Applying Pattern Structures to Document and Reuse Components in Component-Based Software Engineering Environments

Marco Paludo[1,2], Sheila Reinehr[1], Andreia Malucelli[1], Lucas Bruzon[2], Pamela Pinho[2]
[1]*PUCPR Pontifical Catholic University of Parana;* [2]*FESPPR Higher Education College of Parana*
[1]*R. Imaculada Conceição, 1155, Curitba-PR,Brazil;* [2]*R. Dr. Faivre, 141, Curitba-PR,Brazil*
{marco.paludo, sheila.reinehr, andreia.malucelli}@pucpr.br; lucasbarke@hotmail.com, pamela.m.pinho@hsbc.com.br

Abstract

One of the challenges for software development organizations that try to apply software reuse programs is to make the specification, persistence and easy access to the component repository feasible, mainly considering the elaboration phase, but also addressing the construction phase of the software product. This paper uses some component documentation initiatives based on analysis and design patterns, and proposes a component specification structure, presenting a tool to support this process. The general purpose of patterns is to document, retrieve and, mainly, capture composition and functionalities of the components in order to achieve software reuse. The objective of integrating patterns and components approaches is to leverage the software reuse process by creating a documentation structure and applying a component repository able of supporting the software developers.

Keywords: Software Reuse; Components, Analysis and Design Patterns; Specification

1. Introduction

Software reuse is the process of creating software systems from existing software, instead of constructing software from scratch [1], and it has been extensively addressed by researchers and practitioners, however a more systematic approach applying reuse in all life-cycle phases and iterations has to be part of the process [2].

Reuse initiatives have the main objective of using intermediate or final products conceived in other projects that have already been successfully tested (certified) and implemented before, aiming to reduce time-to-market and getting better general quality, testability and debugging procedures. Even considering that reuse of classes and components libraries are useful, it does not improve fundamentally the software development process, so concepts and tools of a higher level are necessary to leverage the development process [3].

Even when there is a component repository available to developers, it is very important to consider that the components have to be easily discoverable, otherwise it could be easier to develop a new component rather than spending much effort trying to reuse it. Up to now, the activity of "finding reusable components remains a significant barrier for exploiting systematic software reuse" [4].

Another aspect to consider when implementing reuse initiatives is software architecture, which is also considered as the basis for achieving reuse, and when combined with component-based software development, the result is the notion of software product lines [5]. Clements et al., in [6], consider that architecture is a reusable model that can become the basis for an entire family of systems, built using common assets. Also states that "Software architecture is an asset an organization creates at considerable expense. This expense can and should be reused".

2.1. A. Motivation and Objective

This project started by investigating previous patterns and components literature, identifying the patterns methods adherent to Component-Based Software Engineering (CBSE), with special attention to specification structures. One of the main objectives was to help software engineers when creating or searching components to be incorporated to their software products development projects, aiming component reuse.

The subsequent steps resulted in the proposition of a specification structure (template) for documenting components, presented in Section IV, conceived to become a part of the software development process, in order to avoid rework in the elaboration and construction phases of the software development life-cycle.

Another subsidy for the project was the ISO/IEC 25.010 (formerly ISO/IEC 9.126) series of International

Standards of software product quality, that has provided the quality characteristics and subcharacteristics concepts, which have been incorporated to the specification template, in order to call the attention earlier in the development process for quality attributes and metrics of components [7].

Hence, the objective of this paper is to initially present some patterns and components methods considered by this project, and especially to propose a component specification template that incorporates some characteristics of patterns approaches, component documentations and software product quality models. The Section II of this work presents a brief introduction to patterns approaches and Section III details some structures of patterns that are used along with component methods. Section IV discusses the rational used to apply patterns documentation to specify components, as well as proposes the specification structure (template), supported by a tool adherent to this work. This section also presents some characteristics of the tool and a partial class diagram, showing the classes and relationships, as well as some of the core methods and attributes. Section V concludes the paper and presents some future works.

2. Patterns

A well-known definition of patterns is provided by Erich Gamma et al. [8], stating that design patterns capture solutions that were developed and evolved, with a succinct and easily applicable way. Each design pattern systematically nominates, explains and evaluates an important subsystem, and occurs many times in object-oriented projects. Patterns, along with frameworks, have played an important role reusing software artifacts and code, but the efforts are gradually migrating toward the intermediate products used in earlier phases, i.e. analysis and even requirements assets. Some example of these target assets are use cases, requirements, sequence diagrams etc.

