Modularization of Logical Software Architectures for Implementation with Multiple Teams

Nuno Ferreira
I2S Insurance Software Solutions S.A.
Porto, Portugal
nuno.ferreira@i2s.pt

Nuno Santos
CCG - Centro de Computação Gráfica
Guimarães, Portugal
nuno.santos@ccg.pt

Ricardo J. Machado
Centro ALGORITMI, Escola de Engenharia, Universidade do Minho
Guimarães, Portugal
rmac@dsi.uminho.pt

Abstract— **In the end of a requirements elicitation phase, it is expectable that all information can be properly perceived by the implementation teams. In our work, we faced the problem of dealing with a large set of requirements that need to be implemented by different teams, to ensure on-time delivery. The teams are physically separated and with different working cultures and skills. The system requirements were (semi-)automatically derived from user requirements, are framed within a logical architecture (of the intended system) and the desired high-level execution scenarios are defined and included in the requirements description. Due to the large size of the architecture and to the nature of the working environment, the architecture must be divided before being delivered. In this paper, we present our approach for dividing the logical architecture of the intended system into modules to be delivered for implementation. The division in constructed upon requirement views that provide information about the modules, their interfaces, the intended execution scenarios, and the interfaces with other modules or systems. The approach is evaluated using the ISOFIN project.**

Keywords— ***Logical software architectures; architectural design, architecture modularization***

I. INTRODUCTION

For many decades, project managers and "software architects" have complained that they have no accurate way to tell a developer what a product or component must do. Moreover, the code tells what the code does, not what it was intended to do [1]. Although many software documentation methods have been proposed in the research literature [1-3], none has proven entirely satisfactory for developers. Some methods have worked well in projects but had serious limitations; others were theoretically adequate but found to be difficult for developers to use [1]. Documentation that acts as reference for developers should provide an easily accessed source of trustworthy and detailed information about the program and its behavior. Using a reference document, one should quickly be able to answer questions such as [1]: (1) "What will this component do if it receives ..…?"; (2) "What are the circumstances that lead to the output of ..…?"; and (3) "How can I get this component to ..…?". Reference documents include the description of the process, requirements to the execution and conclusion, actions and behaviors, *i.e.*, the requirements to the development of the intended artifact that make up a software solution.

Part of the representation for the requirements of the intended software solution are represented in logical architectures [4]. A logical architecture is a model that provides an organized vision of how functionalities behave regardless of the technologies that require implementation.

In [4], we presented the V+V process, a requirements elicitation approach, composed by two V-Models [5], for creating context for business software implementation teams in contexts where requirements cannot be properly elicited, based on successive models construction and recursive derivation of logical architectures. The requirements are expressed through logical architectural models and stereotyped sequence diagrams in both a process- and a product-level perspective. The result of the requirements elicitation process using logical models is however inadequate and, this single perspective, provides incomplete information for implementation teams. A complete system architecture, in software engineering, cannot be represented using a single perspective [6]. Using multiple viewpoints, like logical diagrams, sequence diagrams or other artifacts, contributes to a better representation of the system and, as a consequence, to a better understanding of the system. Software requirements artifacts should allow understanding of the "who", "what" and the "why") [7].

At the end of the requirements elicitation phase (the execution of V-Model presented in Fig. 1), after deriving a software logical architecture composed by more than one hundred elements, each one representing a given software functionality, the authors faced the problem that development teams were unable to understand such information. Due to the large size of the architecture and to the nature of the working environment we were facing, it is necessary to distribute the work (*i.e.*, the CPI presented in Fig. 1) by multiple teams, in order to prevent a delay on the project execution that could endanger the product time to market. Our approach is executed in scenarios (working environments) where the business domain is stable, the projects macro-planning is performed and the teams are allocated and formalized. Each development team receives a subset of the architecture, *i.e.*, a set of modules. It is easier to present to development teams a

978-1-4799-4264-0/14 $31.00 © 2014 IEEE
DOI 10.1109/ICCSA.2014.14

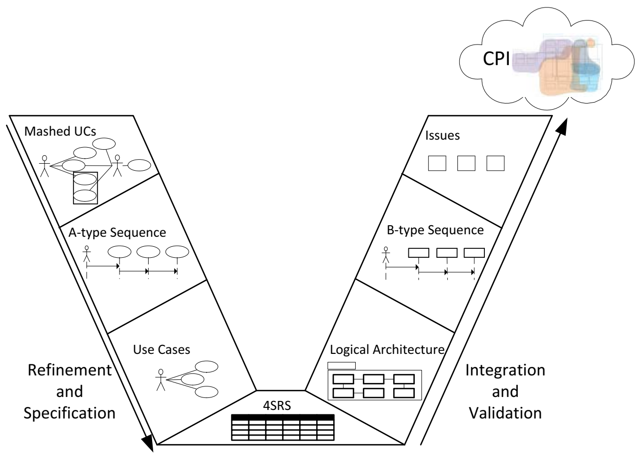


Fig. 1. The product-level V-Model

set of architectural modules than the whole architecture and its inherent complex specification. After presenting the modules, implementation teams were able to understand the functionalities they will develop and even to assign to each team the modules' development.

We present in this paper a requirements elicitation approach that allows creating information to provide context for software product implementation using logical architectures. This information is created in a multi-view artifact, called *Combined Multiple View* (CMV), later detailed in section IV. This artifact intends to provide information regarding software functionalities, so teams are perfectly aware of the product they are going to develop and implement. It is not the purpose of CMV artifact to provide technical information regarding the module implementation (*e.g.*, service protocols, message data, etc.), rather it provides functional and behavioral information of the module in design that will later support the technical specifications of the modules.

We propose to modularize large logical architectures by specifying scenarios in stereotyped sequence diagrams to partition development information regarding functional requirements assigned to multiple development teams. We also discuss the requirements prioritization and the delivery to development teams of component-based combined multiple views regarding the identified modules.

This paper is structured as follows: in section 2 it is presented a literature review on the main topics issued in this work. Additionally, we revisit the previous work regarding the V+V process. In section 3, we present the architecture modularization and related decisions. Section 4 presents the CMV for the architectural modules; section 5 presents a demonstration case of a modularized architecture and its CMV; and section 6 presents the conclusions.

