A modeling language to visually represent architectural design decisions and their *rationale*

Un lenguaje de modelado para representar visualmente las decisiones de diseño arquitectónico y su rationale: *rationale*

Milton Sanchez-Grueso[1]
Julio Hurtado-Alegría[2]

[1]Universidad del Cauca (Colombia). Email: sanchezg@unicauca.edu.co
ORCID: https://orcid.org/0000-0003-2109-4969

[2]Universidad del Cauca (Colombia). Email: ahurtado@unicauca.edu.co
ORCID: http://orcid.org/0000-0002-2508-0962

Received: 14-01-2020 Accepted: 30-07-2020

How to quote: Sánchez-Grueso, Milton; Hurtado-Alegría, July (2020). A modeling language to visually represent architectural design decisions and their *rationale*. *Informador Técnico*, *84*(2), 155-174.
https://doi.org/10.23850/22565035.2622

Abstract

The architectural rationale is the set of reasons behind the decisions made when designing the architecture of a software system. Normally, this *rationale* remains in the minds of designers and others involved in the design. Therefore, the reasoning behind the decisions that underpin the architecture model may be lost if not properly documented, later causing maintainability problems in the software. In practice, the *rationale* is not documented or is documented in the middle of the architectural descriptions, which hinders the understanding and support of subsequent decisions, within the development and maintenance of software, which is more critical in the agile approach to development. To address this problem, this paper proposes a documentation approach that combines rationale modeling with a focus on decisions made in projects that use agile methods, to specify a language that provides the basis for the construction of a tool that we have called DRML (Decisions and *Rationale* Modeling Language). The language is evaluated to record the *rationale* in the framework of the Unique Indigenous Information System (SUIIN, for its acronym in Spanish) project, in the context of a public entity that within its processes has a work team comprised of systems engineers focused on software development. This evaluation has established that the approach provides sufficient expressiveness to document the decisions and their rationale, however, it has limitations to scale the modeling for a large number of decisions and their relationships.

Keywords: *rationale*; software architecture; architectural design decisions; decision model; architectural *rationale*.

Resumen

El rationale arquitectónico es el conjunto de razones detrás de las decisiones tomadas al diseñar la arquitectura de un sistema de software. Normalmente, dicho *rationale* se queda en las mentes de los diseñadores y demás involucrados en el diseño. Por lo tanto, el razonamiento detrás de las decisiones que sustentan el modelo de arquitectura puede perderse si no se documenta adecuadamente, causando problemas de mantenibilidad en el software. En la práctica, el *rationale* no se documenta o se documenta en medio de las descripciones arquitecturales, lo cual dificulta su comprensión y apoyo a las decisiones posteriores, dentro del desarrollo y mantenimiento de software, lo cual resulta más crítico en el enfoque ágil de desarrollo. Para abordar este problema, en este trabajo se propone un enfoque de documentación que combina el modelado del rationale, con foco en las decisiones que se toman en proyectos que utilizan métodos ágiles, con el fin de especificar un lenguaje que fundamente las bases para la construcción de una herramienta que hemos denominado Decisions and *Rationale* Modeling Language (DRML). El lenguaje es evaluado para documentar el *rationale* en el marco del proyecto "Sistema Único de Información Indígena (SUIIN)", en el contexto de una entidad pública que dentro sus procesos tienen un equipo de trabajo conformado por ingenieros de sistemas, enfocados en el desarrollo de software. Dicha evaluación ha permitido establecer que el enfoque brinda la suficiente expresividad para documentar las decisiones y su *rationale*, sin embargo, presenta limitaciones para escalar el modelado para un número grande de decisiones y sus relaciones.

Palabras clave: rationale; arquitectura de software; decisiones de diseño arquitectónico; modelo de decisiones; rationale arquitectónico.

1. Introduction

Software architecture has been identified as an artifact of great value for the evolution of software systems. However, it is difficult for software organizations to find the architecture document useful, because it is usually incomplete, or it is very extensive, or incomprehensible to the different stakeholders. Therefore, there is usually an important gap between the potential value and the real value of software architectures for organizations. In agile projects, the way of designing the architecture changes, because it is not a one-person activity, consequently, design decisions are made in a group and are focused on the continuous delivery of the product within a short time. Despite this, the teams make an effort to combine architectural practices with agility, in order to obtain quality products in a very short time (Lopes; Aquino, 2017). In addition, there are problems during the design of architecture, related to the early making of design decisions, with limited rationale and cognitive biases about the problem being solved, as well as the impact of these first decisions on the rest of the development (van Vliet; Tang, 2016). There are myths and beliefs about software architecture, supported by success or failure stories, ranging from the amount of effort needed for documentation, to the size of the team or the people responsible for making architectural design decisions. Most beliefs are based on the idea that the outcome of the project depends largely on the methods used during design and decision-making (van Der Ven; Bosch, 2016). During the creation of software architecture, architects and other stakeholders make decisions. These decisions can be directly related to functional or quality requirements. Some design decisions are made on the fly, more or less arbitrarily, following personal experience, domain knowledge, financial constraints, and available experience, inter alia. Decisions, as well as the reasons for those decisions, are often not explicit, they are rather implicit and usually remain undocumented (Roeller; Lago; van Vliet, 2006).

A way to facilitate bidirectional traceability between quality aspects, architectural decisions, *rationale*, and affected modules, including a code perspective, may provide critical support for several areas of the software engineering process. Some of the items that list of areas currently miss are the change impact analysis, requirements validation, architectural preservation, construction of safety cases and, in the long term, system maintenance. For example, practice has shown that architecture erosion often occurs when developers change the code, without fully understanding the underlying architectural decisions and quality-related concerns (Cleland-Huang; Mirakhorli; Czauderna; Wieloch, 2013). Unfortunately, nowadays this fact is a reality, since software architecture is defined by many as the composition of a set of architectural design decisions, where the objective is to avoid the evaporation of knowledge on such design decisions, insofar they have become an explicit part of the architecture. This, in turn, shows the high cost of change in software architectures, as well as the complexity of these changes and how these architectures erode during their evolution. The problems are due to loss of consciousness. Currently, information on architectural design decisions is implicitly illustrated, but it lacks explicit representation. Consequently, knowledge about these architectural design decisions disappears.

For example, during the design, development, evolution, reuse, and interpretation, the rationale of the decisions made is beyond the scope of the new decision-making. In design, the main concern is which decision to make. In development, it is important to know what and why certain design decisions have been made. Architectural evolution is all about new design decisions or eliminating old ones to meet changing requirements. The challenge is to do this in harmony with existing design decisions, in particular, to understand the reasons that led to the making thereof (Jansen; Bosch, 2005).

This work proposes an integrated language that may be expressed as architectural design decisions, as well as the *rationale* that supports them within an agile context with software architecture practices. To do this, a model has been defined as abstract syntax and a set of representations as concrete syntax, so that architects can express their decisions and the *rationale* behind them. Furthermore, the meta-model of language was used in the construction of a tool, with model-driven engineering (MDE) approach to facilitate the architect's modeling and storage. The language has been assessed in a pilot test in a software company in the Colombian Southwest, using the proposed modeling tool. The case study has shown that the tool enables its user to express most aspects pertaining to the decisions taken and their rationale, as well as their limitations to scale the models and relating design decisions.

