION

It was developed a parser that reads java codes and generates a language-independent OO model was developed. It allows to represent the collected data without loss of information and it is free of language details. The permits analyze the code of a project by comparing a set of classes to a design template.

Archer can define rules at a low level of abstraction. In the current version, these rules are created by programming in java language using model's classes provided by Archer API. We are aware that there is a need to define the architecture in a higher abstraction level. We have already tried to represent these architectural rules in ADLs (Architectural Description Language) such as ACME [12] and Wright [13], however, they do not allow us to express the level of details it is needed to perform the enforcement of the architecture.

Archer has some features integrated to eclipse. It can notify the developers if some architecture has a potential problem. At the moment they save their code, the evaluation appears in the Eclipse problems panel. As future work, the tool will be integrated with subversion version control system. This feature would allow verifying whether an architectural rule is broken at the moment developers commit their code. It is being studied a way to represent the architecture through an ADL. The idea is to follow a bottom up approach, i.e., in the current version, the architecture must be defined programmatically using the Archer API. Each architecture defined is included in the architecture database. Once the database becomes bigger, it will be possible to reuse code rules of previously defined architectures. Then, as the level of reuse increases, it will be possible to create a language to represent these code rules.

ACKNOWLEDGMENT

This work is partially supported by CNPq/Brazil under the project PROCOCO (620063/2008-4)

 ISBN: 978-1-61208-105-2

REFERENCES

[1] Ambler,S.W.(1997), Análise e projeto orientados a objetos, Infobook, Rio de Janeiro.

[2] Carvalho F., Barroso L, Seufitele V., Vasconcelos A. ArchJava: Reconhecimento de padrões arquiteturais em sistemas Java, CEFET CAMPOS.

[3] Buschmann, Frank et al. Pattern-Oriented Software Architecture: a System of Patterns. Wiley, 1996.

[4] Kazman, R. and Carrière, S. J. 1999. Playing "Detective: Reconstructing Software Architecture from Available Evidence". Automated Software Eng. 6, 2 (Apr. 1999), pp. 107-138.

[5] Murphy G., Notkin D. Lightweight Lexical Source Model Extraction. In ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 3, July 1996, pp. 262-292.

[6] Murphy, G., Notkin, D., and Sullivan, K. "Software Reflexion Models: Bridging the Gap between Source and High-Level Models". in Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, (Washington, D.C.), October 1995. pp. 18-28.

[7] Eclipse, http://www.eclipse.org/ 04.14.2010

[8] JDT, "Java Development Tools", http://eclipse.org/jdt/ 04.14.2010

[9] Unified Modeling Language, http://www.uml.org/ 01.29.2010

[10] Gamma, E., Helm, H., Johnson R., Vlissides, M. J. "DesignPatterns: Elements of Reusable Object-Oriented Software."

[11] Sommerville, I. "Engenharia de Software" Addison Wesley 8ª ed.

[12] Garlan, D., Monroe, R. T., and Wile, D. 2000. "Acme: architectural description of component-based systems". In Foundations of Component-Based Systems, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, New York, NY, pp. 47-67.

[13] Allen, Robert J. "A Formal Approach to Software Architecture" (Ph.D. Thesis, CMU-CS-97-144 ed.). Carnegie Mellon University. (May 1997).

[14] Shaw M., DeLine R., and Zelensnik G. "Abstractions and Implementations for Architectural Connections". Technical Report CMU-CS. pp. 95-136, CMU, March 1995.

[15] Sefika, M., Sane, A., and Campbell, R. H. 1996. "Monitoring compliance of a software system with its high-level design models". In Proceedings of the 18th international Conference on Software Engineering (Berlin, Germany, March 25 - 29, 1996). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, pp. 387-396.

[16] Abowd, G., Allen, R., and Garlan, D. 1993. "Using style to understand descriptions of software architecture". In Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering (Los Angeles, California, United States, December 08 - 10, 1993). D. Notkin, Ed. SIGSOFT '93. ACM, New York, NY, pp. 9-20.

[17] Olshefski D. P. and Code A. "A Prototype System For Static and Dynamic Program Understanding". In Proceedings of the Working Conference in Reverse Engineering, Baltimore, MD, USA, 1993. pp. 93 - 106.

[18] Ritsch H. and Sneed H. M. "Reverse Engineering Via Dynamic Program Analysis". In Proceedings of the Working Conference in Reverse Engineering, Los Alamitos, CA, USA, 1993.