II. Architecture Modularization within a Context for Product Implementation

Our approach starts by eliciting requirements at a process-level perspective, and then, by successive models derivation, creates the context for transforming the requirements expressed in information system logical architecture into product-level context for requirements specification[8]. From our approach, two main artifacts are considered as the result of the requirements elicitation and delivered to implementation teams: the software logical architecture and *B-type* stereotyped sequence diagrams [8]. The approach is used to elicit requirements for creating context for business software implementation teams in contexts where requirements cannot be properly elicited, deriving the product-level requirements based on the process-level requirements in an aligned way, through well-defined steps and rules [9]. This architectural logical representation is however inadequate and provides incomplete information for implementation teams, since the information may be considered to still have a high-level of abstraction, without well-defined boundaries between software products (interfaces) and no usage scenarios. Instead of dealing with logical AEs, the software architecture should be represented by component-based diagrams, like in [10, 11]. Implementation teams are more aware of component diagrams, because the type of information they typically deliver (*i.e.*, behavior in terms of provided and required interfaces [12]) are more useful for the implementation itself.

Development teams use "component" (1) and "module" (2) as near synonyms with a subtle difference. (1) A component is a collection of programs that are distributed as a unit and used in larger systems without modification [1], represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment [12], and represents pieces that are independently purchasable and upgradeable [13]. Modern software systems are based on integrated components [14]. A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [15]. (2) A module is a collection of programs to be implemented as a single work assignment by a programmer or group of programmers [1].

In other words, a module is a unit for software development while a component is a unit for software distribution and deployment purposes. A component may comprise several independently developed modules and to build a module (or parts of a module) is used one of more components. A component serves as a type whose conformance is defined by the provided and required interfaces (encompassing both their static as well as dynamic semantics) [12]. Since Card [16] defines software modules as the basic unit of software development, maintenance, and management, large sized software architectures should be divided into modules.

The amount of architectural elements and functionalities that compose a software logical architecture may result in the need for modularization and implementation by multiple teams, instead of delivering it to a single team and endanger the project's execution on time. The ultimate purpose of software modularization is to increase software productivity and quality, and to ease software maintenance [17], and there are some approaches for effort estimation [18-20]. These approaches are based on use cases and decisions are based in use case points, which makes them inappropriate for architecture modularization since elements from the software logical architecture are represented instead of use cases.

A basic activity of the software design process is the partitioning of the software specification into a number of program modules that together satisfy the original problem statement [16]. Aspect-oriented programming [21] and feature-oriented programming [22] are the most common modularization techniques. By assessing software evolution using dynamic metrics, it is possible to link software modularity with quality metrics directly [17]. All metrics and factors can be quantified, and their relations can be statistically calculated [17].

A very common used artifact in requirements elicitation regards software architectures. A software architecture is a fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution [23]. There are approaches to support the design of software architectures, like RSEB [24], FAST [25], FORM [26], KobrA [27] and QADA [28]. The product-level perspective of the 4SRS [29] method also promotes functional decomposition of software systems, resulting in a logical architecture. A logical architecture can be considered a view of a system composed of a set of problem-specific abstractions supporting functional requirements [10].

However, the software logical architecture specification should not be the only information passed on to implementation teams. Krutchen's work [6] refers that the description of the architecture can be represented into four views: logical, development, process and physical. The fifth view is represented by selected use cases or scenarios. A set of specifications regarding multiple views, as the CMV, the multi-view artifact proposed in this paper, adds more representativeness value to the specific model.

A component interface document must include [1]: (1) a complete description of the component s inputs (their type); (2) a complete description of the component s outputs (their type); and (3) a description of the relation of the value of each output to the history of the values of the inputs. The range of each relation will be the set of possible values for the associated output variable. The domain of each relation must be a subset of the set of all possible histories for that component. A component interface description of an implementation of a module is a component interface document that characterizes the set of traces that are possible with that implementation. A software component forms the basic unit for reuse and, as part of a software system, is encapsulated by a formal interface. To build a complete working system, developers use a set of components that communicate through their interfaces. However, since effective reuse must be a planned engineering goal, software components must be intended for reuse from the beginning, designed and implemented with their future users in mind. Good documentation facilitates communication between a component's creator and its users, providing insight into design intent, use cases, and potential problems [3].

A. Background

A typical business software development project is developed so that the resulting product properly aligns with the business model intended by the leading stakeholders. In situations where organizations focused on software development are not capable of properly eliciting requirements for the software product, due to insufficient stakeholder inputs or some uncertainty in defining a proper business model, a process-level requirements elicitation is an alternative approach. In the V+V process [4], the requirements are expressed through logical architectural models and stereotyped sequence diagrams in both a process- and a product-level perspective. The first execution of the V-Model acts in the analysis phase and regards a process-level perspective. The first execution of the V-Model (*i.e.*, the process-level perspective) is out of the scope of this paper.

The second V-Model (at product-level), previously presented in Fig. 1, is composed by *Mashed UC*s model [9], *A-* and *B-type* sequence diagrams [8], and *Use Case* models (UCs) that are used to derive (and, validate) a product-level logical architecture (*i.e.*, the software logical architecture). The product-level V-Model enables the transition from analysis to design trough the execution of the product-level 4SRS method (see [29] for details).

As recommended by the ARID [30] method, the V-Model process is able to conduct reviews regarding architectural decisions, namely on the quality attributes requirements and their alignment and satisfaction degree of specific quality goals that are imposed to the created scenarios (*A-Type* sequence diagrams). Those requirements were imbued in design decisions related to the logical architecture. This approach assures the continuous execution of validation tasks along the modeling process. It allows for validating: (1) the final software solution according to the initial expressed business requirements; (2) the *B-type* sequence diagrams according to *A-type* sequence diagrams; (3) the logical architectures by traversing it with *B-type* sequence diagrams. It is a common fact that domain-specific needs, namely business needs, are a fast changing concern that must be tackled. Process-level architectures must be in a way that potentially changing domain-specific needs are local in the architecture representation. Our proposed V-Model approach encompasses the derivation of a logical architecture representation that is aligned with domain-specific needs and any change made to those domain-specific needs is reflected in the logical architectural model through successive derivation of the supporting models (*Mashed UC*s, *A-* and *B-Type* sequence diagrams, and Use cases). In addition, traceability between those models is built-in by construction and intrinsically integrated in our V-Model approach.

At the end of the product-level V-Model execution, the resulting information regards a *Context for Product Implementation* (CPI) model. The CPI information is constructed upon the product-level logical architecture and *B-type* sequence diagrams. Implementation teams may not yet have the CPI information. Delivering a logical architecture composed by more than one hundred architectural elements overwhelms the team's perception of their work, thus the architecture should be modularized and the information arranged (regarding the given module only) to provide an answer to the 'who?', 'what?', and 'why?'. Complex architectures should be modularized into small projects in order not to endanger the software implementation project. The CPI should then be divided and then each division distributed to multiple implementation teams, each one assigned to implement one or more modules, and each team receiving multi-view information (as presented in Fig. 1, at the end of the V-Model). Such information can be included in an artifact as the one presented later in this paper, called *Combined Multiple View* (CMV).

