Source Code Documentation: a Tool Focused on Business Requirements

Humberto Ferreira da Luz Junior
Departamento de Computação
Universidade Estadual de Londrina
Londrina, Brazil
hfluz@uel.br

Rodolfo Miranda de Barros
Departamento de Computação
Universidade Estadual de Londrina
Londrina, Brazil
rodolfo@uel.br

***Abstract*—Aiming to fill a gap in traditional methods of source code documentation, which focus mainly on the API documentation for other programmers, this article presents a new approach for business requirements, mapping them through a set of annotations. These annotations, in turn, are interpreted by the GaiaDoc tool, which is specified in this paper and is able to generate documentation in form of use case specifications in a language and format easily understandable by the project stakeholders. A case study of the proposed methodology's application is presented before the final considerations.**

***Keywords*-Source code documentation; requirements engineering; javadoc; documentation generator;**

I. Introduction

The source code documentation (SCD) is considered one of the best practices for any programming language. It allows other developers to learn to use the code without the need to analyze hundreds or even thousands of lines of complex algorithms and it ensures that the knowledge of its rules does not get stored only in people.

Almost all practices for this type of documentation focus on API (Application Programming Interface) users, i.e., other programmers, who generally adopt an overly technical language, and often simply describe an algorithm that should be readable only by its own code.

In RUP (Rational Unified Process), the business rules of software projects are specified primarily as functional requisites in the use case model, that represents interactions between the actors and the system, and the detailed descriptions of each of these use cases, which are defined as use case specifications (UCS) [5].

The problem lies in the fact that the specifications for code documentation, as well as their tools, were created especially for API documentation, leading programmers to document incorrectly the classes which implement business rules.

A clear example of classes that implement business rules resides in the controller layer of the Model-View-Controller (MVC) architecture, a standard architectural design developed which proposes that business rules should be decoupled from the vision and the application domain and is therefore a three-tier architecture [2]. Accordingly is quite possible to link the controller layer classes to the use cases of Unified Modeling Language (UML).

It is also trivial to find cases where there is little integrity and traceability between the business rule documentation, the SCD and the implementation itself. This occurs because the requirements are frequently changed, making the source code updated, and for lack of time, the respective requirements documented in the UCS come to be neglected [3].

Consequently, the current version of the implementation is rarely equivalent to the documentation, since it demands too much effort to keep it up to date. To address this issue, several tools have emerged to concentrate the documentation in the source code and export it to other formats which are more accessible to their readers.

Considering the above problems, it is proposed in this paper a new approach for code documentation focusing only on the business rules. This approach is given by its own set of annotations mapped to UCSs and is available through a language and format easily understandable to the project stakeholders.

This paper also presents GaiaDoc tool, responsible for mapping the annotations in the UCS, and some advantages by storing the documentation in the source code. Among the advantages, there is the versioning, which can be accomplished through a configuration management system such as Subversion or GIT, and the bug tracking on requirements, which can be obtained by bug tracking systems, such as Bugzilla or JIRA.

This approach makes it possible to maintain the traceability and integrity of requirements in relation to its implementation, which ensures that all requirements that are understood by the requirements analyst and approved by the client are present in the final product, besides preventing that unnecessary features are constructed [8].

II. Methodology

The main objectives of the proposed methodology are: (i) to fill a gap in SCD methods currently used in classes that represent the business rules and to avoid those requirements duplication among the project's artifacts, resulting in less effort to maintenance; and (ii) greater traceability between requirements, implementation and product delivered to the customer.

Since the annotations present in the source code require well defined rules, the next step was to determine what should be the resulting requirements document. Due to the popularity of UML in the requirements elicitation for software development projects, use cases and their specifications have been estabilished as key artifacts for the proposed approach.

Consequently, although UML is not attached to a specific SPD, it was natural to adopt the RUP's requirements discipline, whereas this process was specifically developed using the UML.

A. Mapping Between the SCD and the UCS

The mapping between the code documentation of classes and the UCS is done using annotations within comment blocks of the Java language, similar to Javadoc.

