A Decision Support System for Pattern-Driven Software Architecture

Siamak Farshidi$^{1\,[0000-0003-3270-4398]}$ and Slinger Jansen$^{1,2\,[0000-0003-3752-2868]}$

1 Department of Information and Computer Science, Utrecht University, The Netherlands
2 Visiting Scientist, School of Engineering Science, LUT University, Finland
{s.farshidi, slinger.jansen}@uu.nl

Abstract. The selection process of architectural patterns is challenging for software architects, as knowledge about patterns is scattered among a wide range of literature. Knowledge about architectural patterns must be collected, organized, stored, and quickly retrieved when it needs to be employed. In this tool paper, we introduce a decision support system that uses a decision model for supporting software architects with the pattern selection problem according to their requirements, including functional and quality requirements. The decision model is built based on a technology selection framework for modeling multi-criteria decision-making problems in software production. Twenty-four software architects in the Netherlands have evaluated the tool. They confirm that the tool supports them with their daily decision-making process.

Keywords: *Architectural patterns; Pattern-driven software architecture; multi-criteria decision-making; decision support system; decision model;*

1 Introduction

Software architecture is fundamental for the development of a software product and plays an indispensable role in its success or failure as software architecture deals with the base structure, subsystems, and interactions among these subsystems [4]. Software architecture design can be viewed as a decision-making process: software engineers consider a set of alternative solutions that could solve a system design problem, and select the set that is evaluated as the optimal [14].
Software architecture is the composition of a set of architectural design decisions, concerns, variation points, features, and usage scenarios that address various system requirements, including functional and quality requirements [2]. Each architectural design decision is made with a design rationale [6], which represents the knowledge that provides the answers to questions about the design decision or the process followed to make that decision.
An architectural pattern describes high-level structures and behaviors of software systems and addresses a particular recurring problem within a given context in software architecture design [3]. Architectural patterns aim to satisfy several requirements and help to document the architectural design decisions [1]. So that selecting architectural patterns is a subset of architectural design decisions [22], and it is a challenging task for software architects, as knowledge about patterns, such as their application domains

and their interactions with quality attributes, is scattered among a wide range of literature [18]. Thus, a decision support system (DSS) is needed to support software architects with architectural pattern selection intelligently.

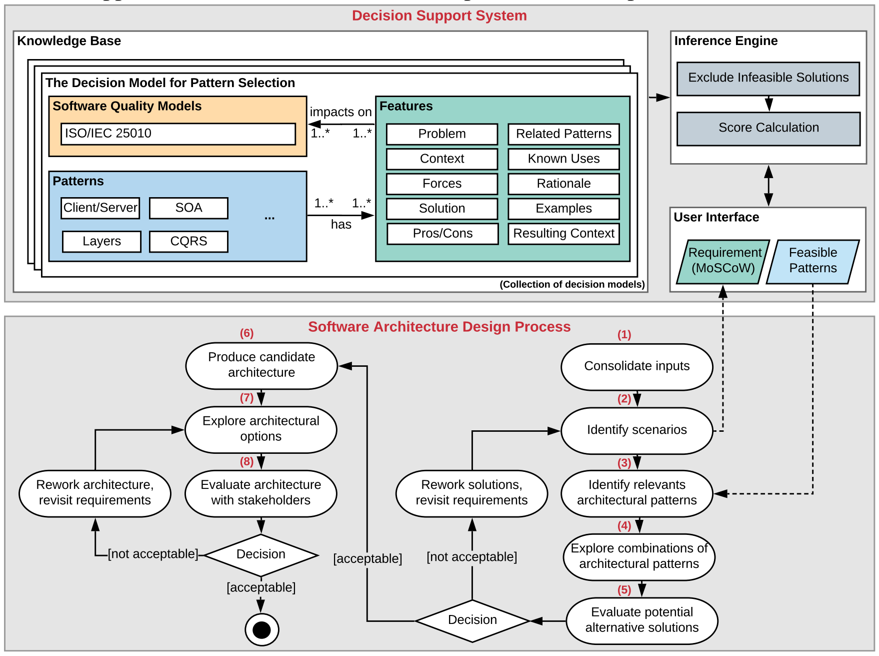
In this article, we present a DSS for Pattern-Driven Architecture, which assists software architects in selecting the best fitting set of patterns. The DSS asks architects for their requirements in terms of functional requirements and quality concerns. Accordingly, several sets of architectural patterns are returned that match these requirements. Subsequently, architects can start tweaking the requirements to find the most suitable set of patterns for their design. The DSS is based on several well-known software engineering concepts, such as the ISO/IEC software quality models and the MoSCoW prioritization technique. Architects will indicate their preferences using primary selections such as 'The application *must have* high availability' and 'The application *could have* accessibility'. Using a literature study, we have assessed how patterns perform on these quality criteria. The DSS bundles this knowledge and provides architects with an interactive and collaborative decision tool.