III. Logical Software Architecture Modularization

The logical architecture (Fig. 2) is one of the resulting artifacts from the V+V process. Its architectural elements (AEs) and associations between the AEs are derived from the execution of the 4SRS method [29]. An AE is a representation of the pieces from which the final logical architecture is built. We use the term AES to distinguish them from the components, objects or modules used in other contexts, like in the UML structure diagrams. The AEs, the associations between them and the packages are resulting from the 4SRS method execution.

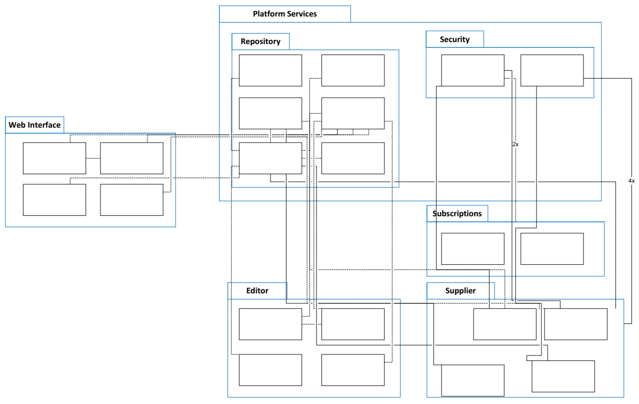


Fig. 2. An example of a logical architecture

Dealing with a logical architecture composed by more than one hundred AEs is an overwhelming task for implementation teams. In our implementation context, we only have Scrum [7] teams that by its nature are usually small. These are the reasons to perform a modularization of the architecture and deliver the information to multiple implementation teams instead of delivering the whole architecture to a single team. In this section we do not propose a well-defined set of rules that architecture modularization decisions should be based on. Instead, we believe that an analyst or manager with knowledge regarding the whole architecture is able to identify the modules, and thus compose it by selecting the AEs from the logical architecture. To support that selection, we identify some concerns that should be considered at the time of the modularization.

The analyst or manager that has the overall insight on the project context and of the final product processes' must carry out the task of identifying the modules. There is no automatism for the module identification so it must be manually executed. The mindset of the analyst or manager has an impact in the resulting set of modules. The identification of a given module is based on the recognition of a specific need regarding its execution. These needs can be built on the identification of *concerns* that are dealt within the architecture execution context (in the case of aspect-oriented programming), or by the identification of software features (in the case of feature-oriented programming). If the mindset is oriented for agile development, the module identification will encompass a set of requirements specification (*i.e.*, user stories). Additionally, issues relating the modules' future distribution by the teams are considered, like if the teams are physically separated and/or with different working cultures and skills.

Regarding issues that occur in a product software development process that relate to standard, platform or customized products [31], variability in requirements can be considered in the modularization decisions (especially the variability inherent to the overlapped AEs, due to the work coordination and communication required between teams). Variability is exposed in functional requirements for product line architectures [11]. The module identification can be performed by estimating the effort for their development, as in [20], but it needs to be adapted since we are dealing with AEs instead of use cases. Although the 4SRS method provides traceability from the use cases to the AEs, it is not certain that the approach could be directly applied: since the decisions regarding the module identification are not based on well-defined rules, the identification may be performed by using a fully ad-hoc approach. If the identification is performed by the same element(s) that executed the V+V process, it is believable that such element(s) possess the required information for the module identification, as they are aware of the architecture(s) big picture. The project stakeholders must assess and validate the module identification.

Regarding the CPI, there are two main artifacts that can be used for the module identification: the software logical architecture and the *B-type* sequence diagrams. The software logical architecture is the artifact typically delivered to implementation teams, and it regards the architecture diagram that is modularized in this task. Besides the AEs that will compose the module, the packages are also used to identify a module. One of more packages composes a module. A given package can also be included in one or more modules.

The other main artifact used regards *B-type* sequence diagrams. The *B-type* sequence diagrams represent scenarios of using the software architecture. These scenarios are the basis for identifying the required software modules. A given *B-type* sequence diagram allows identifying the need for developing a given module that responds to the execution requirements of the given scenario. However, there is not a direct relation from one *B-type* sequence diagram to a given module. A module can be composed by one or more *B-type* sequence diagrams. Additionally, a given *B-type* sequence diagram may be included in one or more modules. In its essence, a module is identified as a requirement for operationalizing a given *B-type* sequence diagram and/or the required packages.

We present in Fig. 3 an example of an architecture modularization, where the architectural elements (representing software functionalities) that compose it, were classified as belonging to a given software module, and thus covered by the module area (identified with a given color). The architecture in this example is purely for representation purposes. There is no need of modularizing such architecture with so few AEs.

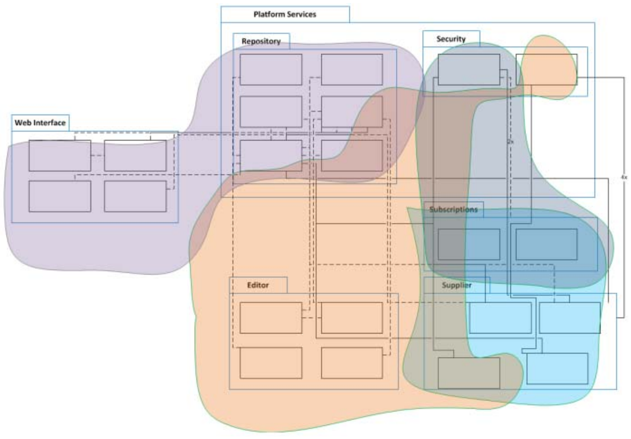


Fig. 3. Architecture modularization example

The subsystems that compose the overall architecture are the identified modules that require implementation. It is irrelevant the number of implementation teams, since one team is assigned to implement one or more modules. After the module is assigned to the team, they must consider the module as the system (leaving the subsystem perspective), with no concerns regarding other modules (except if dependencies/interfaces between modules are identified, as discussed in the next section). Since multiple teams are working for the same product development, the implemented modules must execute altogether to obtain the whole software solution, *i.e.*, the sum of the individual modules must result in the whole architecture.

IV. Combined Multiple Views

More than just providing to implementation teams the information of which functionalities (by the architectural elements) compose the software module, implementation teams must be aware of the overall context where the module will execute. Having in mind three types of information implementation teams should have ('who?', 'what?', and 'why?'), the software module itself only can respond to the 'what?' question, since the only included information regards the module's AEs and their specification. From the integrator's point of view, component documentation should provide information to assist in [14]:

- the selection of components,
- the validation of interoperability of components,
- the analysis of combined quality attributes, and maintenance of components.