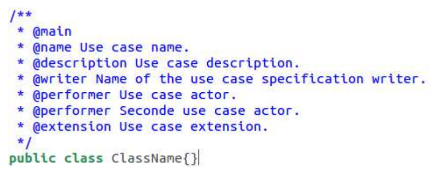
Annotations located in blocks preceded by /** and closed by */ are analyzed by the tool, as well as by Javadoc. Any comment block contained by /* and */ and line comments preceded by // is ignored.

The following rules are defined for the mapping in class scope:

1) Every class that represents business rule requirements must contain the *@name* annotation in its comment block, indication from which use case it was mapped;
2) Whenever a use case is mapped to more than one class, one must contain the *@main* annotation;
3) The annotation *@description* is required when the use case is mapped to only one class. If the use case is mapped to different classes, at least one of them must contain this annotation, and if more than one class have the description, then the one which class is annotated with *@main* will be adopted in the resulting UCS;
4) The *@writer* and *@performer* annotations are mandatory and can be repeated several times, if the documentation is written for more than one person or there is more than one actor in the use case;
5) In the class scope there is also the annotations *@extension*, which represents an extension of the use case, and *@specialRequirement*, that represents the use case special requirements;
6) In the case of the *@writer*, *@performer*, *@extension* and *@specialRequirement* annotations, when the use case is mapped in several classes, these properties are merged.

The figure 1 shows the model of documentation in class scope.

The class attributes, together with their description, are used to generate the Glossary, which is a dictionary of terms related to the use case in form of a table. The following rules are adopted for this scope:

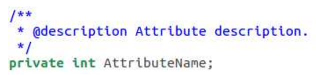


```
/**
 * @main
 * @name Use case name.
 * @description Use case description.
 * @writer Name of the use case specification writer.
 * @performer Use case actor.
 * @performer Seconde use case actor.
 * @extension Use case extension.
 */
public class ClassName{}
```

Figure 1. Model of documentation in the class comment block.

1) Only the *@description* annotation is allowed. In case this annotation is not present, the attribute is ignored during the generation of the UCS;
2) The attribute name is obtained from the variable name to which belongs its comment block.

The model of documentation in the attribute scope is exemplified by the figure 2.



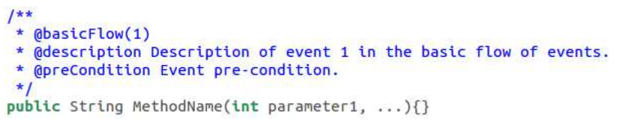
```
/**
 * @description Attribute description.
 */
private int AttributeName;
```

Figure 2. Model of documentation in the attribute comment block.

The class methods represent the event streams of the use case. An event stream can be divided into different classes, provided that all belong to the same use case. The rules for its comment block are listed below:

1) The method comment block should contain the annotation *@description*, which cannot be repeated in the same block;
2) By default it is considered that the event actor is the one defined by the *@performer* annotation in the class scope. If the same annotation is present in the scope of the method, this value will be overwritten;
3) If the event has pre or post-conditions, the annotations @preCondition and @postCondition can be used as many times as necessary;
4) It is mandatory in the method scope that the *@basicFlow* or *@alternativeFlow* annotation be present. However both cannot be used together;
5) The annotation *@alternativeFlow* has a parameter indication to which alternative event stream the method belongs.

According to the delimitations specified above, the model for method documentation is shown in the figure 3.



```
/**
 * @basicFlow(1)
 * @description Description of event 1 in the basic flow of events.
 * @preCondition Event pre-condition.
 */
public String MethodName(int parameter1, ...){}
```

Figure 3. Model of documentation in the method comment block.

After reading the annotations in the file that contains the class, GaiaDoc converts these data into the class structure shown in figure 4.