[19] Harris, D. R., Reubenstein, H. B., and Yeh, A. S. 1995. "Reverse engineering to the architectural level". In Proceedings of the 17th international Conference on Software Engineering (Seattle, Washington, United States, April 24 - 28, 1995). ICSE '95. ACM, New York, NY, pp. 186-195.

[20] JChess, http://jchessboard.sourceforge.net/ 20/08/2010.

[21] Blewitt, A., Bundy, A., and Stark, I, "Automatic Verification of Design Patterns in Java". In ASE 2005: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, California, USA, November 7–11, 2005, pages 224–232. ACM Press, 2005.

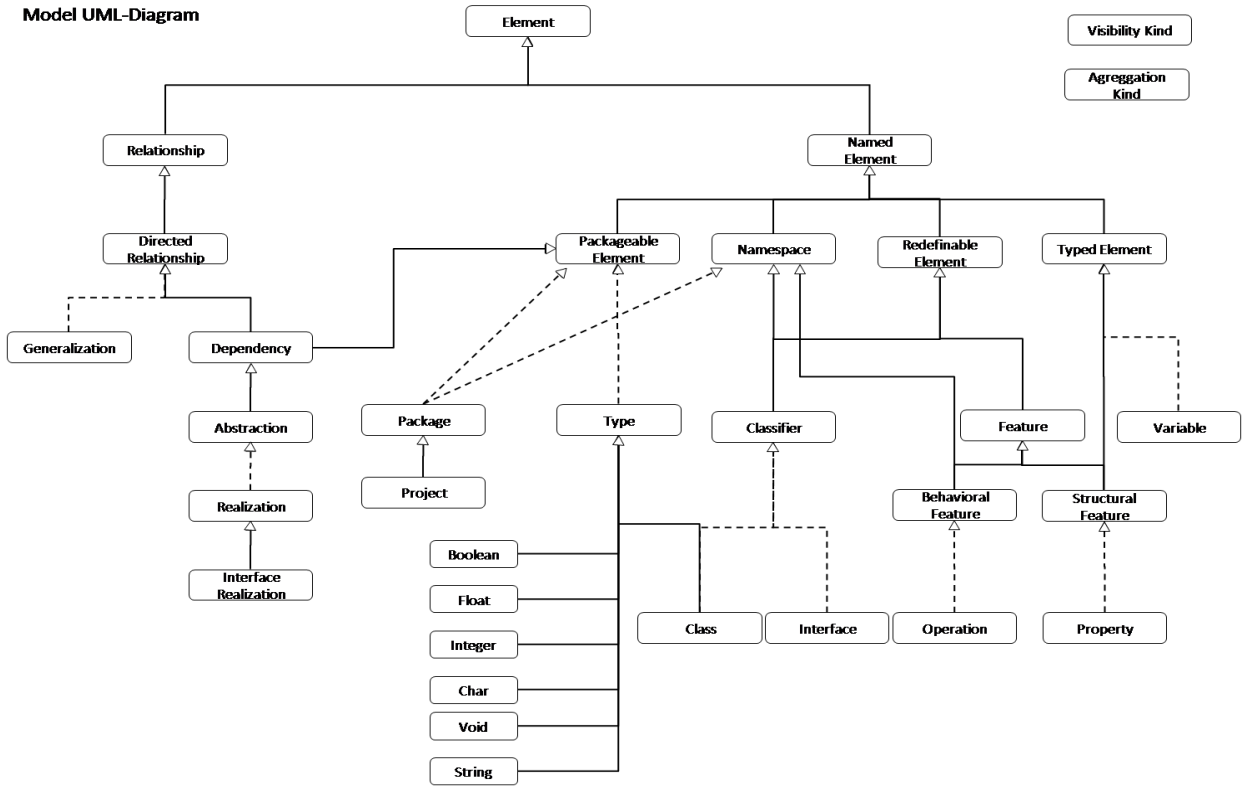

Figure 2. Simplified Representation of Archer Meta-Model

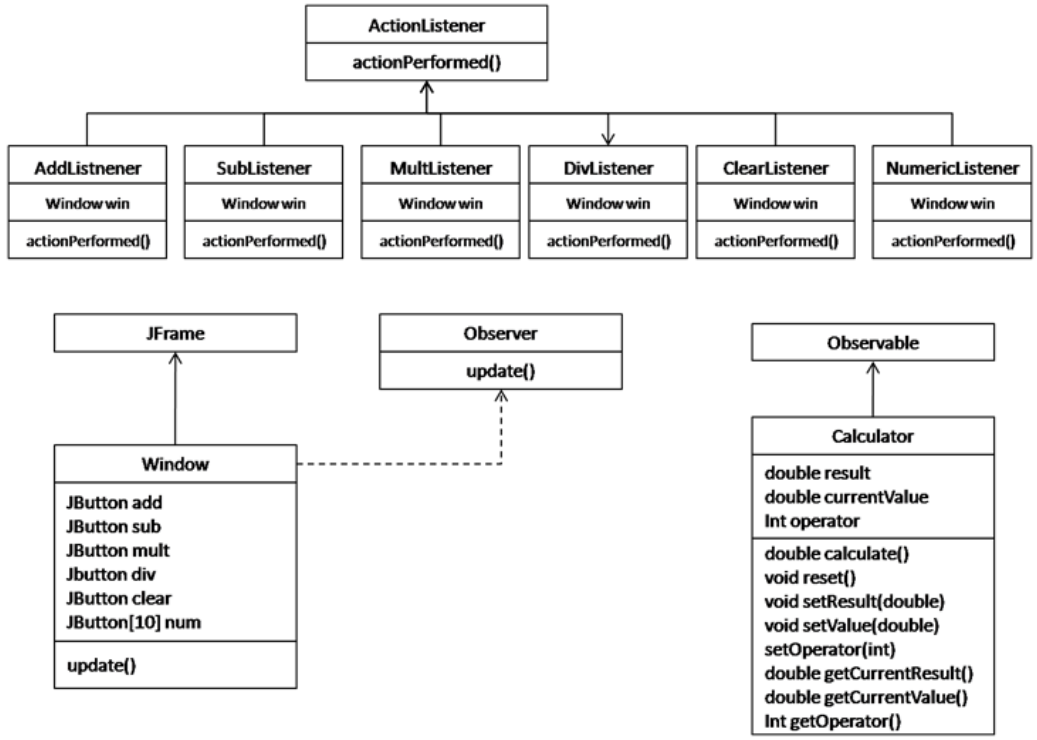

Figure 4. MVC Calculator Sample UML Description

rules / classes	Observed	Observer	Command	Graphical ref.	Contain Model component	Contain Control component	...
AddListener		X	X	-	X		
SubListener		X	X	-	X		
Window				X			
Calculator	X			-			
...							

rules / mvc	Observed	Observer	Command	Graphical ref.	Contain Model component	Contain Control component	...
Model	X			-			
View		X		X		X	
Control		X	X	-	X		