We propose in this section an organization of the information regarding multi-view perspective of the software module to be delivered to implementation teams, the CMV. The CMV provides the development teams not only the information regarding the functionalities of the module but also to describe how and in which scenarios they will be used by their future users. This artifact is composed by the AEs that compose the module and their interfaces with external modules (the technique for obtaining the AEs and their interfaces is presented further in this section).

Additionally, the CMV includes information regarding the modules usage in the real world. Such information may be described though one or more scenarios, or by including a scenario representation through a *B-type* sequence diagram previously modeled during the product-level V-Model. It is not automatic that a *B-type* sequence diagram originates the module, because the module can be defined based on more than one *B-type* sequence diagram. One of those previously modeled *B-type* sequence diagram may be included in the CMV artifact, but it is advisable that the selected diagram encompasses as much as functionalities as possible, in order to provide the implementation team with much context information as possible. An example of a CMV is depicted in Fig. 4.

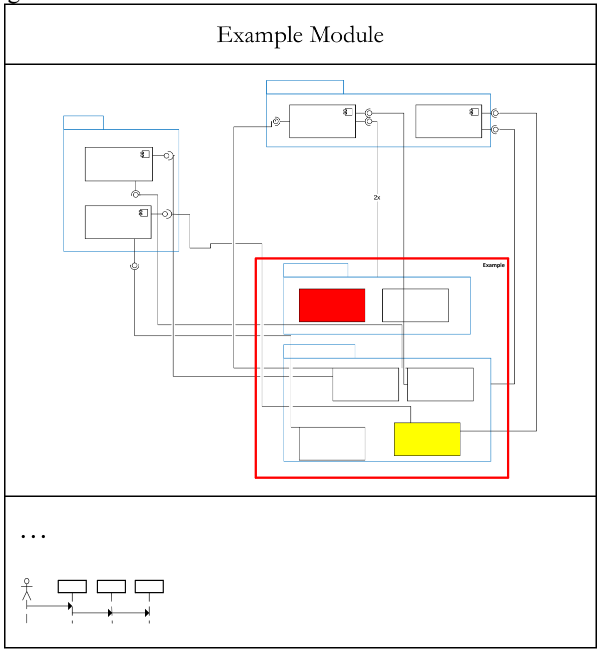


Fig. 4. Combined Multiple View Example

The most critical aspect is the component-based diagram. The logical architecture provides an organized vision of how

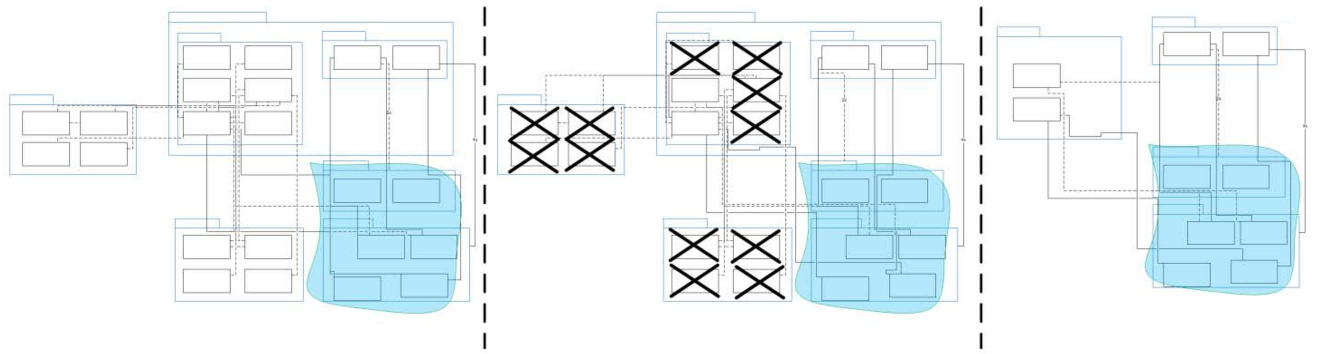


Fig. 5. The resulting AEs from filtering and collapsing technique

functionalities (through its AE's specifications) behave regardless of the technologies to be implemented. It is not possible to depict sequences and flows for the application execution as well as the components that interface with the application by just representing the architectural elements from the logical software diagram that compose the module.

We applied filtering and collapsing techniques [32] to the border that relates to the architectural elements within the module. These techniques redefine the system boundaries, which now regards only the given module as a subsystem for design. During the filtering process, all entities not directly connected to the module must be removed from the resulting filtered diagram.

Inside the system border defined for the given module, through the respective coverage, the architectural elements were maintained as originally characterized. This is due the fact that the product-level architectural elements represent a given functionality or set of functionalities and not necessarily a software component. We depict the filtering and collapsing process in Fig. 5, where the module is chosen (from the architecture in Fig. 2), the AEs with direct connections to the module are maintained, and the ones without direct connection are removed. On the other hand, for representing the interfaces that are outside the system border, we adopted the UML notation for components, to represent inputs and outputs of the functionality. The component-based diagram uses a typical representation of UML component graphic nodes [12]. A connector may be notated by a "ball-and-socket" connection between a provided interface and a required interface. The final representation of the modules' AEs and interfaces is presented in Fig. 6, and it is this representation that is included in the CMV depicted in Fig. 4.

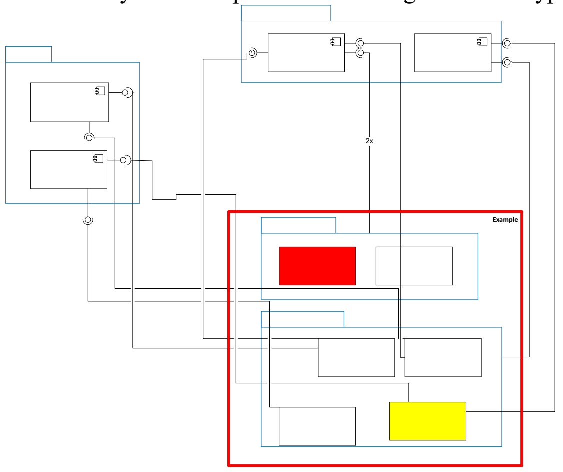


Fig. 6. The module representation to include in the CMV

The architecture modularization technique (through architectural elements coverage) also allows depicting in the diagram the existence of architectural elements that execute in more than one module (the architectural elements where there is an overlap of module coverage). Such situation requires that their implementation must be properly managed and coordinated by the development teams, *i.e.*, its development must be agreed by the teams. In order to facilitate the identification of such situations, which makes the implementation of the given architectural element more critical than the ones that are part of only one module, those architectural elements are identified using colors. Critical architectural elements can be identified, for instance, in yellow if they are executed in two modules or red if they are executed in three modules. It is advisable that, as more critical the architectural element is considered, the "stronger" should be the color used.

