SOURCE CODE DOCUMENTATION: A TOOL FOCUSED ON BUSINESS REQUIREMENTS

Humberto Ferreira da Luz Junior
Universidade Estadual de Londrina
Londrina, Brazil

Rodolfo Miranda de Barros
Universidade Estadual de Londrina
Londrina, Brazil

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, interpreted by the GaiaDoc tool, that 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.

1. INTRODUCTION

The source code documentation 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 (Kruchten, 2003).

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, which is not usually used to provide services, such as an API, being quite possible to link it 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 source code documentation 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 use case specification come to be neglected (Kiniry, 2009).

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 use case specifications 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 use case specification, 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.

This approach makes it possible to maintain the traceability of requirements, leaving them well bound with the code and make it easier to maintain the integrity of requirements in relation to its implementation.

2. BACKGROUND

This section describes the scenario that has motivated the execution of the study for this paper. The state of the art about source code documentation is presented and the surrounding context such as Javadoc tool, modeling of use cases and MVC architecture is introduced.

2.1 Source Code Documentation

The source code documentation is among the most basic practices taught in programming courses, consisting of a natural language description of functions, methods, classes, variables and constants.

New developers and API users take advantage of this documentation to understand the functioning of the application without creating the need for careful analysis of hundreds or even thousands of lines of source code to understand its logic.

The author of the source code, after a long period of time, can use the documentation to remember his decisions about the way the code was implemented and prevents information from being lost, even after his departure from the organization.

Since legibility of documentation is harmed by being mixed with the source code, usually the programming languages have tools that allow you to export it to other formats like HTML, DOC, PDF, among others.

According to Spinellis (2010), developers are always close to disaster when they are commenting the code. This is due to the habit of describing all the functionality of the code in excessive details, which leads to a documentation, almost entirely, useless and difficult to maintain.

The documentation should not be done this way, because the code should be self-describing, through variables and functions which are well named and easy to understand. The comment blocks should be used to describe the reasons that led the developer to fix the problem in a specific way or describe the rules that apply to that class or function, for example.

2.2 Javadoc

Javadoc is a tool that parses the comment blocks in a set of java classes and it produces their respective HTML pages describing the classes, interfaces, constructors, methods and attributes.

The creation of this tool, according to Kramer (1999), happened because of the need to avoid the fragmentation of the Java API documentation. Despite having a single documentation, for various reasons, documentation existed in secondary sources, sometimes complementing the principal and, sometimes, overlapping it.

The first step of Javadoc specification was deciding to whom the documentation was written. Possible audiences identified were the hundreds of thousands of application developers, hundreds of organizations that used the license to incorporate the Java language in their operation systems and dozens of conformance test engineers.

Further, it was decided unanimously, that the documentation should reside in the source code in order to generate an accurate and updated documentation. Whereas there is a strong link between documentation and code, there is a greater convenience for the API developers, being simpler to write the documentation in the same place the code is written.

The last step consisted in the definition of guidelines to instruct the way developers should proceed in implementing the API and also the way in which the documentation should be written.

2.2.1 Javadoc Operation

In relation to its operation the Javadoc has been implemented to use only part of the java compiler, processing only the declaration of the methods, without its contents, including the hierarchy of classes, their relationships and all the content added within comment blocks, and then generates the HTML.

Relying on the compiler ensures that the HTML output corresponds exactly to the actual implementation, also analyzing the implicit code as, for example, the class default constructor.

Taking into account that only part of the java compiler is used, the Javadoc can be executed even if the source is incomplete or incorrect, so it is possible to generate HTML output even before the clearance and resolution of present problems in the code.

Whereas only the declaration of the methods is compiled and its content is ignored, you can use the Javadoc from the early development of the API, before writing its implementation.

The HTML output of the tool can be customized via doclets, programs written with the Doclet API that specify the content and format of the output to be generated by Javadoc. Using doclets is even possible to generate documentation in other formats like XML and RTF as well as perform other tasks not related to the production of documentation.

2.3 Use Case Modeling

Within the RUP's requirements flow, use case modeling comes after the definition of vision document, which contains the general scope, basic needs and features of the project, besides the analysis of stakeholder expectations. It occurs simultaneously with the user interface design and management of documented requirements throughout the project due to changes and refinements.