Figure 5. The evaluation process

 ISBN: 978-1-61208-105-2

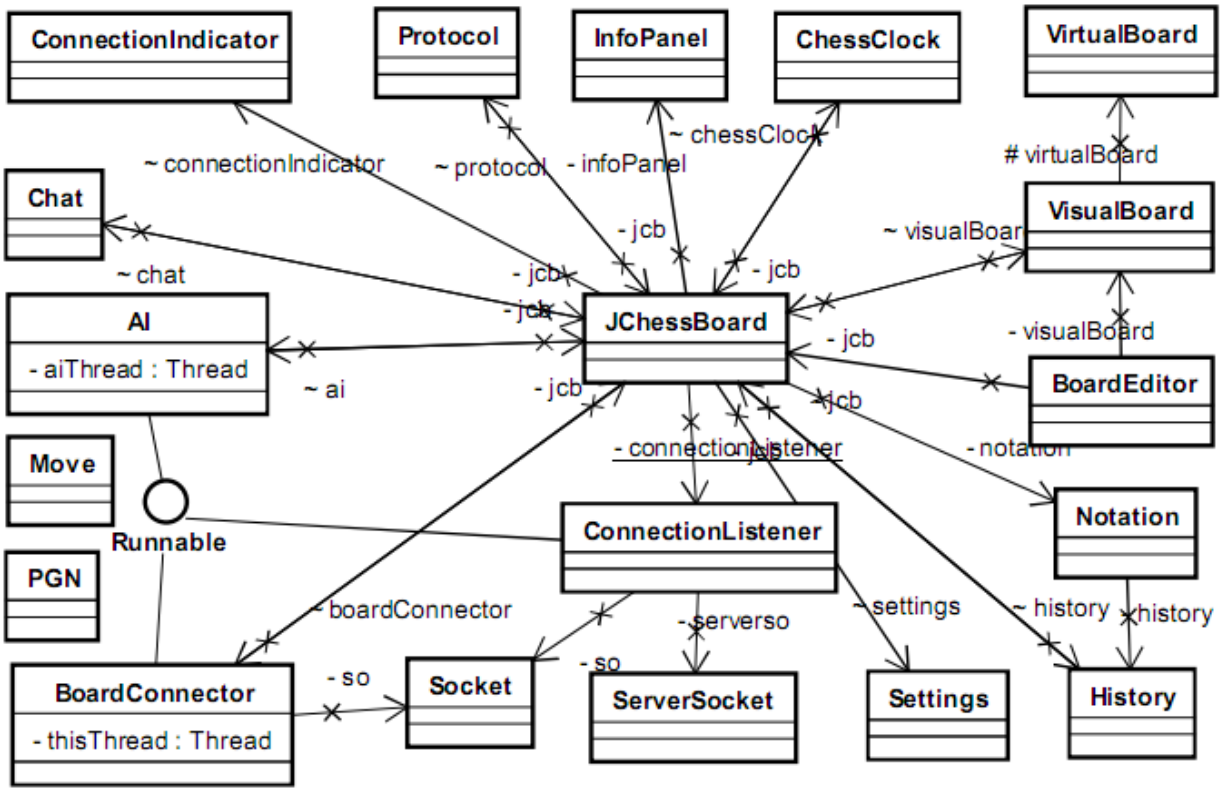

Figure 7. JChess UML Description

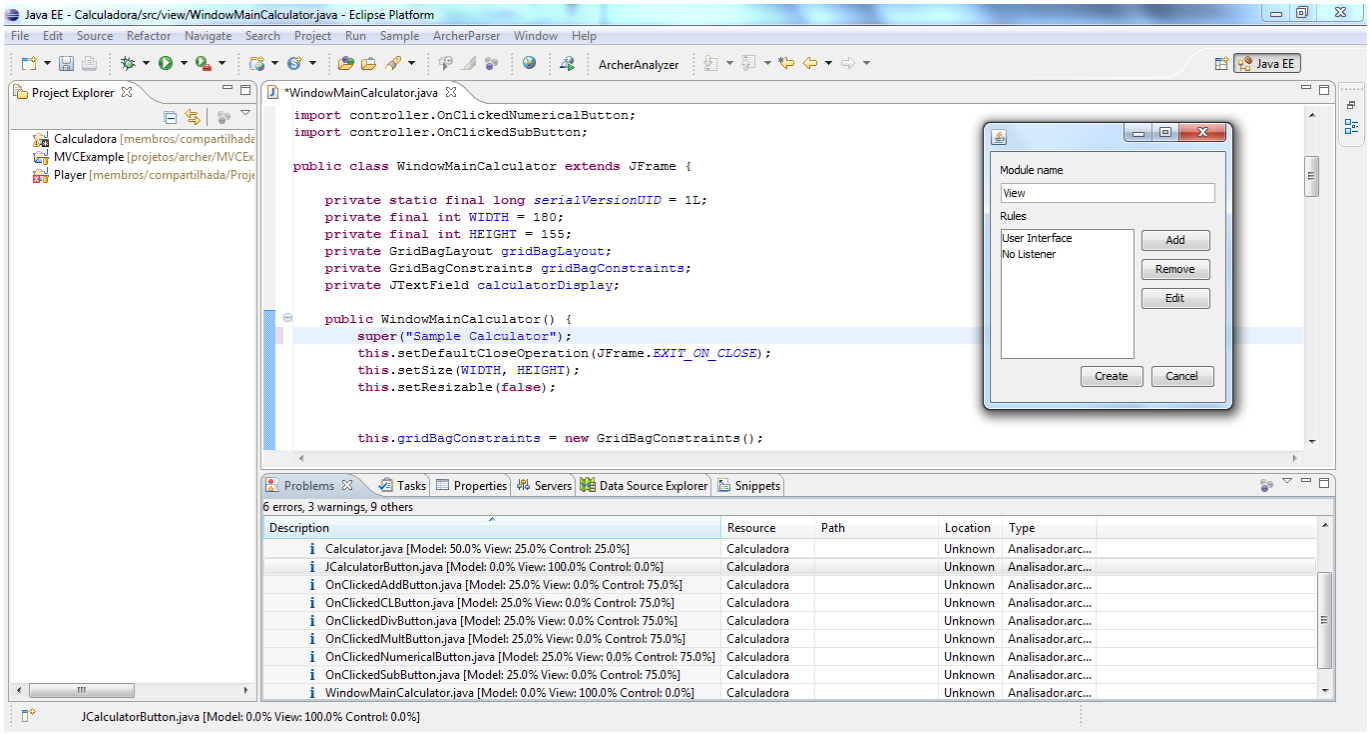

Figure 9. The Archer plug-in