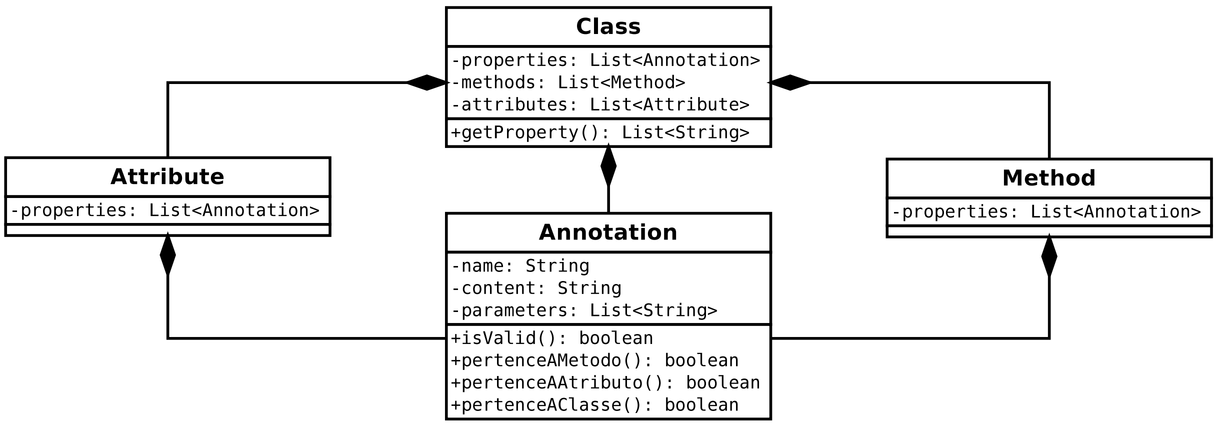

Class
-properties: List<Annotation>
-methods: List<Method>
-attributes: List<Attribute>
+getProperty(): List<String>
Attribute
-properties: List<Annotation>
Method
-properties: List<Annotation>
Annotation
-name: String
-content: String
-parameters: List<String>
+isValid(): boolean
+pertenceAMetodo(): boolean
+pertenceAAtributo(): boolean
+pertenceAClasse(): boolean

Figure 4. Class diagram of the structure that represents the data read from the Java file by the GaiaDoc tool

The objects of the class *Annotation* contain the data for each annotation individually, as its name, its content and its parameters, if applicable. Furthermore it provides methods that help to identify its scope and also in checking its validity.

The *Class* stores a set of properties relative to the class scope, which are in fact objects of type annotation, which belong to its scope. It also has a set of methods and attributes. Each element of the set of methods is an object of type *Method* that stores the properties related to its scope. The same occurs with the set of objects of type *Attribute* that contains the annotations associated with the attribute.

B. The GaiaDoc Tool

The GaiaDoc tool was designed to be executed via command line, taking two parameters: the first is the root directory that contains all classes to be analyzed and the second is the output directory, which is where the UCSs generated from the class annotations.

There are also optional parameters to remove specific parts of the UCS such as, for example, the glossary. In the case you should add the –r glossary parameter.

The PDF was chosen as the output format in order to hinder the changes made in the generated document, encouraging the changes directly in the source code.

By the time the command is executed, the tool traverses all the files in the given directory, including those in its subdirectories. Each found class is parsed to determine whether it contains all the required annotations and, if so, the documentation is obtained from the file and it is used to generate the UCS.

During the execution of the parsing, the annotations are located in the comment blocks and classified as tokens by the parser. The reading of the file that contains the class must occur in the sequence outlined by the state diagram shown in figure 5.

If the parser does not follow the provided flow by the states diagram, the documentation is considered invalid and the UCS is not generated for the class.

When changes occur in the requirements documentation in the source code, you need only re-execute the command to overwrite the generated specifications.

C. Requirements Flow

Despite the use case modeling does not limit the choice of the Software Development Process (SDP), it is often used in conjunction with RUP, which is interactive, incremental and oriented to planning and risks, and focused on the architecture of the system to be developed [1].

The adoption of this approach requires some changes in RUP's default requirements pattern, which is handled by the Requirements Discipline.

The first artifact generated by this discipline is the vision document, which sets out what is the problem to be solved by the software system, what is the delimitation of the scope, as well as who are the people interested in the project.