The translation of the needs and interests of project's stakeholders contained in the vision document originates the detailing of requirements in form of use cases. These use cases serve as a basis for designing and building the system, identifying test cases and validating the project's outcome, ensuring that all agreed requirements were adequately met (Kruchten, 2003). The use cases specification must occur before the start of the implementation phase of the project and together with other additional specifications, like the other UML diagrams.

The use case model describes the behavior and the relationship between the system and the actors that represent roles played by users through interaction, without going into details about how the system will be implemented (Booch, 2005).

For each identified use case is created its corresponding use case specification in natural language, which contains the pre-conditions, the extensions, the basic description and the step-by-step flow of events run by the actor in that use case. This event stream is divided between the basic, which is what must occur normally, and the alternative one, which begins when unexpected events such as, for example, system failures happens.

Importantly, the modeling of use cases is an extremely popular method for documenting the functional requirements of object-oriented projects in organizations. The use case specification is usually created and maintained as a text document (.doc or .odt, for example), which can cause integrity problems between the source code, the documentation and the specification.

2.4 Model View Controller

Model View Controller is a standard architectural design developed for the Smalltalk language which proposes that business rules should be decoupled from the vision and the application domain and is therefore a three-tier architecture. Its goal is to simplify the implementation and to facilitate code reuse.

The model represents the data and behavior that will be displayed to the user. A view is created for each user interface that can be presented to the user and its content depends on the domain. The controller receives and processes a particular user input to a specific function and is responsible for maintaining the domain always synchronized with the view, coordinating the data flow between them (Jacyntho, 2002).

In web applications, which usually adopt the MVC architecture, the user triggers a request through the Web browser. On the server side the request is handled by the controller, which updates the model and then determines which view should be presented to the user. Finally, the server sends a response to the client and the result is displayed to the user.

This architecture was chosen due to the decoupling of the business rule, which makes the MVC architecture an ideal example of how GaiaDoc can be used efficiently. Use cases can be represented in the Controller layer and its documentation represented later in the form of the use case specification.

3. METHODOLOGY

The mapping between the code documentation of classes and the use case specification is done using annotations within the block comments 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 annotations @name and @description are mandatory in the class scope, while @writer and @performer must be cited at least once. @extension, @specialRequirement, @preCondition and @postCondition are optional and may occur several times in the comment block.

In the attribute scope only @description is allowed. The attribute name is obtained from the variable name that belongs to the comment block attribute. The attributes are used to generate the Glossary which is in a dictionary of terms relating to the use case in the form of a table.

The method's comment block has the @description and @performer annotations. It is also required that the @basicFlow or @alternativeFlow annotation be present in the block.

After reading the annotations in the file that contains the class, GaiaDoc converts these data into the class structure shown in figure 1.

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.

3.1 GaiaDoc

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 use case specifications generated from the class annotations, will be saved in PDF (Portable Document Format).

The main difference between Javadoc and GaiaDoc is that the first one focus on documentation of APIs, therefore the documentation readers are mainly developers, while the second one focus on business requirements mapped on classes and written in the source code through a different set of annotations.

There is also optional parameters to remove specific parts of the use case specification 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.

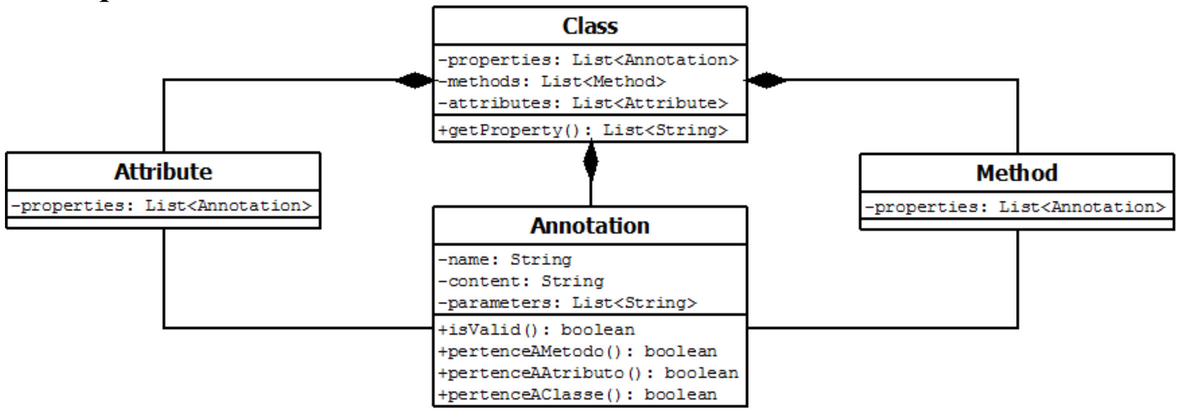
By the time the command is executed, the tool traverses all the files in the given directory, including those in its subdirectories. Each found class is parsed to determine whether it contains all the required annotations and, if so, the documentation is obtained from the file and it is used to generate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

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 2.