An earlier study [3] has analyzed some patterns initiatives and presented them categorized by the emphasis on analysis or emphasis on design (implementation), as well as categorized as process-driven or example-driven. Among the methods considered, the ones that are more adherent to this work are those with special emphasis on analysis, considering that components, depending on their granularity, tend to represent higher structures than objects and classes.

So, it is possible to infer that components have more chance to leverage reuse in a higher level of abstraction. The process adopted by each method is not relevant to this work, considering that the main contributions obtained from the methods are the documentation structures of patterns, to be used to document components, not the phases, tasks, and roles from the processes.

3. Patterns approaches and CBSE

Erich Gamma et al. [8] state that ideally, new components should not be created in order to achieve the specified level of reuse, but all the required functionalities should be obtained just putting together existent components using composition. On the other hand, at that time, affirmed that it was not common because the number of components available was almost never sufficient to fulfill the requirements in the real world. This situation has changed and, sometimes, the number of components available in an organizational repository is so big that the searching task can hinder the reuse of components.

Another classical reference presented by Graig Larman, in [9], proposes to apply reuse by integrating classes and objects with patterns and frameworks. When introducing the GRASP Patterns (General Responsibility Assignment Software Patterns), the method defines that the patterns describe the fundamental principles for assigning responsibilities to the objects. This method makes use of component concepts just when addressing the Implementation Diagrams, referring to the Component Diagram and Distribution Diagram.

One previous work [3] have analyzed how adherent patterns methods are to components, especially addressing the way components are defined in the context of Component-Based Software Engineering (CBSE) [10]. The result is that patterns mainly consider objects (from the object paradigm), not components. Among other methods, it is possible to identify that three have a clear focus on objects, rather than on components, each of them with a particular emphasis, notation and process [3]. Nevertheless, they also could contribute to the component specification template proposed by this work. The following subsection identifies some component methods and the relationship with patterns, with special attention to the structure of documentation and specification.

3.1. Component approaches

The efforts to conceive reusable and scalable structures have changed from objects to components, as addressed by CBSE. Some methods presented below demonstrate such evolution, where components are treated as a native element for composing the patterns and frameworks. This initiative is, sometimes, referenced as componentware, but the objective is the same: make components available and follow a process to obtain reuse of higher level assets, rather than just addressing the object level.

The Catalysis Method [11] [12], considering CBSE and patterns/frameworks approaches, strongly emphasizes the need for an architecture that supports components and states that development projects that try to apply patterns and components without an established architecture tend to fail. Also considers natively the relation of patterns and components, all making use of the architecture.

Another method, the UML Components [13], states that components extend the object-orientation principles of data and function union, encapsulation, and identity, by specifying an object with its explicit interfaces representation. Some characteristics, such as inter-component dependency, turns to be restrict to the individual interfaces and that future implementations in components will not cause great impacts to the intermediate or final software products. The specification of interfaces is considered an essential task when dealing with components and reuse.

The last method, presented by Alan Brow, Large-Scale Component-based Development [2], considers components and patterns integrated, not with the same depth of [11], however the use of patterns is simple once the patterns concepts can be implemented in the original structure proposed by the method. It states that the software construction is largely done by component selection, evaluation, and assembly process. When analyzing the persistence requirements of components in this approach of CBSE, it is possible to make a comparison with patterns catalogs. The patterns catalogs usually have a well-defined structure, what helps to retrieve them with efficiency and making the reuse process feasible. Following the patterns specification structure, with some complementation, it is possible to save components in a way to have the component retrieving easier and standardized, also aiming the reuse.

Assuming that object-oriented approaches have evolved toward CBSE, it is considered that this change also may, and should, be applied to the patterns methods, in order to get reuse of a larger set of assets in all stages of the software development life-cycle.

4. Using patterns structures to specify components

Before discussing the specification structure, it is important to present some elements that are addressed by this section and also considered by patterns approaches. The essence of patterns definitions can be stated as: a recurring solution to a problem in a context [14].

It is important to present the concepts of solution, problem and context. The Context information represents the environment, surroundings, or interrelated conditions within which something exists. The Problem information is an unsettled question, something that needs to be investigated, analyzed and solved. Typically, the problem is constrained by the context in which it occurs. The Solution information refers to the answer to the problem in a context that helps resolve the issues.