2. Theoretical framework and previous studies

Kruchten, Obbink, and Stafford (2006) define software architecture as the structure and organization of components and subsystems that can interact with each other, in order to form complex systems with a much higher life expectancy. Software architectures are comprised of components, connectors, and constraints represented by different perspectives, called views. These views provide a complementary description of software architecture, which represent the different concerns of software stakeholders, and serve as a means of communication for the architecture and its raison d'être (Roldán; Gonnet; Leone, 2016).

Software architecture designers inevitably work with architectural patterns and tactics. Architectural patterns describe the structure and high-level behavior of software systems as the solution to multiple system requirements, while tactics are design decisions that enhance individual quality attribute problems. Tactics that are implemented in existing architectures can have a significant impact on architecture policies in the system. Similarly, the tactics that are selected during the initial design of the architecture have a significant impact on the architecture of the system to be designed, specifying which patterns to use and how they should be changed to accommodate the tactics (Harrison; Avgeriou, 2010).

Quality attributes are characteristics that the system has, such as usability, maintainability, performance, and reliability. Typically, systems have multiple important quality attributes, and decisions made to satisfy a particular quality attribute can affect another quality attribute. Tactics are also measures taken to improve quality attributes. Tactics impact architectural patterns in a number of ways. In some cases, a tactic can be easily implemented using the same structures (and compatible behavior), such as a particular architecture pattern. On the other hand, a tactic may require significant changes in the structure and behavior of the pattern or completely new structures and behaviors. In this case, the implementation of the tactic and the future maintenance of the system is considerably more difficult and error-prone. Tactics can be "design time" or general design and implementation approaches, such as "hiding information" to improve modifiability, or "runtime tactics," which are features aimed at a particular aspect of a quality attribute, such as "authenticating users" to improve security (Harrison; Avgeriou, 2010).

An architectural design decision is defined as the description of the set of additions, subtractions, and modifications to the software architecture, its logic, design rules, design constraints, and new requirements that result in new decisions (Jansen; Bosch, 2005). On the other hand, the *rationale* or architectural foundation refers to the justification of an architectural design decision. Reynoso (2004) defines *rationale* as the "underlying basis for architecture in terms of constraints arising from system requirements (p.9)". *Rationale*, as a component of software architecture, becomes a critical point for design, by describing decisions, evolving, and making new decisions. Tang, Babar, Gorton, and Han (2006) explore the value of *rationale* from the point of view of design, seeking to document background knowledge with their design decisions and reflect the importance thereof for architects, documenting the rationale as an alternative to analyze decisions.

For the design and documentation of software architecture, architecture definition languages (ADLs) are used, which provide elements to model the conceptual architecture of a software system, thereby distinguishing it from its implementation. Some ADLs are generic, for example, Unified Modeling Language (UML) and others specific to the application domain. The best-known languages are Acme, Aesop, and Sad (Hernández; Hurtado, 2016). As for model-driven engineering (MDE), it is a software development paradigm that aims to raise the level of abstraction, focusing on modeling activities, rather than coding. As per the MDE paradigm, from a model and through transformations, it is possible to automatically obtain a variety of elements, such as new models and code, inter alia. In this context, software development can be seen as a transformation process, where low-level abstraction models are obtained automatically or semi-automatically by transforming high-level abstraction models (Bucaioni; Cicchetti; Ciccozzi; Mubeen; Sjodin 2017).

Aldrich, Chambers, and Notkin (2002) introduced ArchJava, a small Java extension that integrates software architecture specifications into Java implementation code, with the purpose of making the implementation conform to architectural constraints. To evaluate their approach, they implemented the extension to a circuit design application, using an informal hand-drawn diagram of the architecture, which they use as a guide to making this architecture explicit in the code. Finally, they concluded that ArchJava enables programmers to express the architectural structure, which will be later filled into the application with Java code. This work shows the importance of expressing architecture in the code, but uses another approach, that of representing abstractions. Furthermore, it does not consider the *rationale* and architectural design decisions that cannot be represented in the code.

Cleland-Huang *et al.* (2013) present an architecture-centric approach to follow up on stakeholder quality concerns such as reliability, availability, security, integrity, performance, portability, architecturally significant requirements, design reasons, and source code. In decision-centric traceability (DCT), all tracking links focus on architectural decisions that include factors as varied as platforms, languages, frameworks, high-level design patterns, communication mechanisms, low-level architectural tactics, and architectural styles, inter alia. Nowadays, software systems are designed to meet functional requirements as well as a wide scale of quality interests pertaining to the aforementioned attributes (Bass Clements; Kazman 2003). For example, an architect may decide to address a portability goal by using a strictly layered approach, which simplifies the process of creating new platform-specific graphical user interfaces (GUIs), or to achieve an availability goal by utilizing the control tactic to monitor the health status of a critical component or meet a performance goal through the use of a set of threads to manage shared resources (Cleland-Huang *et al.*, 2013). Facilitating two-way traceability between quality concerns, architectural decisions, justifications, and relevant areas of code provides critical support to various disciplines of the software engineering process (Cleland-Huang *et al.*, 2013). For example, the practice has shown that architecture erosion often occurs when developers make changes to code without fully understanding the underlying architectural decisions and their quality-related concerns (Cleland-Huang *et al.*, 2013). The tracking links that are used to communicate the main design decisions at the architectural level are in line with the approach established hereby, which seeks to model the *rationale* and design decisions in an agile context.

Hadar, Sherman, Hadar, and Harrison (2013) performed a case study purported to identify the difficulties that architects and other stakeholders face when documenting architecture in agile development. The findings show that the document containing the architectural specification is usually very extensive, complex, and, in many cases, not easily explainable. In order to align architecture documentation with the lightweight and minimal documentation approach of agile processes, these authors propose a shorter specification document that requires reduced documentation efforts, resulting in simplified documentation that is easier to review, update, and communicate. Architectural documents are usually extensive and complex, and can grow from tens to hundreds of pages; consisting of multiple documents that encompass multiple concepts, relations, views, and levels of abstraction. Jansen, Avgeriou, and van Der Ven (2009) identified a list of challenges related to architecture documentation is based on the following three challenges: first, documents comprehensibility by architects; second, localization of relevant architectural knowledge; and third, keeping architecture documentation up to date.

Hesse, Kuehlwein, Paech Roehm, and Bruegge (2015) studied the importance of a software development team to document implementation decisions. When the code is reviewed during maintenance, the decisions behind the architecture must be understood and possibly adjusted to the current situation. Therefore, knowledge of the decision is seldom documented and thus becomes inaccessible, especially when developers are no longer part of the team. This hinders effective maintenance. To solve this issue, the authors have developed a model of recording for integrated knowledge and decision through a knowledge management tool called UNICASE, an extension of Eclipse that provides a model for documenting knowledge of decisions previously made, especially at the design level. The approach enables developers to document decisions within the code. This project similarly intends to document architectural design decisions through a software plug-in in the eclipse development environment at the modeling level.