If the parser does not follow the provided flow by the states diagram, the documentation is considered invalid and the use case specification 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.

3.2 Requirements Flow



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

Despite the use case modeling does not limit the choice of the Software Development Process (PDS), 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 (Heijstek, 2008).

Figure 1: Class diagram of the structure that represents the data read from the java file by the GaiaDoc tool.

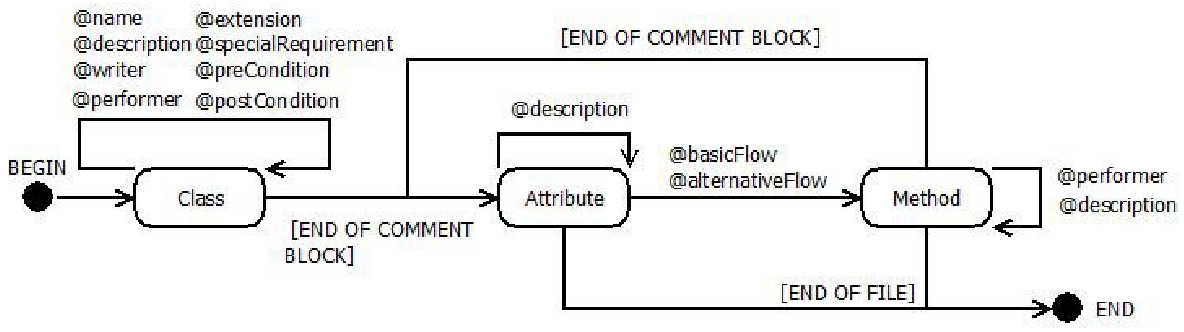


Figure 2: State diagram of the GaiaDoc parser.

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 use case specification 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's scope, subsequently much of the already documented requirements are duplicated in the source code documentation, 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 recommended by RUP after the definition of the use case diagram. Following the identification of use cases, a preliminary class 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 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 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 use case specifications located in the source code and even the code itself.

The fixes of bugs identified in the documentation can be registered in a bug tracking system, 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.

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 source code documentation in real projects, it is common that the documentation requirements is 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.

In relation to the 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 use case specifications, 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 (2002) 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.

4. CASE STUDY

To demonstrate, in fact, the operation of the GaiaDoc tool, it was prepared the case study of a software system of certain dental clinic, assuming that the vision document of the project has already been created and requirements analysis has already begun.

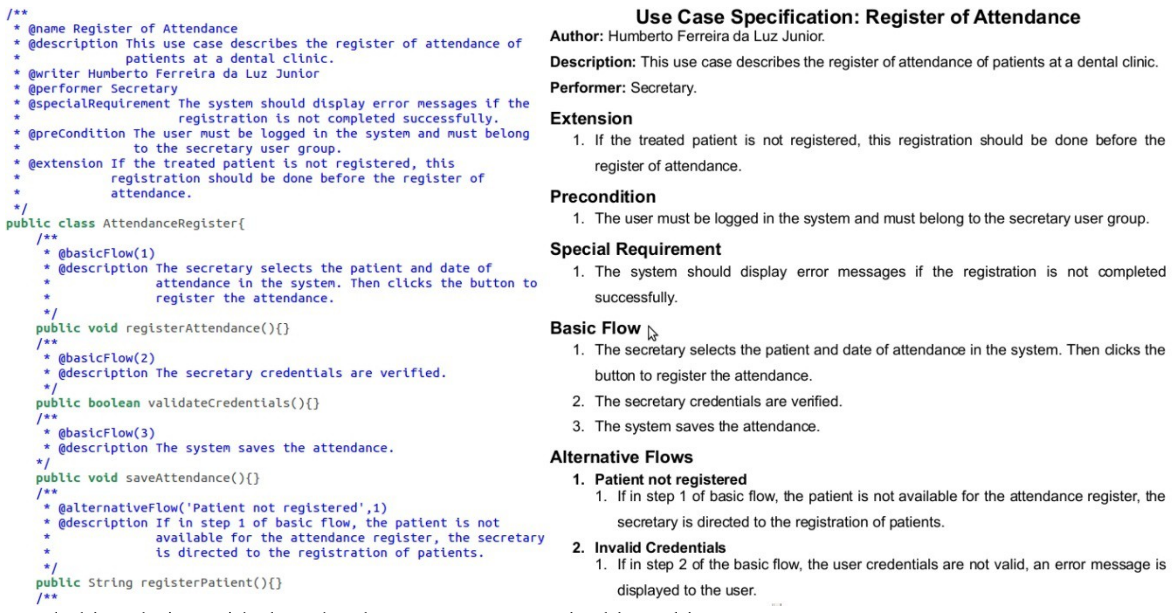
The use case has only one actor, the clinic secretary, who may register both patients and appointments. When a service is registered in the system, it is required that the treated patient is already in the database, otherwise, he should be inserted into the system before the registration of the appointment.