So, a solution to a problem in a context, by itself, is not considered a pattern or a component due to a common aspect, the recurrence. To be worthy to be incorporate in a framework or in a reuse repository, a component has to be useful in a context and likely to be applied repeatedly in other contexts. When the interfaces provided by a component are not well-known and well-defined, the overheads involved in searching the particular solutions into a component repository may not be justified [15]

It is known that one of the pitfalls when considering reuse within a software development process is the effort of the software engineer to find the right pattern or component, and also the effectiveness of the information available for this activity. Sometimes, even due to cultural issues, the developer prefers to conceive a solution or component rather than reusing one.

These are some of the issues considered by this work when conceiving the specification structure, trying to go beyond the problem-solution-context aspects of CBSE.

A more complete component specification structure aims to improve the cataloging process and persistence of components created by a developer or team, as well as to assist others who will make use of that previous solution addressed by a component.

Heineman [10] states that this structure may express the specification, implementation, and the expected execution and deployment, representing types of components. It is important to consider that each of these components shall have a particular level of abstraction when following a specification structure. Hence, besides the types of components, one should also consider the granularity and scope of the component.

A mistake that is commonly made in many software development organizations is to treat the architecture of a set of related systems and the architecture of all systems across the enterprise at the same level of abstraction [16]. The architecture is not the focus of this work but it is essential to succeed when trying to implement reuse initiatives.

In general, higher level components are conceived after one specific solution has been implemented. Sometimes even during maintenance some of them are also considered eligible to be incorporated in the repository. It demands, then, that the detailed and standardized documentation has to be created or complemented during the elaboration phase, and before the transition.

4.1. Software life cycle processes – Maintenance

Some activities of the ISO/IEC Software Life-cycle Processes – Maintenance [17] International Standard are presented, analyzing the influences on the proposed component specification template presented in Table I. Three activities have been used, and are following presented, as well as the corresponding items (with item number) of the specification template proposed by this work, that are directly affected by the activities:

a) Understand the problem domain (the type of application), using the existent documentation (if available), discussing the software product with developers (if available), and operating the software product. This inspection activity should raise information that could help filling in the Context (item 4 of the template), Introduction (item 4), Problem (item 5), and Applicability (Purpose) (item 6) fields in the component specification proposed by this work.

b) Learn the structure and organization of the software product, including control and data flow. Inventory the software product, placing the product under configuration management, and analyze the structure of the software product. These activities address the Description (item 7 of the template), Implementation (item 7.3), Solution (item 7.2), Variants (item 12), and Relationship (item 13) fields of the specification template.

c) Determine what the software product is doing. Review specifications (if available) and overall structure, read and provide comments to the code. This should help filling up the Purpose (item 4 of the template), Example (item 11), Interfaces (item 8), and Forces (item 9).

These activities were used as subsidy to conceive the specification template and are strongly suggested to be executed when creating, updating or retrieving components to/from the repository.

4.2. Quality model and component specification

When considering software requirements specifications (SRS), all methods address functional and non-functional requirements, and some make use of a quality model as a check-list, helping the software engineer to find essential non-functional requirements, like usability, efficiency, time behavior, among others.

The International Standard ISO/IEC 25.010 Quality Model present characteristics (functionality, reliability, usability, efficiency, maintainability, and portability), with associated subcharacteristics.

The Quality model for internal and external quality [7], categorizes software quality attributes into six characteristics, and each of them are further subdivided into subcharacteristics, that strongly influences the quality of the software product. This categorization, originally used to determine the software product quality, was considered in this work to compose a section of the proposed component specification template (item 10 of the specification template).

Some examples of quality characteristics that a component usually have and can be easily specified are Functionality, Usability, Efficiency, and Maintainability. As subcharacteristics represent the breakdown of a characteristic, one possible example is the Security subcharacteristic, from the group of Functionality characteristic, with the associated metric of Access Auditability, where the number of access types that the component is logging correctly can be assessed to determine its compliance to the specified in the metric. A large number of metrics can be specified and the context of use will determine the quantity and granularity of the metrics and quality characteristics.

4.3. Component specification template

Some patterns structures that are able to document components are present in the methods of Gamma et al. [8] and Buschmann et al. [18], and were considered as a subsidy when conceiving the proposed specification template. Other two methods also considered, and even more adherent to the objective of this work, were proposed by Stelting et al. [19] and Szypersky [15]. These methods form the basis for conceiving the following component specification template proposition, presented in Table I.

The use of most parts of the component specification template has been described in subsection A of this section (IV), when considering the Software Engineering International Standards [17], [7]. In addition, the items 1, 2 and 3 of the template are well-known and largely used by many patterns documentation structures.