Regarding language structuring, only the elements of documentation that were considered relevant for the purpose of the modeling that involves the decisions and their *rationale* to be applied to the agile context have been considered. Therefore, this proposal takes as its starting point the meta-model proposed by Plataniotis, Ma, Proper, and de Kinderen (2015). Their main contribution is a formal meta-model that captures the *rationale* and interrelationships of design decisions; the work of Dorado and Hurtado (2019), where they document *rationale* through annotations in the code; and the definition made in van Der Ven Jansen, Nijhuis, and Bosch (2006), who propose to integrate the rationale and architectural artifacts in a design decision, which in turn combines the software architecture *rationale*. The approach proposed by Gilson and Englebert (2011) considers that the modeling of architecturally significant requirements and architecture modeling, where constraints and requirements are attached to architectural constructions, and any modifications to the architecture model resulting from a decision made from the requirements model, is recorded as a model transformation. Che's (2014) methodology, which explicitly documents architectural design decisions, using a scenario-based approach, encompasses a set of software architecture views to record architectural knowledge, using features focused on managing the evolution of architectural design decisions, in order to reduce the evaporation of associated knowledge. The template proposed by Dermeval *et al.* (2013), captures the rationale of architectural design decisions, relating functional requirements, quality requirements, stakeholders, and *rationale*. Likewise, the proposed model is based on the proposed decision model by Manteuffel, Tofan, Koziolek, Goldschmidt, and Avgeriou (2014), which suggests a tool supported in the ISO/IEC/IEEE 42010 meta-model to document decisions and their implicit knowledge. Finally, it is important to note that software architecture decision-making is not an individual activity, but a general process, where architectural design decisions are made by heterogeneous and dispersed stakeholder groups (Malavolta; Mucdni; Rekha, 2014).

An analysis of the studies was carried out to determine which of these had greater relevance for this research, i.e., that provide conceptual and theoretical elements to build the documentation language for the *rationale* hereunder. Table 1 lists the studies that influenced the construction of language and its most important abstractions.

Table 1.
Works related to rationale and architectural design decisions

DRML language pools	Rationale	Context			Consequence				Alternative	Architectural Decision		Justification			Tactic	Pattern	Strategy
Elements by the study	R	M	P	C	To	Cn	Pr	Cq	Alt	ADD	D	Qa	BR	QG	T	P	E
(Cleland-Huang *et al.*, 2013)	X									X				X			
(van Der Ven *et al.*, 2006)		X	X	X	X	X	X	X									
(Dermeval *et al.*, 2013)								X	X		X						
(Dorado and Hurtado, 2019)	X	X	X		X	X	X				X	X	X	X	X	X	
(Gilson and Englebert, 2011)									X		X						
(Plataniotis *et al.*, 2015)									X		X						
(Harrison; Avgeriou, 2010)											X	X		X	X	X	
(Jansen; Bosch, 2005)		X	X	X	X					X	X						
(Che, 2014)	X				X				X							X	
DRML language	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Source: own elaboration.

Table 1 shows a comparison of the groupings proposed by the language decisions and *rationale* modeling language (DRML), regarding documentation elements proposed by each of the studies. In the first row, we can see the main abstractions that the DRML language groups together by explicit constructs; the second row shows the elements of each proposal, ad in that order, such elements are compared and grouped: *rationale* (R), motivation (M), problem (P), cause (C); trade-off (To), con (Cn), pro (Pr), consequence (Cq); alternative (Alt), architectural design decisions (ADD), decision (D); quality attribute (Qa), business rule (BR), quality goal (QG); tactic (T), architectural pattern (P) and strategy (E). In the proposed DRML language some elements are first-class citizens, that is, there are constructs to model them and the other concepts are included therein, although they are not explicit. However, it has a general space that offers the possibility of describing them, for example, the pros and cons are freely related in a consequence (Cq)-type element.

In the literature review, different types of approaches are identified that are intended to record the *rationale* of architectural design decisions, for example, tools, templates, annotations in the code, models, methodologies, templates, surveys, and capture cycles, many based on meta-models. However, these proposals generate decision models, that is, they focus on generating elements to document the decision, generating distance between the *rationale* and the decision. On the other hand, the proposals can generate an excessive cognitive load upon the designer, because they include too many elements of documentation. Therefore, it is necessary to have a balance sheet since it can be hard to keep updated documentation in an agile context. The rationale's approach to documentation with annotations in the code is of great value, since the code is the most reliable source of information regarding software maintenance. However, not all decisions and their *rationale* can be located in a single place in the code, since architectural decisions and their rationale can impact many parts of the system, so a model that enables to record this knowledge in an explicit and transversal way, would provide a better view of the architectural design of the system, and this, in turn, can be more practical for maintenance, even regarding those extensive architectural and design documents that become unreliable over time (Singer, 1998). So, our approach to code annotations is based on two strategies that can be integrated through the MDE, in such a way that they can provide a practical and simplified strategy, particularly, to support agile development, with a focus on architecture.

The first step would be to extend the code annotations so that the languages are semantically equivalent to facilitate MDE transformations and model to text/code (Model-To-Text or M2T). Furthermore, the language offers the minimum constructs to integrate the decisions with the rationale, this would enable the stakeholders to model an integrative view that is very necessary for architecture modeling. For example, Kruchten, Capilla, and Dueñas (2009) in their approach of 4 + 1 views speak of a fifth view (+1), and these are the scenarios that enable the evaluation of the consistency between the views. This model fulfills the role of making explicit the design decisions that are verifiable in the other views and the *rationale* behind these decisions. Therefore, this work contributes to the area of architecture documentation, particularly to the approach wherein software architecture is treated as a conjunction of architectural design decisions (Singer, 1998). That is, the work of modeling decisions and their rationale is part of the architectural modeling that deals with the representation, capture, management, and documentation of design decisions measured in the software life cycle (Kruchten *et al.*, 2009).

Designing a software architecture is a decision-making process (Lopes; Aquino, 2017). Agile methods drastically changed the way to design software architecture. In projects that use agile methods, for example, Scrum, making architectural design decisions is not the responsibility of a single person, but of the entire development team. Furthermore, the team focuses on the continuous delivery of the product, however, we can see how they make an effort to incorporate architectural practices in their processes (Lopes; Aquino, 2017). Our approach fits appropriately in this context, as it allows us to capture decisions and their *rationale* in an agile way.