The requirements are then identified through interviews with stakeholders and classified into functional, that are specified by use cases, and non-functional, which are documented as a supplementary specification.

Following the traditional modeling of the RUP, the functional requirements, represented by UCS in a language understandable to all stakeholders in the project, are refined. This specification is usually available in text format document, including the description of the use case, the list of actors who participate in it and the basic and alternative flow of events.

Although not in the RUP scope, subsequently much of the already documented requirements are duplicated in the SCD, increasing the effort to keep both updated with the latest version of the code.

For this reason, the approach of the proposed methodology in this paper differs from the traditional one recommended by RUP after the definition of the use case diagram. Following the identification of use cases, a preliminary class

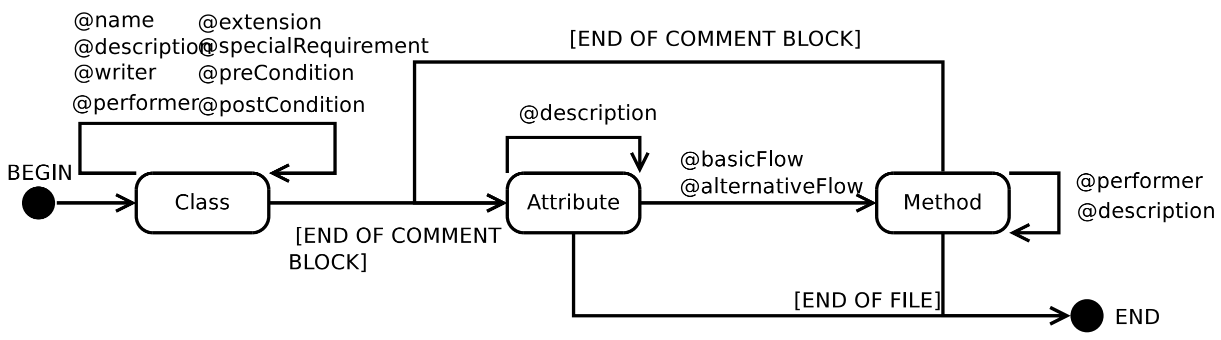


Figure 5. State diagram of the GaiaDoc parser.

diagram is defined and then a skeleton project is created, without their specification in text documents.

The skeleton of the project includes the specified class in the class diagram, its attributes and methods, as well as the documentation suggested by this article.

The act of creating the skeleton of the project improves the understanding of project requirements by the analysts, and as a consequence, the class diagram and other UML artifacts are refined. These improvements are propagated back to the skeleton and the class diagram, generating, consequently, a continuous improvement process.

After the implementation of the skeleton of the project, the classes are ready to be documented by the requirements analyst through the use of the previously described annotations. When the documentation is finished, specifications are then generated in PDF format and distributed to stakeholders.

At the start of system's implementation, most of the requirements are already documented in the source code. Throughout the development of the project, the requirements are iteratively refined, modified and enhanced, since changes have been approved by the change management committee. As a consequence, all artifacts can be modified, including the use cases, the class diagram, the UCSs located in the source code and even the code itself.

The fixes of bugs (failures) identified in the documentation can be registered in a bug tracking system and attached to a specific version of the UCS through versioning, helping to keep track of changes in documented requirements and also defining the priorities of these modifications in relation to corrections made in the source code itself. Thus, a centralized repository of all changes made in both the code and documentation is maintained.

Nowadays versioning is almost always applied to source code and bug tracking systems are aways present in software development projects, accordingly the addition of this resource to requirements management would represent little to no cost for the organization.

The result of the described practices is a code very well documented, updated and in line with the other project artifacts.

Although RUP does not address the SCD in real projects, it is common that the documentation requirements are duplicated, even in different languages and levels of depth, and therefore the update of requirements becomes more costly to the organization.

If the organization chooses to document an artifact over another, with the aim of reducing costs, the integrity of the artifacts is broken.

In the proposed method, the cost of updating the artifacts that contain the requirements of the project is reduced, since the specification of functional requirements is concentrated in one place. The requirements analyst updates the documentation located in the source code and uses the GaiaDoc tool to regenerate the specification of use case in its updated version.