Considering that this proposed specification structure have many similarities to the originally presented in [8], it is possible to conclude that the component specification based on the original patterns structure is adequate. Most of the original items of patterns and components were partially (adapted) or totally used.

Another method addressed by this project is presented by Alur et al. [14], that consider patterns, components and particularly the Java language, and has points in common with [17], emphasizing the Java language and Sun Microsystems, but using a particular structure to document the component patterns J2EE.

When analyzing the variety of patterns and components structures, it is possible to observe the similarities among them, even considering that there are significant differences concerning the abstraction level of the patterns. And when comparing these approaches with the one presented by Cheesman et al. [13], it becomes evident the need to explicitly determine the interfaces of the components.

TABLE I. COMPONENT SPECIFICATION STRUCTURE

Component Specification
1. Component Name: A descriptive name of the component.
2. As known as: Alternate name or names.
3. Properties: Classification considering the type, subtype and level.
3.1 Type: Creational, Behavioral, Structural and System.
3.2 Subtype: Specification, Implementation, Execution and Deployment.
3.3 Level: Unit, Component or Architectural.
4. Purpose and Context: Explanation of the scope and the environment under which the component exists.
5. Problem: A brief description of the problem to be treated by the component, presenting the design issues faced by the developer. It is recommended including an example to illustrate.
6. Applicability: The pros and cons when using this component. The benefits and drawbacks, representing the consequences and difficulties when using the component. Emphasize the result of the component use.
7. Description: detailed discussion of the component, what it does and how is the behavior.
7.1 Structure Solution – Class diagram with the basic solution Structure and sequence diagram representing the dynamic model.
7.2 Solution – strategy: Presents the ways that a component can be implemented.
7.3 Implementation: Describe what has to be done to implement the component.
8. Interfaces: The way the component makes its service available. Commonly multiple interfaces are provided corresponding to different access points.
9. Forces: Lists the rational and motivations that affect the problem and the solution. The list of forces highlights the reasons why one might choose to use the component and provide a justification for using it.
10. Quality Characteristics and Subcharacteristics addressed: present the quality model attributes that are addressed by the component. Whenever possible, provide validated or widely accepted software product quality metrics.
11. Example code: implementation examples or source code of the component, when applicable.
12. Component variants: possible alternate implementations.
13. Related Components: a set of other components associated or related, internal or externally, from the perspective of the repository.

Without even taking into account which patterns structure and method is used, it is important to consider that component specification requirements should be fully satisfied in order to make the component reuse process successful.

4.4. Component repository tool

Based on the component specification template presented in Table I, a tool was modeled and built implementing the component cataloging and searching/retrieving features.

The queries to the repository are made considering the textual fields as Purpose, Problem, Applicability etc., and can also consider the source code of the software component, whenever available. In order to make the component search and retrieve procedures more precise, the result can be sorted by the number of times the component was retrieved, indicating that the first components in the list were more times used and probably will be more times reused.

Some configuration management principles are implemented by controlling the version of the component, as an internal functionality. Also, there are two distinguished roles for the users. The first role is played generally by the software engineer or system analyst, who proposes and retrieves (uses) components from the repository. The software development team usually has just this privilege to access the tool.

All the updates proposed to a component has to be approved by a second role, the manager user, who is responsible for verifying if the component is pertinent, follows the organization's standards and if the documentation form is complete and correct. This role is usually found in the architecture team of organizations.

There is, also, a message control feature that puts together the consumer and the producer of the component, in order to report bugs, improvements, and corrections in the documentation or even in the source code, whenever available and pertinent.

All the artifacts, models and source code can be downloaded from the project's web page[1]. Among the artifact develop in the inception and elaboration phases, are the Sequence Diagrams, including the ones treating CRUD (Create, Retrieve, Update and Delete) functionalities, Print Component, Generate Report, and Validate Component. The Use Cases diagrams developed are: Issue Managerial Report, Issue Operational Report, Create Components Statistics, Create Form, Manage Components (CRUD), Print Components, and Validate Components. Other diagrams as the State Machine and Activities are also part of the project documentation and available at the projects web page.

One of the most important functionality is the Insert Component, and it has a main focus on textual fields, as it includes all the sections of the component specification template, previously presented in Table I. The searching functionality will be as effective as the quality of descriptions made for each component, so the process of creating and validating the components with as much relevant information as possible is so important.