3. Decisions and rationale modeling language - DRML

This article presents a documentation language that combines explicit links between architectural design decisions and their justifications. The language seeks to offer a mechanism to document the reasons, alternatives, justifications, consequences, and context behind each architectural design decision. The proposed technique is designed to help stakeholders and architects share, understand, and keep architectural designs. Previously, attempts were made to specify design decisions in the code, however, the code is a single perspective of the software, with many possible others. In a design decision that involves a runtime quality, such as performance, where in the code is it annotated? Considering that performance is affected by the way in which the components are installed, it may be necessary to consider if these components have constant communication, if they are going to be installed on the same machine or server, but, if they are on geographically dispersed servers, there may be noise, and therefore replication must be made, sending information, packages, aspects that slow down the process and that cannot be located in the code. So the code has certain limitations. In agile development, one of the criteria applied in these cases is to use annotations within the code. The problem with software architecture is that not everything can be specified at that level, therefore, a simple, easy-to-load, and adaptable specification that can be added separately as one more architectural artifact and can be associated with new or existing documentation, may be necessary. This is where our language comes in, insofar it is a language that can be used at the conceptual and annotations level, based on the previous studies of Dorado, and Hurtado (2019), which determined that code annotations have a positive impact on the efficiency, effectiveness, and understanding of software architecture and its *rationale*. Therefore, based on this work, the annotation model was complemented with our language. The language is used so that those annotations that are not limited to only possible code annotations can be represented.

The difference between documenting *rationales* and the decision is often unclear. This work is not intended to provide a clear line between them. In fact, this line depends on the domain of the application, the quality requirements of the architect and the will of the system architect. It is common for software architects and stakeholders to follow quality standards to document architecture design. Sometimes, according to the experience of architects, they may or may not explain the reason behind each architectural design decision (Jansen; Bosch, 2005). This is why to document the design reasons behind each decision, we opted only for those that are significant and are architecture-type. Figure 1 presents the meta-model that determines the design decision specification language and its rationale.

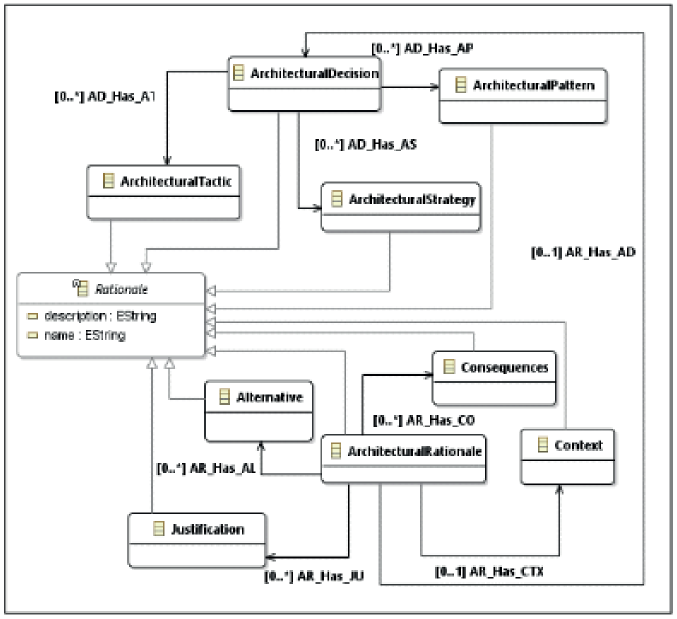

Figure 1. Meta-model of Rationale Architectural
Source: own elaboration.

The meta-model is comprised of the following abstractions: *Architectural Rational* and *Architectural Decision*, which are meant to provide specific links between the reasons and the final design decision. The documentation links enable designers to explicitly structure the reasons behind a solution. In our approach, the first thing that is recorded is the *rationale*, through the key element *Architectural Rationale*, which is related to *Justification*, which is in turn responsible for grouping the justifications. *Alternative* defines a set of alternatives, where the designer must note the alternatives before selecting a particular option. "Consequences" is a description of the expected consequences of a particular solution within the software architecture.

The item should provide additional information on the pros and cons of the solution. For example, a decision can introduce the need to make other decisions, create new requirements or new restrictions in the environment and modify existing requirements, inter alia. *Context*, decision making is often subject to various factors, for example, the experience of the designer, the context that surrounds him, fashion technology, the academic background, inter alia. It's important to write down that knowledge. *Architectural Decision* represents the final design decision selected, which in turn relates to *Architectural Pattern*, in order to provide concrete follow-up links between the final decision and related architectural patterns, where the designer can specify multiple solutions that may be reusable in the future. *Architectural Tactic* is important to document knowledge about tactics, as they are design decisions that influence the response to quality attributes. In the meta-model, it is related directly to *Architectural Decision*. Finally, *Architectural Strategy*, it is vital to note the architectural strategies that are comprised of a set of tactics.

3.1. Technical basis

The proposed approach makes use of model-driven engineering (MDE). To create the domain-specific modeling tool, *Eclipse Modeling Framework* (EMF) and *Graphical Modeling Framework* (GMF) were used, both provided by the Eclipse platform. GMF requires a meta-model and therefore uses EMF, which is a modeling framework that supports and generates XMI and XML documents, which are a standard for the exchange of metadata information with the domain specification.

The first step is to specify in Ecore (the meta-model of EMF) the abstract syntax of the language through a meta-model, then GMF establishes a series of steps to build a tool of this type, taking as input the meta-model and the definition of the graphical metaphor of the language (concrete syntax) is generated semiautomatically from the tool (Montenegro; Gaona; Cueva; San Juan, 2011). From this process, you get a plug-in for Eclipse that contains the built DRML tool. The appearance of the tool is shown in Figure 2.

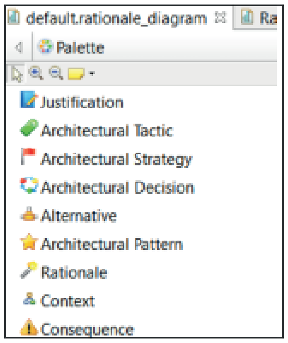

Figure 2. Tool to model the *Architectural Rationale*
Source: own elaboration.

The tool is easy to operate: to create the modules, the operative just has to drag the nodes of the "Pallet" to the work area, fill in the fields and connect them.

3.2. Language characteristics

Language provides several expressive functions that allow it to document the fundamental reason, following the variety of processes and any requirements thereof. Below is an overview of these features:

Lightweight: the language with its domain model can be totally or partially adopted by different types of approaches, proposing a language with the necessary elements that allow documenting the *rationale*, of the main architectural design decisions, but seeking to alleviate the heavy work associated with the activities of software architects and designers when documenting and monitoring.

Simple: the language raises the minimum elements of work for the documentation of the architectural rationale, including and extending the most common process elements found in the scientific literature and the elements most used by software architects and designers.

Expressive: each development and project entity are different, the language considers the main possible scenarios to record a change of architectural design and includes several constructs to assemble a great variety of decisions and their rationale.

3.3. DRML Solution Approach

In this section, the tool for recording the architectural *rationale* is presented, which is implemented based on the proposed language. DRML provides several features for documentation and also provides an editor that supports specific editing for the basis and decision. First, within this editor, architects and designers can create and edit any architectural foundation component together, as shown in Figure 3, during the requirements engineering stage, and the analysis and design, implementation, testing, and deployment stages.