The disadvantage of this approach, as well as various other techniques of software engineering, is that the duration of the phases of analysis and planning becomes larger. Nevertheless, it is expected that the duration of subsequent phases is considerably reduced, generating a positive cost-benefit for the project.

1) Traceability of Requirements: According to Lamsweerde [7] the overall objective of traceability management is to support consistency maintenance in the presence of changes, by ensuring that the impact of changes is easily localizable for change assessment and propagation.

Revisions result from evolution over time to improve or correct the document and should contain the date of modification, their author, their contributors and the reason of the change. In relation to the proposed approach, the contributors and the reason of the change can be managed by a bug tracking system (adapted to be used a a requirement tracking system). All other informations related to a new UCS version are easily managed by a versioning tool as mentioned before.

Besides the versioning management of the requirements documents, the dependency between the requirements contained in each artifact should be tracked. In the scope of this paper a vertical traceability link can be found between: (i) the class diagram and the project skeleton; (ii) the use case diagram and the UCS; and (iii) the UCS and the source code. A horizontal traceability link can be found between the project skeleton and the UCS.

The advantage of this proposal is that the traceability between those artifacts can be easily identified, what is very important considering that specially in the beginning of the specification many changes might be made in the project skeleton while the requirements are being better understood.

2) Relationship between the documentation writer and the developer: Some rules must be established in the relationship between the programmer and writer of the documentation - usually, the requirements analyst.

The developer can and should make simple comments (preceded by //), which are not mapped by the documentation generator tools, while the writer can freely edit the documentation located in the source code, which is used to generate the UCSs, however, he should not, under any circumstances, modify the code itself.

As for the documentation of classes mapped by Javadoc, which is used by other developers as an API, it consists of a strategic decision of the organization if the documentation should be written by the programmer or analyst requirements. Leslie [6] recommends that in cases where the requirements are technical, the developer should be allowed to write comments, however, the requirements analyst can also edit and complement them.

Developers can and should review the documentation written by the requirements analyst, however they should propose changes through a change management system, such as a bug tracking system, which fits perfectly to the situation, considering that the documentation is present in the code source. Approved changes are, then, made by the requirements analyst.

III. Related Works

Related works that address the automated generation of business requirement documents from source code could not be found. The closest approach was related to tools that get the non-business requirements from source code to generate API documentation for software developers, like Javadoc.

IV. Case Study

The case study was applied to the project of the Dental Clinic Management System of the Londrina State University, developed by the Department of Computer Science at the same university between the years of 2009 and 2011.

The project team consisted of four undergraduate students, two MSc students and two teachers who worked in project management. The adopted process was RUP and the standard code documentation chosen was Javadoc. Yet no standard on *how* the requirements should be written or validated have been set.

The project was implemented in the J2EE architecture, following the MVC pattern. We identified 12 core classes that implement the business rule and through their analysis was possible to note differences in the level of detail of each class comments:

- Very well commented (on average 3-5 comment lines per method): 4 classes;
- Commented (on average 1-2 comment lines per method): 6 classes;
- Not commented: 2 classes.

It was observed that the simpler the business rule implemented, the less likely the class contained documentation describing the class itself and its methods. When the complexity increased, the average number of lines of comment per method is proportionally raised.

Following, the documentation contained in the source code was compared with the UCS, seeking overlapping of requirements, which leads to a greater effort to maintenance and higher probability of requirements inconsistency. Classes commented in more details were those which had more duplicate requirements in relation to its respective UCS. Classes with few comments contained little or no overlap, but it was identified that in many cases only the documentation of the source code was not enough to understand their business rules without examining the source code itself or the UCS document.

Subsequently the identified use cases and business rule classes of the project were correlated. Only one use case was mapped into two classes, each of all the others have been mapped in only one class.

The classes were re-documented following the GaiaDoc approach taking care to keep all information contained in the UCS. The figure 6 shows the class AppointmentRegister documented and figure 7 presents its respective UCS generated by the GaiaDoc tool.