[1] All artifacts of the software product can be found at www.ppgia.pucpr.br/pesquisa/engsoft/comp_repository

5. Conclusions

This paper has presented the analysis of patterns and components methods, in order to conceive a component specification structure, in the form of a template that can be applied to promote reuse of components. Considering the similarities between both documentation structures, it is possible to conclude that the component specification based on the original patterns structure is adequate.

Therefore, it is proposed that CBSE should make use of the patterns methods to leverage reuse in all stages of the life-cycle, considering the granularity of the component. The template proposed by this work is one step toward improving the actual patterns documentation, to be effective when addressing components.

The scope of this project also included modeling and developing a tool that persists software components, applying the proposed documentation structure, able of creating, retrieving and managing the content of the component repository, where regular users can conceive and retrieve components, and manager users validate and approve all the component updates in the catalog.

The next step planned is to deploy the repository tool as-is in a Software Product Line environment aiming to evaluate the quality in use results and the integration with the software development process institutionalized.

Alnusair et al., in [4], propose a comparison of some query approaches, considering semantic, key-words and signature based queries. One future work could be implementing semantic-based representation and annotation of library components, in order to get the searching and retrieving effectiveness improved.

Other extension to this work could consider exceptions to be part of the component specification structure, defining the expected behavior when some abnormal, but anticipated, condition occurs to a component [20].

Some component properties and characteristics (identification, use, maturity, documentation, among others) have generated a Component Reference Model in [21], and also could be used as a basis for mapping the specification structure of this work into a maturity software component classification and reference model.

10. References

[1] C. Krueger, Software reuse. ACM Computing Surveys, 24(2), 1992, pp.131-183.

[2] A. Brown, Large-scale component-based development, Upper Saddle River: Prentice-Hall, 2000.

[3] M. Paludo, R. Burnett, and S. Reinehr, Applying pattern techniques to leverage component-based development, In: Proceedings of the IASTED International Conference on Advances in Computer Science and Technology. Puerto Vallarta: 2006.

[4] A. Alnusair, and T. Zhao, Component search and reuse: a ontology-based approach, In Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI 2010), pp. 258-261, 2010.

[5] J. Bosch, Design and use of software architectures, England: Addison-Wesley, 2000.

[6] P. Clements, R. Kazman, and M. Klein, Evaluating software architectures: methods and case studies. Boston: Addison-Wesley. 2002.

[7] ISO/IEC FDIS (Final Draft International Standard) 25.010:Software engineering - software product quality requirements and evaluation (SQuaRE), 2010.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns: elements of reusable object-oriented software. Reading: Addison-Wesley, 1994.

[9] C. Larman, Applying UML and patterns – an introduction to object-oriented analysis and design and the Unified Process (2nd Edition). Upper Saddle River: Prentice Hall PTR, 2001.

[10] G. Heineman, and W. Councill, Component-based software engineering: Putting the pieces together, Boston: Addison-Wesley, 2001

[11] D. D'souza, and A. Wills, Objects and frameworks with UML: the Catalysis approach,Addison Wesley, 1998.

[12] D. D'souza, Catalysis – systematic components and frameworks with UML. http://catalysis.org/publications/papers/2000-components-frameworks.pdf, 2000

[13] J. Cheesman, and J. Daniels, UML components: a simple process for specifying component-based software. Boston: Addison-Wesley, 2001.

[14] D. Alur, J. Crupi, and D. Malks, Core J2EE™ patterns: best practices and design strategies. Upper Saddle River; Prentice Hall PTR, 2001.

[15] C. Szyperski, Component software: beyond object-oriented programming; Harlow: Addison-Wesley, 1998.

[16] C. Allen. Realizing e-business with components, Harlow; Boston: Addison-Wesley, 2001.

[17] ISO/IEC JTC1/SC7 FDIS 14764 – Ssoftware life cycle processes – maintenance, 2005

[18] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-oriented software architecture: a system of patterns, John Wiley, 1996.

[19] S. Stelting, and O. Maassen, Applied java patterns, Upper Saddle River; Prentice Hall PTR, 2002.

[20] F. Castor Filho, P. Guerra, V. Pagano, and C. Rubira, A systematic approach for structuring exception handling in robust component-based software, Journal of the Brazilian Computer Society. Number 3, Vol. 10, April 2005. SBC, 2005.

[21] G. Redolfi, L. Spagnoli, P. Hemesath, R. Bastos, R. Ribeiro, M. Cristal, and A. Espindola, A reference model of reusabel components description, In Proceedings of the 38th Hawaii International Conference on System Sciences, 2005.