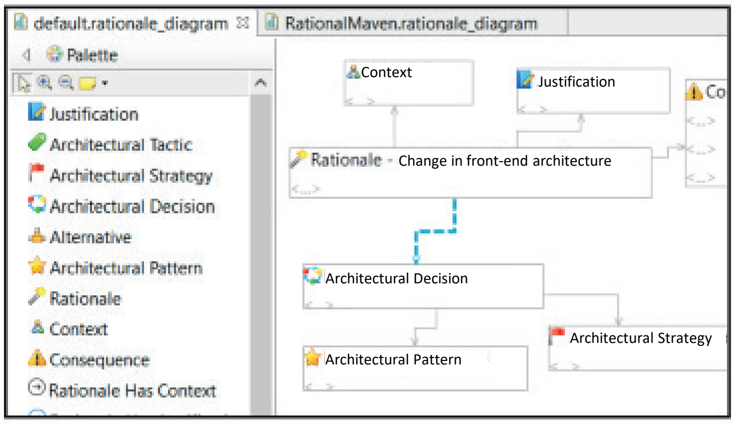

Figure 3. Editor of *Rationale Architectural*
Source: own elaboration.

The language enables the operative, in a simple and elegant way, to create more than one architectural foundation model, that can be implemented for each software component if required, as well as enabling the use of different elements just by selecting and dragging from the tool panel, and this way, the documentation can be built. For example, it is possible to use the *rationale* and *Architectural Decision* element and complement them with the context, justification, architectural strategies, tactics and the parts that are considered important and that impact the designs of the software architecture. As shown in Figure 4, it is also important to highlight that each of the pieces that comprise the tool is optional when building the knowledge of the decision.

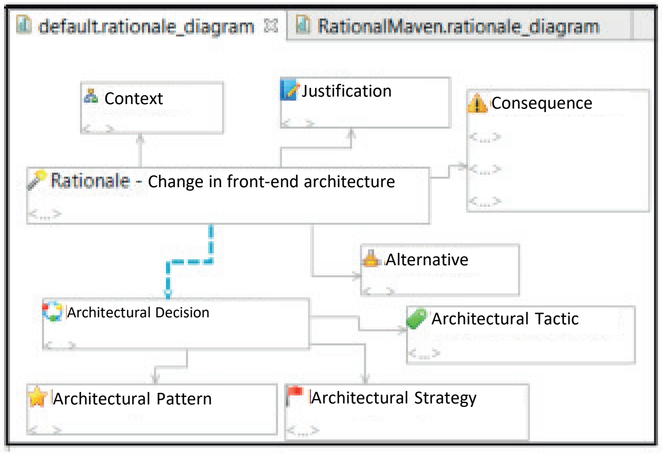

Figure 4. Architectural Rationale Elements
Source: own elaboration.

The DRML-generated design can be added separately as one more architectural elements associated with new and existing documentation. This is shown in Figure 5.

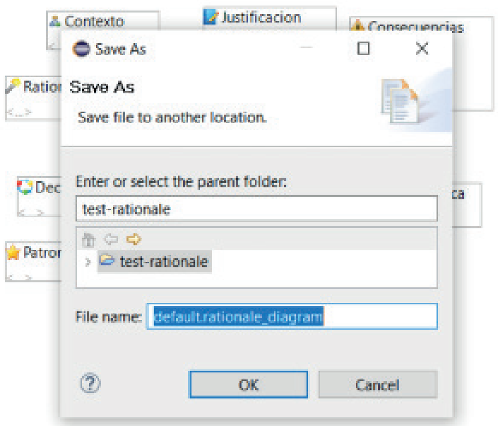

Figure 5. Artifact of Architecture Software
Source: own elaboration.

4. Methodology

To achieve the objectives proposed above, the research guidelines of the Bunge method (2002) were used for the specification of the tracking language that facilitates the documentation of architectures. The evaluation of the process followed the case study method proposed by Runeson and Höst (2009). The case study is a research methodology suitable for software engineering, which enables the study of the engineering in its real context, with the purpose of keeping the integrity and the significant characteristics of the events. It is executed when the researcher has little control over the events and when the subjects of study are easier to observe in group, rather than separately (Runeson; Host, 2009). The stages of language development are described below.

Exploratory stage: consists of the approach to the problem, construction of the theoretical model, recognition of the facts, study of the theoretical references of guidelines and techniques to document the rationale, of the main architectural design decisions in an agile context with software architecture practices.

Formulation stage: the mechanism that allows documenting the main architectural design decisions is defined through a visual language of architectural rationale.

Evaluation stage: adjustments to the process are made following the findings obtained in the case study and the evaluations carried out.

In the following subsections, examples of design decisions of the SUIIN (Single Indigenous Information System) project are used to explain and illustrate the proposed tool.

4.1. Case context

The study was performed in a public health-promoting entity, which seeks to strengthen the administrative and organizational capacity of its internal processes and procedures, through the automated administration of information, to provide a better health service to its users and optimize internal processes. Therefore, within its organizational processes, a software development sub-process was implemented. The team in charge was comprised of systems engineers, with a high degree of technical knowledge, focused on the formalization, optimization, and systematization of processes, through the SUIIN system that centralizes information. It should also be mentioned that they implement good practices in the development of information systems. For example, the methodology they use to develop software is the unified agile process (AUP), which is used to develop the software in an agile way, based on good architectural practices. However, the organization's legacy systems, which were not built with these same practices, require modifications (including new design decisions) that require the understanding and analyzing previously made decisions. Therefore, the company has a suitable scenario to apply DRML and its support tool. The source code of the system is in the company's repository, in order to describe the software as accurately as possible. The structure is based on the "4 +1" architecture view model.

4.2. Indicator and metric

Table 2 presents the metric used to evaluate the value that the proposal has on the documentation of the architectural rationale. Subsequently, its measurement and management are described.

Table 2.
Indicator and metric

Indicator	Metric	Instrument
User perception of usability	Ease of use of the elements of the prototype implemented through the ¬DRML model.	Survey: https://github.com/ tomil-ton/PercepcionUsabili dad.git

Source: own elaboration.

The indicator and how it is calculated through the identified metric are described in detail below.

4.2.1. User perception of usability

To assess the satisfaction perceived by users, a seven-question questionnaire was applied based on the standardized questionnaire of Sauro and Lewis (2016). The items in the form generate four scores, one overall, and three subscales. The rules for computing are:

- Overall: average of the answers for items 1 to 16 (all items).
- Tool quality: average elements from 1 to 6.
- Information quality: average elements from 7 to 12.
- Interface quality: average elements from 13 to 16.

Equation 1 is defined to calculate the perceived usability as follows:

$$Pu = \frac{\sum_{i=1}^{n} RA_i}{n}$$

Where Pu is the perception of usability; RAi is the assimilated result for question i, which can take values from 1 to 7; and n is the number of items evaluated in the survey by the engineers.