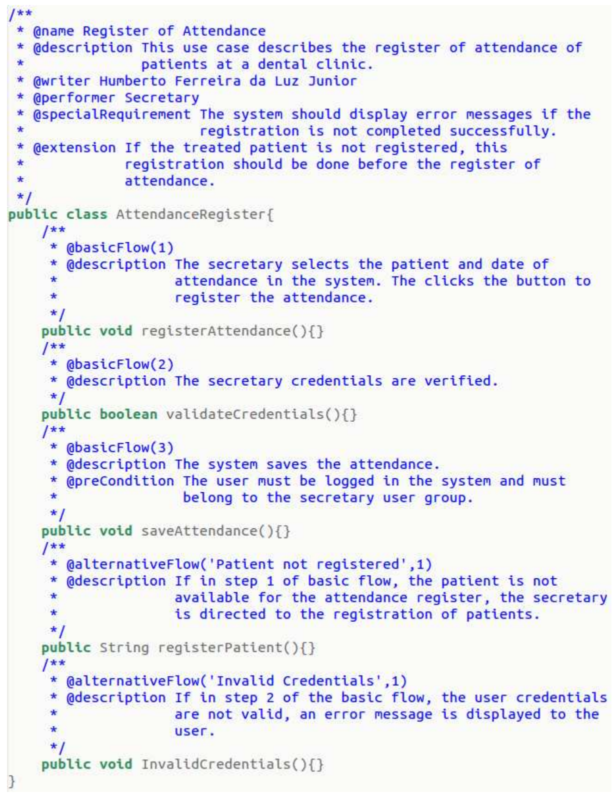
Since GaiaDoc in its current version does not support the addition of pictures, use case diagrams contained within the UCSs could not be included in the version generated by the tool. So this was the only missing information identified in the re-documentation of the classes.

Whereas GaiaDoc was applied to a project already in place, the study was concerned only in verifying the efficiency of the proposal related to requirements duplication in the source code and UCS, as well in the assurance that the requirements are consistent with the project artifacts.

The next step consists in the application of GaiaDoc in a project since its inception and the validation of the requirements process proposed in this work, as well as to analyze the relationship between the requirements analyst and developer in a practical way.

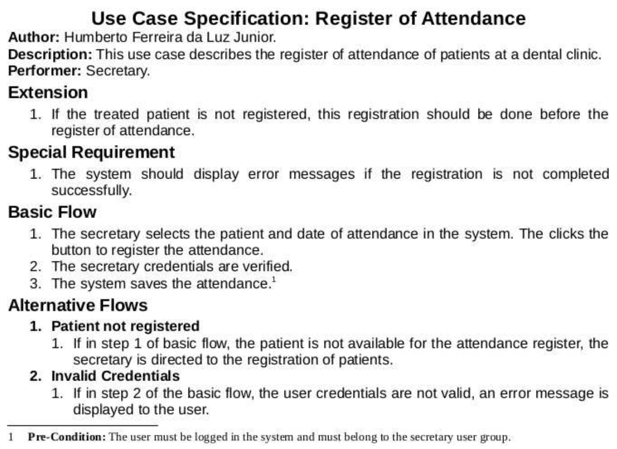
V. Conclusion

It was suggested a new perspective on the RUP's requirements flow through the use of a new tool that generates the UCS in an automated way, from the SCD using annotations.



```
/**
 * @name Register of Attendance
 * @description This use case describes the register of attendance of
 *              patients at a dental clinic.
 * @writer Humberto Ferreira da Luz Junior
 * @performer Secretary
 * @specialRequirement The system should display error messages if the
 *                     registration is not completed successfully.
 * @extension If the treated patient is not registered, this
 *            registration should be done before the register of
 *            attendance.
 */
public class AttendanceRegister{
    /**
     * @basicFlow(1)
     * @description The secretary selects the patient and date of
     *              attendance in the system. The clicks the button to
     *              register the attendance.
     */
    public void registerAttendance(){}
    /**
     * @basicFlow(2)
     * @description The secretary credentials are verified.
     */
    public boolean validateCredentials(){}
    /**
     * @basicFlow(3)
     * @description The system saves the attendance.
     * @preCondition The user must be logged in the system and must
     *               belong to the secretary user group.
     */
    public void saveAttendance(){}
    /**
     * @alternativeFlow('Patient not registered',1)
     * @description If in step 1 of basic flow, the patient is not
     *              available for the attendance register, the secretary
     *              is directed to the registration of patients.
     */
    public String registerPatient(){}
    /**
     * @alternativeFlow('Invalid Credentials',1)
     * @description If in step 2 of the basic flow, the user credentials
     *              are not valid, an error message is displayed to the
     *              user.
     */
    public void InvalidCredentials(){}
}
```