V. The ISOFIN Project

The ISOFIN (Interoperability in Financial Software) [33] architecture aims to deliver a set of cloud-based functionalities enacting the coordination of independent services relying on private clouds. The resulting ISOFIN platform will allow the semantic and application interoperability between enrolled financial institutions (Banks, Insurance Companies and others).

The global ISOFIN architecture relies on two main service types: Interconnected Business Service (IBS) and Supplier Business Service (SBS). IBSs concern a set of functionalities that are exposed from the ISOFIN core platform to ISOFIN Customers. An IBS interconnects one or more SBS's and/or IBS's exposing functionalities that relate directly to business needs. SBS's are a set of functionalities that are exposed from the ISOFIN Suppliers production infrastructure. Fig. 7 encompasses the primary constructors (IBS, SBS and the ISOFIN Platform) available in the logical representations of the system: in the bottom layer there are SBSs that connect to

IBSs in the ISOFIN Platform layer and the later are connected to ISOFIN Customers.

This project is executed in a consortium comprising eight entities (private companies, public research centers and universities), making the requirements elicitation and the definition of a development roadmap difficult to agree. The initial request for the project requirements resulted in mixed and confusing sets of misaligned information. Due to the lack of consensus in the requirements elicitation in this “newfound” paradigm of IT solutions (Cloud Computing), our approach changed the traditional product-level perspective to the described process-level perspective.

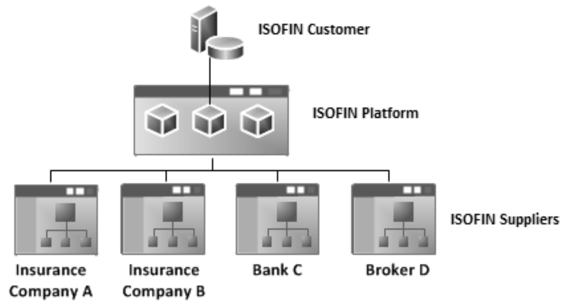


Fig. 7. Desirable Interoperability in ISOFIN

Our proposal of adopting a process-level perspective was agreed on and, based on the knowledge that each consortium member had of the intended project results, the major processes were elicited and a first approach to a logical (process-level) architecture was made. After execution of the process-level perspective, it was possible to gather a set of information that the consortium is sustainably using to evolve to the traditional (product-level) development scenario. The *Mashed UC* model regards the first model to execute the product-level V-Model, and was used as input for the successive derivation of the rest of the models that compose the V-Model. Once the V-Model was applied, *A*- and *B-type* sequence diagrams and the product-level logical architecture were derived. The execution of the product-level 4SRS method resulted in a logical architecture (the architecture from Fig. 8) composed by more than one hundred architectural elements (depicted in the zoomed area from Fig. 8).

Although there weren’t any metrics considered or complexity study performed, it is clear that so many functionalities are unadvisable to provide to a single implementation team. Thus, a modularization of the architecture was mandatory in order to divide and distribute module implementations for more than one implementation team. The scenarios considered for modeling the stereotyped sequence diagrams during the product-level V-Model and the identification of *concerns* that are dealt within the architecture execution context allowed the identification of candidate modules. The architecture modularization is represented by the set of colored areas that traverse the ISOFIN logical architecture in Fig. 9. There only exist situations where some AEs are part of two modules or, at most, three modules. The AEs to be executed in two modules are marked in yellow, and the ones to be executed in three modules are marked in red.

We present in this section one of the modules to illustrate the demonstration case: the IBS Management module. By executing this module, An IBS Developer develops a new IBS by modeling the IBS, selecting the available IBSs and SBSs from the pallets. Besides modeling its structure, the IBS Developer is also responsible for defining permissions, manually filling gaps in the IBS code, publishing the information in the catalog and deploying the IBS in the ISOFIN Platform.

The filtering and collapsing technique that was applied

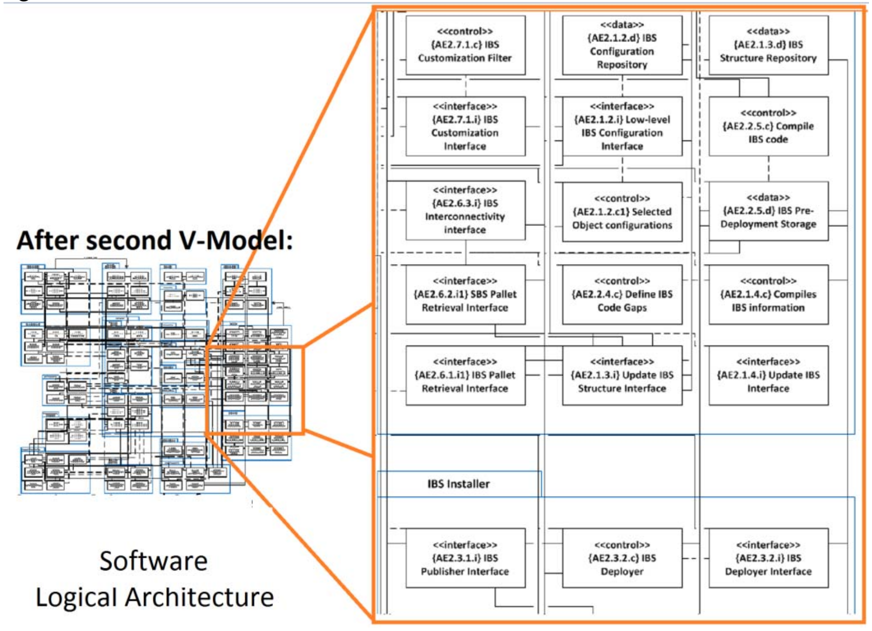

After second V-Model:
Software Logical Architecture
<<control>> {AE2.7.1.c} IBS Customization Filter
<<data>> {AE2.1.2.d} IBS Configuration Repository
<<data>> {AE2.1.3.d} IBS Structure Repository
<<interface>> {AE2.7.1.i} IBS Customization Interface
<<interface>> {AE2.1.2.i} Low-level IBS Configuration Interface
<<control>> {AE2.2.5.c} Compile IBS code
<<interface>> {AE2.6.3.i} IBS Interconnectivity interface
<<control>> {AE2.1.2.c1} Selected Object configurations
<<data>> {AE2.2.5.d} IBS Pre-Deployment Storage
<<interface>> {AE2.6.2.i1} SBS Pallet Retrieval Interface
<<control>> {AE2.2.4.c} Define IBS Code Gaps
<<control>> {AE2.1.4.c} Compiles IBS information
<<interface>> {AE2.6.1.i1} IBS Pallet Retrieval Interface
<<interface>> {AE2.1.3.i} Update IBS Structure Interface
<<interface>> {AE2.1.4.i} Update IBS Interface
IBS Installer
<<interface>> {AE2.3.1.i} IBS Publisher Interface
<<control>> {AE2.3.2.c} IBS Deployer
<<interface>> {AE2.3.2.i} IBS Deployer Interface