4.3. Architectural Changes

Currently, the system on the front-end is using the *Prime Faces* presentation *framework*, which has responded optimally to the needs of the organization, however, to improve user accessibility to the system, the need for the new modules to be adaptable to mobile devices has been identified, besides improving its visual appearance. On the other hand, the *back-end* of the modules that are built for the system lack a mechanism that allows them to interact with other modules, this hinders code reuse. Furthermore, it is necessary to redefine some layers to achieve better modifiability and scalability.

4.4. Participants

The participants were software engineers (systems and electronics) currently linked to the local software industry, so the selected people have certain knowledge in the fields of computer science and similar fields. Three participants were women and three were men over 27 years old, with experience between 2 and 5 years in the industry, playing different roles, such as analyst, product engineer, developer, testing, and software architect.

4.5. Case development

The study begins with an introduction to architectural *rationale* theory to show the concepts to participants, followed by the presentation of the DRML tool. An example was made to understand its use, structures, constraints, and the resulting modeling of the *rationale*. After training the participants, the architectural changes to be made were explained and the study material was delivered. The project was configured in the Eclipse development environment. The application source code was downloaded from the company's Subversion Energy Control (SVN) repository, such as the SAD (Software Architecture Document) architecture document - SUIIN. A guide was also given with the architectural changes and a survey that must be carried out after completing the architectural changes. Participants began to document the rationale of the changes, each participant reviewed the previous documentation of a documented change, with the aim of evaluating whether the *rationale* of each decision is understandable by another participant. As each participant modeled the decision with their own rationale and evaluated the documentation of the other changes, the survey was distributed to collect quantitative and qualitative information that contributed to the definition of the final structure of the documentation model for the architectural *rationale*.

5. Results

Below are the quantitative and qualitative results of the case study.

5.1. Quantitative results

Table 3 shows the results of the usability perception survey for each of the engineers in the evaluated criteria: tool quality, information quality, and interface quality.

Table 3.
Usability perception

		Analysis Unit		
Criteria	Engineer 1	Engineer 2	Engineer 3	Average
Tool quality	6.0	6.6	6.1	6.2
Information quality	5.1	6.6	5.0	5.6
Interface quality	6.0	6.0	3.2	5.0
Overall average	5.7	6.4	4.8	5.6

Source: own elaboration.

The results in Table 1 show that the quality of the tool is approximately 89 %. The quality of the information is approximately 80 % and the quality of the interface is approximately 72 %.

5.2. Qualitative results

Some of the engineers' observations, rationale modeling, and design decision are presented below:

a) Remarks:

The definition of architecture and its logic are the fundamental axis for the evolution of software, it would be interesting to have a repository where documented knowledge could be reviewed. It would be very useful if the tool could be uploaded to a repository and installed as a plug-in of the Eclipse IDE. It would be useful if specified structural design patterns and mini-architecture patterns could be represented. It would be useful if design patterns that have been customized could be represented. The tool needs to allow links to external documents or files, for example, the Unified Modeling Language (UML). It is important that the tool can identify the stakeholder that generates the requirement and the different team members involved in the process, each with their respective roles. When documenting architectural tactics and strategies, it is helpful to be able to note that a strategy consists of a set of tactics and relate it to the decision.

b) Rationale and decision modeling:

Figure 6 shows the final modeling of the rationale of one of the engineers who participated in the study.

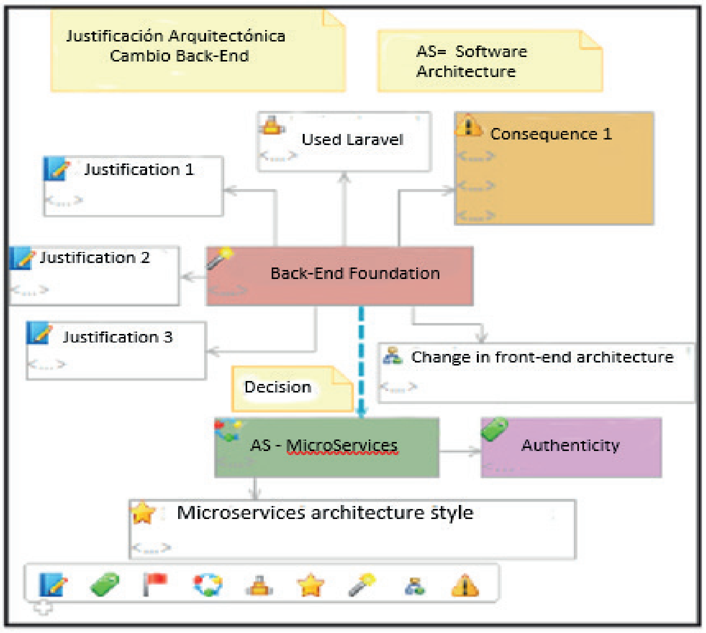

Figure 6. Model resulting from *Architectonic Rationale*
Source: own elaboration.

As shown in the graphic, emphasis was made on supporting the final design decision, and recording the architectural changes correctly, using the elements of the tool, that is, the graphic clearly shows how the *rationale* relates to three justifications. It also notes a possible alternative within a context and a consequence with its pros and cons. It also documents the final decision along with explicit links to a specific architecture pattern, which relates to an architecture tactic.

The solution proposed by the foregoing engineer consists of changing the monolithic architecture for an architecture based on microservices, using the Java language and Maven (software tool for the management and construction of Java projects). Three justifications at the technical level need to be noted to support the final decision. First, the proposal is to create modules with dependency management based on Maven, which guarantees the structuring of the code in separate projects, which enables it to separate the logical layer, the data access layer, and the service layer. Second, through Maven it is possible to design a common archetype, to place reusable logic, for example, access to common methods and classes that are required from different microservices, then he mentions that Maven enables him to create a base archetype for each module. He also noted a possible consequence in microservices that enables modules to be created in different technologies, which causes maintenance to be complex. One advantage is that it is not necessary to stop the application to make adjustments, each component works separately. Furthermore, it records a tactic that consists of authenticating clients through JWT (JSON Web Tokens) security tokens, generating a unique token, and using an authentication service that guarantees the validity of the connection. Finally, he notes a possible technical alternative that consists of using Laravel, a PHP framework.

6. Analysis

Different architectural styles motivated by different design decisions were used in the project. An architectural product was generated by each component during the iterations performed. Different degrees of the details of the decisions taken were recorded, for further manipulation or maintenance process. Both in the first architectural change and in the second there were different opinions. The evaluation of the tool in both changes demonstrated the usefulness of visually representing and managing the *Architectural Rationale* of design decisions. Up to this point, the interface of the tool was enough to easily navigate the rationale and decisions made. Therefore, the architect has a complete view of the overall design process, because the elements shown by the tool chronologically describe the most important events that occurred during the project. This approach facilitates the understanding of the architecture construction process for stakeholders who are directly related to technical problems. Furthermore, useful documentation is generated that includes the decisions made, the patterns applied, and the architectural products generated during the process. The tool provides the model with the *Rationale Diagram* extension that is associated with the new or existing documentation of the software architecture as another view, in an agile context. Although software engineers found the language and tool useful, there are still some doubts about it. First, the quality of the interface was evaluated with only 72 %. Particularly there are limitations to scaling the decision model to the entire project, traceability with requirements, in particular non-functional, as well as its traceability to the architectural model expressed in other models, such as UML.