In any persistence operation the user validation must occur to determine if he belongs to the secretary user group and, therefore, have permission to register patients and appointments, or if the operation is invalid.

For this case study, the Register Appointment use case was arbitrarily chosen, which has extension relationship with the Register Patient use case.

The next step consists in the preliminary scope of the class diagram.

The left part of figure 3 shows the RegisterAppointment class, in the Java programming language, which contains the source code documentation in the GaiaDoc format and represents the controller in MVC layer. The view layer can be created using libraries of components like the framework JSF (Java Server Faces) or only with HTML. The model layer can be mounted with POJO (Plain Old Java Objects) classes or with entities of the framework JPA (Java Persistence API). This demonstrates that the controller is a loosely



```
/**
 * @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.
 * @preCondition The user must be logged in the system and must belong
 *               to the secretary user group.
 * @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. Then 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.
    */
    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(){}
    /**
```

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.

Precondition

1. The user must be logged in the system and must belong to the secretary user group.

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. Then clicks the button to register the attendance.
2. The secretary credentials are verified.
3. The system saves the attendance.

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.

coupled in relation with the other layers components in this architecture.

After the GaiaDoc execution in the AppointmentRegistration class, the output use case specification (shown in the right of figure 3) is generated.

Figure 3: The source code documentation in the left and the resulting use case specification in the right.

Other examples of controller classes and their respective resulting use case specifications can be found in the GaiaDoc SVN repository located in Google Code, whose address is referenced in next section of this article.

5. FUTURE WORK

The work is in progress to implement the GaiaDoc tool, which is already functional, but it has some limitations on the class documentation format that must be addressed. Its source code is available in the project's SVN repository on Google Code, located under the GNU GPL v3 (http://code.google.com/p/gaia-doc/).

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.

Upon completion, the API will be applied to a real project and its results will be reported in another article.

6. CONCLUSION

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

Among the major contributions provided 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.

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 combination of both), according to the objective of the implemented class. It is the responsibility of the requirements analyst, together with the development team, to determine whether the class should be documented using the Javadoc, the GaiaDoc, or in some cases, the use of both.

In relation to the SDP, the GaiaDoc was specified to be used exactly in conjunction with RUP and UML, including some prerequisites, such as use case and class diagrams, as well as the use case specification as output. The control of changes in the source code is more strict, because the use case specification passes to be integrated with source code.

REFERENCES

Booch, G. et al, 2005. *Unified Modeling Language User Guide*. Addison-Wesley Professional.

Heijstek, W. and Chaudron, M., 2008, Evaluating RUP software development processes through visualization of effort distribution. 34th Euromicro Conference Software Engineering and Advanced Applications. Washington, USA, pages 266-273.

Jacyntho, M.D. et al, 2002. A software architecture for structuring complex web applications. Journal of Web Engineering, 1(1), pages 37-60.

Kiniry, J.R., and Fairmichael, F.,2009, Ensuring consistency between designs, documentation, formal specifications, and implementations. 12th International Symposium on Component-Based Software Engineering. Berlin, Germany, pages 242-261.

Kramer, D., 1999, API documentation from source code comments: a case study of Javadoc. 17th annual international conference on Computer documentation. New York, USA, pages 147-153.

Kruchten, P., 2003. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing, Boston, USA.

Leslie, D.M., 2002, Using Javadoc and XML to produce API reference documentation. 20th annual international conference on Computer documentation. New York, USA, pages 104-109.

Spinellis, D., 2010. Code Documentation. *IEEE Software*, 27(4), pages 18-19.