Fig. 8. The ISOFIN logical software architecture resulting from the product-level V-Model

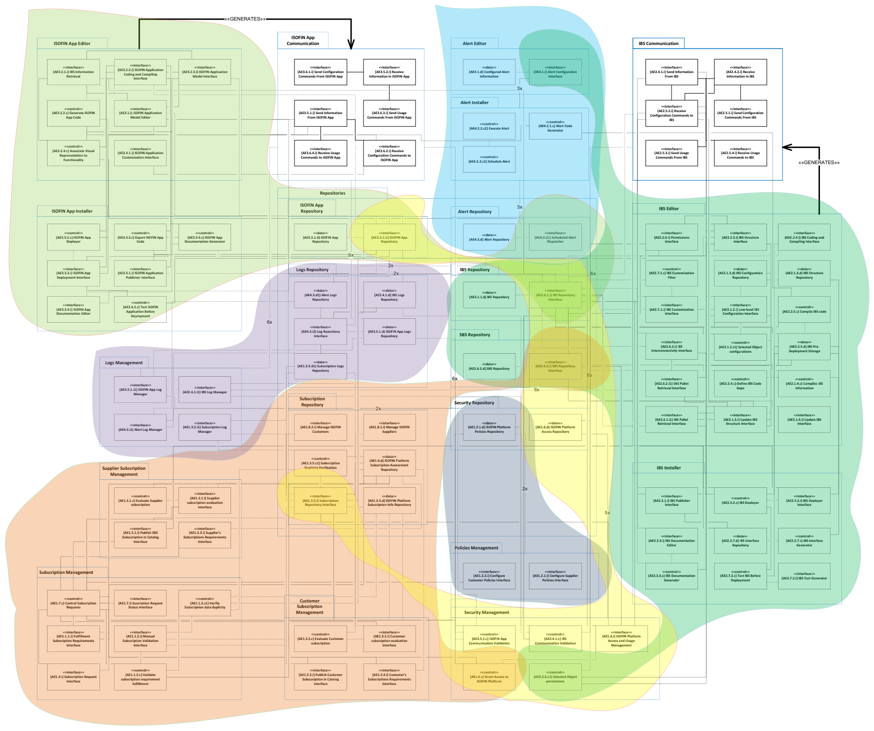


Fig. 9. ISOFIN architecture modularization

allowed depicting the AEs that compose the module and their interfaces (components), and then depict its CMV. For readability purposes, the module (AEs and their interfaces) is presented in Fig. 10. The CMV for the IBS Management module is presented in Fig. 11. Regarding the overlapped AEs in more than one module, one of the teams is nominated to be responsible for implementation of the module and assure that the teams responsible for modules with dependencies with that particular one have all required documentation and provide updates on its implementation.

If instead of providing only a module, the whole architecture was provided to a single team, we believe the project would surely have its success execution endangered. By dividing work by distributed teams working in parallel, also the complexity of the project was divided into small complexity sub-projects, decreasing the overall project's time to market. The CMV for the IBS Management module provides thus the module functionalities (the architectural elements and their specifications) resulting from the 4SRS method execution, as well as the component interfaces resulting from the filtering and collapsing technique – the 'what?'; the actors that execute the functionalities – the 'who?'; and one (or more) scenario description of the usage of the module from the *B-type* sequence diagrams – the 'why?'.

VI. Conclusions

A software architecture is a crucial requirements artifact, but the development work for implementing a complex architecture may require the identification of modules to deliver to a single or to multiple implementation teams. The module identification is not yet performed by effort estimation or well-defined rules, but it is clear that complex architectures should not be delivered entirely to a single implementation team or, at least, the entire architecture should not be implemented as a whole but instead as small sub-systems (i.e., the modules) in order to reduce the teams' work time. If instead of providing only a module, the whole architecture was provided to a single team, we believe the project would surely have its success execution endangered. By dividing work by distributed teams working in parallel, also the complexity of the project was divided into small complexity sub-projects, decreasing the overall project's time to market.

Since the software architecture was derived through successive models from a process- perspective to a product-level perspective, its specification does not always provide sufficient information to implementation teams. We propose in this paper a multi-view artifact to be delivered to a software implementation team. This artifact provides information regarding not just what the software must do, but also by whom and why. Our combined multiple view approach can be seen as a pattern for component documentation. The information contained in the CMV description complies with the information suggested by [14] for pieces of information that a component documentation should possess as basic information:

- the identification, including the identification number and the name of the component,
- the type of the component,