It is important to consider that the case unfolds in the context of a small organization through several modeling sessions, with a small set of design decisions (less than three). However, the experience reported by Zimmermann (2012), has identified 35 recurring decisions in the development of business applications in the context of service-oriented design and architecture. In addition, Kellari, Crawley, and Cameron (2018) analyzed the architectural decisions of aircraft software from dc3 to 787, finding a decrease in the variety of decisions in a dataset, of 157 architectures in aircraft history, thereby concluding that there are 27 architecturally relevant decisions. These two scenarios; a typical scenario and an extreme scenario in terms of reliability, show that the language of decision-making and its rationale should facilitate the formation of the modeling of a not small number of architectural design decisions, facilitating an organizational structure of said decisions (temporal relations or cause/effect), and a relation with external elements to facilitate its integration and traceability. One approach is to visualize architectural decisions from different perspectives. One perspective would address a set of concerns pertaining to the decision. The idea of showing different decision-making perspectives would facilitate direct association with the points of view of those who describe the architecture, although this would result in more complex relations. Furthermore, there is also no relation between design decisions and architectural design artifacts where these decisions are reflected (views and models of architecture).

On the other hand, effective decision-making requires effective communication and coordinated team actions, which adds greater complexity to open source projects, because the developers of these projects generally do not work in the same place (Harrison; Gubler; Skinner, 2016). This increases the possibility of erroneous architectural decisions and the motivation behind them. Moreover, it is likely that there are different objectives, because they do not necessarily work in the same company and for the same interests, so decision-making in a unified way is of vital importance. In this scenario, the DRML tool would provide a mechanism to eliminate the ambiguity of decisions and the possibility of their unification. Besides the issue of scalability, it has been necessary to explore the possibility of providing computer-aided collaborative work capabilities. And this would not only be beneficial in open-source projects, but also in general, because architectural design requires teamwork that often fails for several reasons, such as lack of flexibility, staff ego, and loyalty to a preferred technology, thus preventing consensus in decisions and their rationale. Some team members constantly try to impose their way of doing things onto others, rather than objectively participating in discussions (Dasanayake; Markkula; Aaramaa; Oivo, 2016).

7. Conclusions

This article highlights the importance of documenting the architectural rationale of design decisions as a key piece in the documentation of software architecture, in an agile context. Recording and documenting knowledge of architectural design decisions is important for development and maintenance processes. To achieve the objective of documenting the justification of architectural changes, a language and a tool based on its specified elements have been proposed capable of recording, maintaining and managing the decisions made during the process of construction, and maintenance of the architecture.

Our approach connects requirements with architectures through architectural *rationale* and design decisions, to establish traces between them. The study indicates that working with MDE allows defining and performing transformations between different models, therefore facilitating interoperability between platforms. As a result of the evaluation carried out, it is clear that these kinds of tools and, in particular, the tools based on meta-models, innovate software engineering with a focus on application development from domain-specific models in an agile way while reducing costs, benefiting the software industry and the scientific community. Limitations of the proposal were also identified. It is necessary that the language enables the operative to specify functional and non-functional requirements and scale the modeling for a wide number of decisions and their relations. As for DRML, it is important to have links to external sources of knowledge, for example, UML, architecture documents, and Wikis, inter alia. It would also be useful to generate a navigability tree where documented knowledge can be searched. Since the study is an exploratory case study, it also has limitations in the sample of the data, insofar it depends, to a large extent, on the experience and technical skills of the participants with knowledge of the subject to obtain better results. In future work, the language is proposed to support the modeling of decision packages (packages of concerns and perspectives), traceability with other artifacts of the architecture, as well as incorporating aspects of versioning and collaborative modeling in the tool. Furthermore, leveraging the MDE paradigm, it is necessary to relate the documentation elements of the DRML language with the code annotations developed by Dorado and Hurtado (2019), given that the code continues to be the most frequent and reliable source, wherein developers find information relevant to the justifications of design decisions, particularly, in the agile approach with a focus on architecture.

References

Aldrich, J.; Chambers, C.; y Notkin, D. (2002). ArchJava: connecting software architecture to implementation. *Proceedings of the 24th International Conference on Software Engineering*. ICSE 2002, (pp. 187–197). https://doi.org/10.1109/ICSE.2002.1007967

Bass, Len; Clements, Paul; Kazman, Rick (2003). *Software architectuSoftware architecture in practicere in practice*. Boston, EE.UU: Addison-Wesley.

Bucaioni, Alessio; Cicchetti, Antonio; Ciccozzi, Federico; Mubeen, Saad; Sjodin, Mikael (2017). A Metamodel for the Rubus Component Model: Extensions for Timing and Model Transformation from EAST-ADL. *IEEE Access*, 5, 9005–9020. https://doi.org/10.1109/ACCESS.2016.2641218

Bunge, Mario (2002). *La investigación científica: su estrategia y su filosofía*. México: Siglo XXI Editores.

Che, Meiru (2014). *Managing Architectural Design Decision Documentation and Evolution Committee* (tesis doctoral). University of Texas at Austin, Austin, Texas.

Cleland-Huang, Jane; Mirakhorli, Mehdi; Czauderna, Adam; Wieloch, Mateusz (Mayo de 2013). Decision-Centric Traceability of architectural concerns. *2013 7th International Workshop on Traceability in Emerging Forms of Software Engineering, (TEFSE)*. San Francisco, California, EE. UU. https://doi.org/10.1109/TEFSE.2013.6620147

Dasanayake, Sandun; Markkula, Jouni; Aaramaa, Sanja; Oivo, Markku (2016). An Empirical Study on Collaborative Architecture Decision Making in Software Teams. In: Tekinerdogan B.; Zdun U.; Babar A. (eds). *European Conference on Software Architecture* (pp. 238-246). Springer, Cham: Lecture Notes in Computer Science. https://doi.org/10.1007/978-3-319-48992-6_18

Dermeval, Diego; Castro, Jaelson; Silva, Carla; Pimentel, Joao; Bittencourt, Ibert; Brito, Patrick; Elias, Endhe; Tenorio, Thyago; Pedro, Alan (March de 2013). On the use of metamodeling for relating requirements and architectural design decisions. *SAC '13: Proceedings of the 28th Annual ACM Symposium on Applied Computing* (pp. 1278–1283). Coimbra, Portugal.
https://doi.org/10.1145/2480362.2480601

Dorado, Santiago; Hurtado, Julio (2019). Documenting architectural *rationale* using source code annotations: An exploratory study. *EPiC Seriesin Computing,* 64, 204-214.