Figure 6. Source code documentation of the AttendanceRegister class.



Use Case Specification: Register of Attendance

Author: Humberto Ferreira da Luz Junior.
Description: This use case describes the register of attendance of patients at a dental clinic.
Performer: Secretary.

Extension

1. If the treated patient is not registered, this registration should be done before the register of attendance.

Special Requirement

1. The system should display error messages if the registration is not completed successfully.

Basic Flow

1. The secretary selects the patient and date of attendance in the system. The clicks the button to register the attendance.
2. The secretary credentials are verified.
3. The system saves the attendance.[1]

Alternative Flows

1. **Patient not registered**
   1. If in step 1 of basic flow, the patient is not available for the attendance register, the secretary is directed to the registration of patients.
2. **Invalid Credentials**
   1. If in step 2 of the basic flow, the user credentials are not valid, an error message is displayed to the user.

1 **Pre-Condition:** The user must be logged in the system and must belong to the secretary user group.

Figure 7. Attendance Register use case specification.

Among the major contributions expected by the GaiaDoc tool are: (i) less effort in maintaining the documentation of functional requirements focused on the business rules; (ii) a better control on the changes applied to these requirements; (iii) a greater ability to track changes of requirements and link them to other artifacts like class diagram and source code; and (iv) greater consistency and integrity among the artifacts that document the requirements and the implementation.

Currently only the Java classes are supported by GaiaDoc, however, we study the support of other object-oriented languages in a extensible way. The goal is to enable, by means of a simple file configuration with the syntactical features of the language in question, add its support to the tool.

In the future, it'll be possible to use the GaiaDoc API to create plugins for widely used Integrated Development Environments (IDEs) as, for example, Netbeans and/or Eclipse.

The objective of this study was not to propose a tool that replaces the traditional ones directed to API documentation, but to provide an alternative that may better suit the needs of the project in certain cases.

In the case of the Java language, the ideal is to use both tools (Javadoc and GaiaDoc) in the project, taking one over another (or the combination of both), according to the objective of the implemented class.

References

[1] Werner Heijstek and Michel Chaudron. Evaluating RUP software development processes through visualization of effort distribution. In *Proceedings of the 2008 34th Euromicro Conference Software Engineering and Advanced Applications*, SEAA '08, pages 266–273, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Mark Douglas Jacyntho, Daniel Schwabe, and Gustavo Rossi. A software architecture for structuring complex web applications. *J. Web Eng.*, 1(1):37–60, October 2002.

[3] Joseph R. Kiniry and Fintan Fairmichael. Ensuring consistency between designs, documentation, formal specifications, and implementations. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering*, CBSE '09, pages 242–261, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] Douglas Kramer. API documentation from source code comments: a case study of Javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, SIGDOC '99, pages 147–153, New York, NY, USA, 1999. ACM.

[5] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[6] Donald M. Leslie. Using Javadoc and XML to produce API reference documentation. In *Proceedings of the 20th annual international conference on Computer documentation*, SIGDOC '02, pages 104–109, New York, NY, USA, 2002. ACM.

[7] Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[8] Simon Wright. Requirements traceability - what? why? and how? In *In Proceedings of the Colloquium by the Institution of Electrical Engineers Professional Group C1 (Software Engineering)*, pages 1–2, London, UK, 1991.