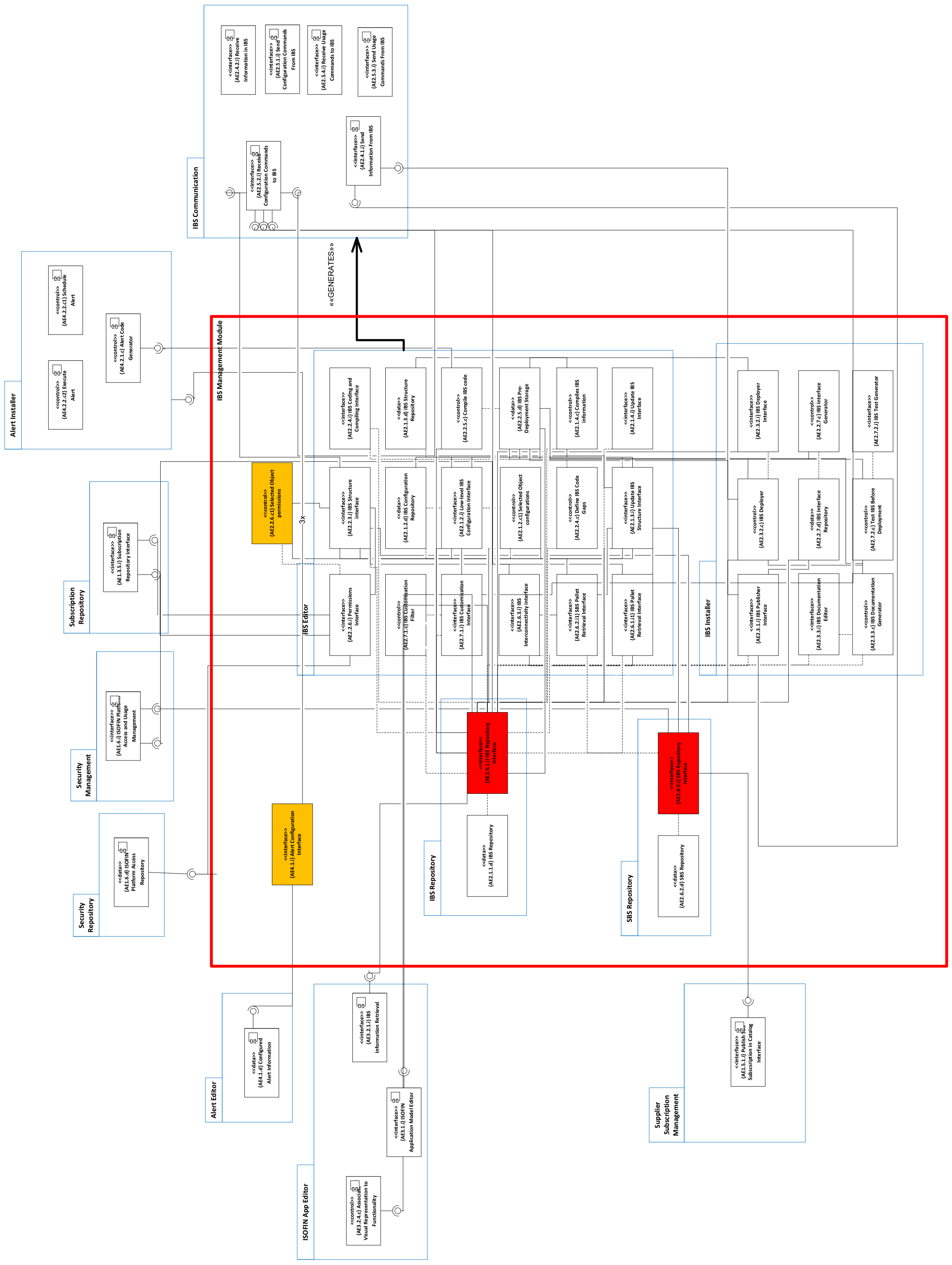
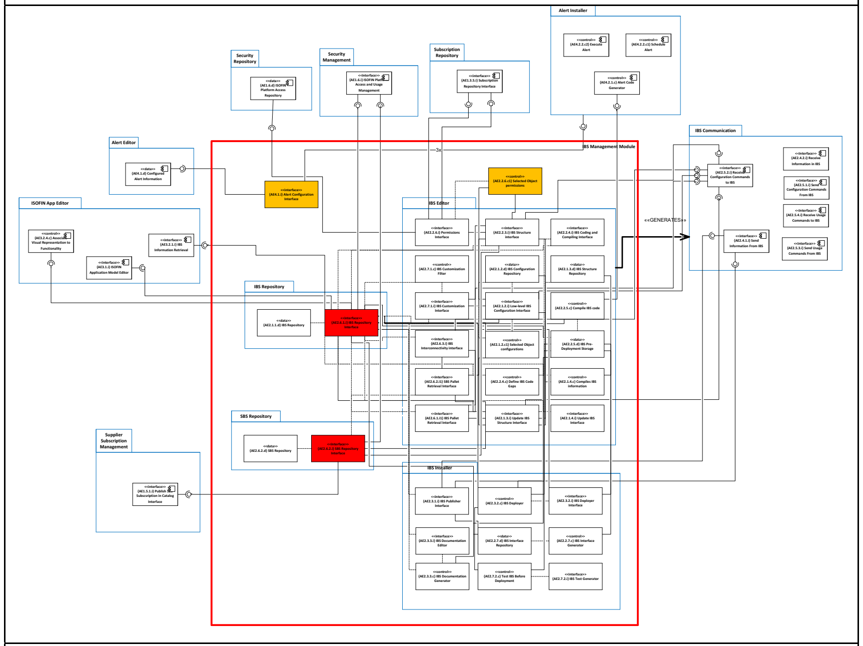


Fig. 10. IBS Management module

IBS Management Module



An IBS Developer develops a new IBS by modeling the IBS, selecting the available IBSs and SBSs from the pallets. Besides modeling its structure, the IBS Developer is also responsible for defining permissions, manually filling gaps in the IBS code, publishing the information in the catalog and deploying the IBS in the ISOFIN Platform. If necessary, the IBS Developer performs test and fixes coding errors. The IBS to be developed may require setting system alerts.

Fig. 11. IBS Management CMV

— the overview of the component,
— history of the component, and the special terms and rules used in the context.

Additionally, the CMV description allows identifying interfaces with other modules, which provides a clear identification of module points that will require coordination between teams, and provides a description of a scenario in which the module is executed, assisting in an overall understanding of the context of the module's usage. This way, development teams are provided with a set of information that only concerns the product subset that they will develop, as well as information regarding contact points with other modules. The identification of contact points with other modules provide information to be consumed by development teams. The module interfaces allow depicting the need to development a module with interfaces and connections with given software components, and the clear identification of dependencies of the module execution with other modules. The dependency identification information is also to be consumed by project management or any other person responsible for team coordination (*e.g.*, the product owner in Scrum methodology), since it identifies needs to coordinate work of separate teams.

The CMV also allows depicting the existence of overlapped functionalities regarding more than one module. This information is also consumed by project management or any other person responsible for team coordination, requiring a coordination between management in order to nominate the team to develop the overlapped functionality. The overlapping and the module interfaces are useful in order to assure the multiple teams work in a well-defined development chain and teams whose work is co-dependent cooperate with each other.

The module identification was made using the scenarios considered for modeling the stereotyped sequence diagrams during the product-level V-Model and the identification of *concerns* that are dealt within the architecture execution context, by analyzing the whole logical architecture and execution scenarios through one or more *B-type* sequence diagrams. For defining the modules, the composition of which AEs can be made by estimating the effort of development of each functionality. The 4SRS method promotes the inclusion of unforeseen requirements by making multiple iterations and also provides traceability between use cases and AEs. As future work, it is intended to study how AEs can be used for effort estimating, the same way Use Case points are. It is also intended to study the promotion of the architecture documents update after its first release and the definition of CMV implementation priorities

CMV artifacts, through the AEs and *B-type* sequence diagrams, provide information regarding the 'who?', 'what?', and 'why?' on modules' software requirements, which is the required information to be delivered to Scrum teams, for instance. The purpose of the CMV artifact is to provide information regarding software functionalities, regardless of the implementation techniques and technologies (*e.g.*, programming languages) the team uses. This means that functional requirements were elicited, but technical decisions will be made by implementation teams. The CMV provides information about software functionalities of the module, interfaces with other modules, and the context of the module's usage. Decisions regarding messaging, protocols, amongst others, are made by the implementation team (or by the person responsible, like a project manager or a product owner). Nevertheless, the CMV artifact provides functional and behavioral information of the module in design that will later support the technical specifications of the modules. These issues are addressed in the description of software functionalities, interfaces between modules and description of the context in use, since these information assists on identifying the technological requirements. Thus, the inclusion of other type of information or models, like, for instance, data information, technical and platform-specific information, will also be analyzed in future works.