Gilson, Fabian; Englebert, Vincent (September de 2011). *Rationale*, decisions and alternatives traceability for architecture design. *ECSA 11: Proceedings of the 5th European Conference on Software Architecture: Companion* (pp. 1-9). New York, EEUU.
https://doi.org/10.1145/2031759.2031764

Hadar, Irit; Sherman, Sofia; Hadar, Ethan; Harrison, Jhon (May 2013). Less is more: Architecture documentation for agile development. *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2013 - Proceedings* (pp. 121–124). San Francisco, CA, USA.
https://doi.org/10.1109/CHASE.2013.6614746

Harrison, Neil; Avgeriou, Paris (2010). How do architecture patterns and tactics interact? A model and annotation. *Journal of Systems and Software,* 83(10), 1735–1758.
https://doi.org/10.1016/j.jss.2010.04.067

Harrison, Neil; Gubler, Erich; Skinner, Danielle (March de 2016). Architectural Decision-Making in Open-Source Systems-Preliminary Observations. *2016 1st International Workshop on Decision Making in Software ARCHitecture - Proceedings* (pp. 16–21). Venice, Italy.
https://doi.org/https://doi.org/10.1109/MARCH.2016.7

Hernández, Flor; Hurtado, Julio (2016). Difficulties and challenges in the incorporation of architectural practices. *Sistemas y Telemática*, 14(38), 74–86.
https://doi.org/10.18046/syt.v14i38.2290

Hesse, Tom-Michael; Kuehlwein, Arthur; Paech, Barbara; Roehm, Tobias; Bruegge, Bernd (2015). Documenting Implementation Decisions with Code Annotations. *SEKE*, 152-157.
https://doi.org/10.18293/SEKE2015-084

Jansen, Anton; Avgeriou, Paris; van Der Ven, Jan (2009). Enriching software architecture documentation. *Journal of Systems and Software*, 82(8), 1232–1248.
https://doi.org/10.1016/j.jss.2009.04.052

Jansen, A.; Bosch, J. (November 2005). Software architecture as a set of architectural design decisions. *5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005 – Proceedings* (pp. 109–120). Pittsburgh, Pennsylvania.
https://doi.org/10.1109/WICSA.2005.61

Kellari, Demetrios; Crawley, Edward; Cameron, Bruce (2018). Architectural decisions in commercial aircraft from the DC-3 to the 787. *Journal of Aircraft,* 55(2), 792–804.
https://doi.org/https://doi.org/10.2514/1.C034130

Kruchten, Philippe; Capilla, Rafael; Dueñas, Juan (2009). The decision view's role in software architecture practice. *IEEE Software*, 26(2), 36–42.
https://doi.org/10.1109/MS.2009.52

Kruchten, Philippe; Obbink, H.; Stafford, J. (2006, March). The past, present, and future for software architecture. IEEE Software, 23(2), 22-30.
https://doi.org/10.1109/MS.2006.59

Lopes, Socrates; Aquino, Plinio (2017). Architectural Design Group Decision-Making in Agile Projects. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*(pp. 201-2015). Gothenburg, Sweden.
https://ieeexplore.ieee.org/abstract/document/7958489/

Malavolta, Ivano; Muccini, Henry; Rekha, Smrithi (2014). Enhancing Architecture Design Decisions Evolution with Group Decision Making Principles. In: Majzik I., Vieira M. (eds). *Software Engineering for Resilient Systems. SERENE 2014. Lecture Notes in Computer Science* (pp. 9–23). Budapest, Hungary: Springer. https://doi.org/10.1007/978-3-319-12241-0_2

Manteuffel, Christian; Tofan, Dan; Koziolek, Heiko; Goldschmidt, Thomas; Avgeriou, Paris (2014). Industrial implementation of a documentation framework for architectural decisions. *2014 IEEE/IFIP Conference on Software Architecture – Proceedings* (pp. 225–234). Sydney, NSW.
https://doi.org/https://doi.org/10.1109/WICSA.2014.32

Montenegro, Carlos; Gaona, Paulo; Cueva, Juan; San Juan, Oscar (2011). Aplicación de ingeniería dirigida por modelos (MDA), para la construcción de una herramienta de modelado de dominio específico (DSM) y la creación de módulos en sistemas de gestión de aprendizaje (LMS) independientes de la plataforma. *Dyna*, 78 (169), 43–52.
https://www.redalyc.org/pdf/496/49622390005.pdf

Plataniotis, Georgios; Ma, Qin; Proper, Erik; de Kinderen, Sybren (2015). Traceability and modeling of requirements in enterprise architecture from a design *rationale* perspective. *2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS)* (pp. 518–519). Athens, Greece.
https://doi.org/10.1109/RCIS.2015.7128916

Reynoso, Carlos (2004). *Introducción a la Arquitectura de Software.*
http://cic.puj.edu.co/wiki/lib/exe/fetch.php?media=materias:introarq.pdf

Roeller, Ronny; Lago, Patricia; van Vliet, Hans (2006). Recovering architectural assumptions. *Journal of Systems and Software*, 79(4), 552–573.
https://doi.org/10.1016/j.jss.2005.10.017

Roldán, Maria; Gonnet, Silvio; Leone, Horacio (2016). Operation-based approach for documenting software architecture knowledge. *Expert Systems*, 33(4), 313–348.
https://doi.org/10.1111/exsy.12152

Runeson, Per; Höst, Martin (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2), 131–164.
https://doi.org/10.1007/s10664-008-9102-8

Sauro, Jeff; Lewis, James (2016). Standardized usability questionnaires. In *Quantifying the User Experience* (pp. 185–248). EE.UU: Elsevier.
https://doi.org/10.1016/b978-0-12-802308-2.00008-4

Singer, J. (1998). Practices of software maintenance. *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)* (pp. 139–145). Bethesda, EE.UU.
https://doi.org/10.1109/ICSM.1998.738502

Tang, Antony; Babar, MMuhammad; Gorton, Ian; Han, Jun (2006). A survey of architecture design *rationale*. *Journal of Systems and Software*, 79(12), 1792–1804. https://doi.org/10.1016/j.jss.2006.04.029

van Der Ven, Jan; Bosch, Jan (2016). Busting Software Architecture Beliefs: A Survey on Success Factors in Architecture Decision Making. *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (pp. 42–49). Limassol, Cyprus. https://doi.org/10.1109/SEAA.2016.35

van Der Ven, Jan; Jansen, Anton; Nijhuis, Jos; Bosch, Jan (2006). Design decisions: The bridge between *rationale* and architecture. In Dutoit A.H., McCall R., Mistrík I., Paech B. (eds). *Rationale Management in Software Engineering* (pp. 329–348). Berlin, Heidelberg, Springer. https://doi.org/https://doi.org/10.1007/978-3-540-30998-7_16

van Vliet, Hans; Tang, Antony (2016). Decision making in software architecture. *Journal of Systems and Software*, 117, 638–644. https://doi.org/10.1016/j.jss.2016.01.017

Zimmermann, Olaf (2012). Architectural decision identification in architectural patterns. In: *WICSA/ECSA '12: Proceedings of the WICSA/ECSA 2012 Companion Volume* (pp. 96–103). New York, EE.UU: Association for Computing Machinery. https://doi.org/10.1145/2361999.2362021