ACKNOWLEDGMENT

This work has been supported by FCT – Fundação para a Ciência e Tecnologia in the scope of the project: Pest-OE/EEI/UI0319/2014.

REFERENCES

[1] D.L. Parnas, "Component Interface Documentation: What do we Need and Why do we Need it?," IOS Press, 2006.

[2] D.L. Parnas and J. Madey, "Functional documents for computer systems," Science of Computer programming, vol. 25, no. 1, 1995, pp. 41-61.

[3] J. Kotula, "Using patterns to create component documentation," Software, IEEE, vol. 15, no. 2, 1998, pp. 84-92; DOI 10.1109/52.663791.

[4] N. Ferreira, N. Santos, P. Soares, R.J. Machado and D. Gasevic, "Transition from Process- to Product-level Perspective for Business Software," Proc. CONFENIS'12, 2012 pp.

[5] C. Haskins and K. Forsberg, "Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities; INCOSE-TP-2003-002-03.2. 1," Proc., INCOSE, 2011, pp.

[6] P. Kruchten, "The 4+1 View Model of Architecture," IEEE Softw., vol. 12, no. 6, 1995, pp. 42-50; DOI 10.1109/52.469759.

[7] K. Schwaber, "Scrum development process," Business Object Design and Implementation, Springer, 1997, pp. 117-134.

[8] N. Ferreira, N. Santos, R. Machado, J.E. Fernandes and D. Gasević, "A V-Model Approach for Business Process Requirements Elicitation in Cloud Design," Advanced Web Services, A. Bouguettaya, et al., eds., Springer New York, 2014, pp. 551-578.

[9] N. Ferreira, N. Santos, P. Soares, R. Machado and D. Gašević, "A Demonstration Case on Steps and Rules for the Transition from Process-Level to Software Logical Architectures in Enterprise Models," The Practice of Enterprise Modeling, Lecture Notes in Business Information Processing 165, J. Grabis, et al., eds., Springer Berlin Heidelberg, 2013, pp. 277-291.

[10] S. Azevedo, R.J. Machado, D. Muthig and H. Ribeiro, "Refinement of Software Product Line Architectures through Recursive Modeling Techniques " Proc. On the Move to Meaningful Internet Systems: OTM 2009 Workshops, Springer Berlin / Heidelberg, 2009, pp. 411-422.

[11] A. Bragança and R.J. Machado, "Deriving Software Product Line's Architectural Requirements from Use Cases: An Experimental Approach," Proc. 2nd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, Rennes, France, 2005, pp.

[12] OMG, "Unified Modeling Language (UML) Superstructure Version 2.4.1," 2011; http://www.omg.org/spec/UML/2.4.1/.

[13] M. Fowler, UML distilled: a brief guide to the standard object modeling language, Addison-Wesley Professional, 2004.

[14] A. Taulavuori, E. Niemelä and P. Kallio, "Component documentation—a key issue in software product lines," Information and Software Technology, vol. 46, no. 8, 2004, pp. 535-546; DOI 10.1016/j.infsof.2003.10.004.

[15] C. Szyperski, Component software: beyond object-oriented programming, Pearson Education, 2002.

[16] D.N. Card, G.T. Page and F.E. McGarry, "Criteria for software modularization," Proc. Proceedings of the 8th international conference on Software engineering, IEEE Computer Society Press, 1985, pp. 372-377.

[17] Y. Cai, "Assessing the Effectiveness of Software Modularization Techniques through the Dynamics of Software Evolution," Contemporary Modularization Techniques (ACoM. 08), 2009, pp. 40.

[18] B. Anda, H. Dreiem, D.I. Sjøberg and M. Jørgensen, "Estimating software development effort based on use cases—experiences from industry," ≪ UML≫ 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Springer, 2001, pp. 487-502.

[19] G. Karner, "Resource estimation for objectory projects," Objective Systems SF AB, vol. 17, 1993.

[20] S. Nageswaran, "Test effort estimation using use case points," Proc. Quality Week, 2001, pp. 1-6.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, Aspect-oriented programming, Springer, 1997.

[22] C. Prehofer, "Feature-oriented programming: A fresh look at objects," ECOOP'97—Object-Oriented Programming, Springer, 1997, pp. 419-443.

[23] IEEE Computer Society, IEEE Recommended Practice for Architectural Description of Software Intensive Systems - IEEE Std. 1471-2000, 2000.

[24] I. Jacobson, M. Griss and P. Jonsson, Software Reuse: Architecture, Process and Organization for Business Success, Addison Wesley Longman, 1997.

[25] D.M. Weiss and C.T.R. Lai, Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley Professional, 1999.

[26] K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin and M. Huh, "FORM: A feature-oriented reuse method with domain-specific reference architectures," Annals of Sw Engineering, 1998.

[27] J. Bayer, D. Muthig and B. Göpfert, "The library system product line. A KobrA case study," Fraunhofer IESE, 2001.

[28] M. Matinlassi, E. Niemelä and L. Dobrica, Quality-driven architecture design and quality analysis method, A revolutionary initiation approach to a product line architecture, VTT Tech. Research Centre of Finland, 2002.

[29] R.J. Machado, J.M. Fernandes, P. Monteiro and H. Rodrigues, "Transformation of UML Models for Service-Oriented Software Architectures," Proc. ECBS'05, IEEE Computer Society, 2005, pp. 173-182.

[30] P.C. Clements, Active Reviews for Intermediate Designs., Technical Note CMU/SEI-2000-TN-009., 2000.

[31] P. Artz, I. Weerd, S. Brinkkemper and J. Fieggen, "Productization: transforming from developing customer-specific software to product software," Software Business, 2010, pp. 90-102.

[32] R.J. Machado, J. Fernandes, P. Monteiro and H. Rodrigues, "Refinement of Software Architectures by Recursive Model Transformations," Proc. Product-Focused Software Process Improvement, Springer Berlin / Heidelberg, 2006, pp. 422-428.

[33] ISOFIN Consortium, "ISOFIN Research Project," 2010; http://isofincloud.i2s.pt.