We regard building a software architecture as a decision-making process [17]: (1) Stakeholders with their requirements are engaged. (2) Scenarios are captured. (3) Architectural patterns are identified to address requirements. (4) Potential combinations of patterns are explored. (5) Architects evaluate the combinations of patterns (alternative solutions). If the alternative solutions do not meed the requirements, they are reworked and requirements revisited. (6) An architecture is drafted using the identified patterns (alternative solutions), viewpoints, and perspectives. (7) Different architectural alternatives for refining the draft are explored, and architectural decisions are made to select among them. (8) The architecture is evaluated with stakeholders. Finally, if the architecture does not fulfill stakeholder requirements, the architecture design is reworked and requirements possibly revisited (see Fig. 1). While this process has been a reliable method for producing architectures, it strongly depends on the architect's limited knowledge and experience, who may have experience with only a small number of patterns. Thus, we envision a process where the architect is supported by tools to enhance her knowledge of the patterns available for particular design problems.

Recently, we designed a framework [8] for supporting software developers and architects (decision-makers) with their multi-criteria decision-making (MCDM) problems in software production. An MCDM problem deals with evaluating a set of alternatives and considers a set of decision criteria [15]. In this tool paper, we introduce a decision model, based on the framework, for the patterns selection problem. The DSS employed the decision model to support software architects with the pattern selection problem. Accordingly, we believe that the DSS can be used in steps (1-5) to facilitate the decision-making process for software architects (see Fig. 1).

The rest of this tool paper is organized as follows. Section 2 outlines a brief description of the DSS components and explains the constituent parts of the decision model. Section 3 presents the DSS and its application through a real-world example. Section 4 positions the DSS, among other tools and MCDM approaches, in the literature. Finally, Section 6 presents the evaluation of the DSS and summarizes this tool paper.

Fig. 1. This figure shows that the DSS can be deployed in the the software architecture design process to support the software architects with the pattern selection problem [7,17].


Decision Support System
Knowledge Base
The Decision Model for Pattern Selection
Software Quality Models
ISO/IEC 25010
impacts on
1..*
1..*
Features
Problem
Context
Forces
Solution
Pros/Cons
Related Patterns
Known Uses
Rationale
Examples
Resulting Context
Patterns
Client/Server
SOA
Layers
CQRS
...
1..*
1..*
has
(Collection of decision models)
Inference Engine
Exclude Infeasible Solutions
Score Calculation
User Interface
Requirement (MoSCoW)
Feasible Patterns
Software Architecture Design Process
(1) Consolidate inputs
(2) Identify scenarios
(3) Identify relevants architectural patterns
(4) Explore combinations of architectural patterns
(5) Evaluate potential alternative solutions
Decision
[not acceptable]
Rework solutions, revisit requirements
[acceptable]
(6) Produce candidate architecture
(7) Explore architectural options
(8) Evaluate architecture with stakeholders
Decision
[not acceptable]
Rework architecture, revisit requirements
[acceptable]

2 Decision Support System

A DSS is an information system that comprises domain-specific knowledge and decision models to assist decision-makers by offering knowledge about a set of alternatives [20]. In this tool paper, the DSS integrates key aspects of knowledge-driven and model-driven DSSs [16] to store and organize the extracted knowledge regarding architectural patterns systematically facilitate the decision-making process. Note, for the sake of simplicity, we use *patterns* to refer "architectural patterns".
Additionally, we follow the framework [8] for modeling decision problems in software production as MCDM problems. The framework applies the six-step decision-making process [15] to build decision models for MCDM problems. The knowledge base of the DSS is a collection of decision models for different MCDM problems [7,10,11,9]. According to the framework, the decision model for the pattern selection problem contains three sets (including Patterns, Software Quality Models, and Features) besides the mapping among the elements of these sets (see Fig. 1).
- Patterns: Patterns are the building blocks that, when assembled, can provide complete solutions for an architect's problem (see Fig. 1). Patterns have relationships to each other. For example, patterns can be alternatives to each other, for example, *Interpreter*, *Rule-Based System*, and *Virtual Machine* [1]; Moreover, some patterns can also be complementary and combined. For instance, the *Client-Server* pattern can be combined with the *Broker* pattern [12].
]

- Software Quality Models: A set of quality attributes, such as Availability and Security, should be defined in the decision model. We employed the *ISO/IEC 25010* standard [13] as a domain-independent quality model. The key rationale behind using this software quality model is that it is a standardized way of assessing a software product's quality. Moreover, it describes how easily and reliably a software product can be used.

- Features: Each pattern has a set of features, for instance, "centralized governance" is a feature of the "Client-Server". We identified the following types of features through a Systematic Literature Review (SLR) [11]. We reviewed *21,373* articles, and finally, *232* high-quality primary studies have been selected for performing the knowledge extraction process. Note, such feature types can be found in most patterns, even with different titles. (1: Problem) Descriptions of the problems indicating the intent in applying patterns. (2: Context) The preconditions under which patterns are applicable. (3: Forces) Descriptions of the allied forces and constraints. (4: Solution) Static structures and dynamic behaviors of patterns. (5: Resulting Context) The post-conditions after a pattern has been applied. (6: Examples) Some sample applications of patterns. (7: Rationale) An explanation/justification of each pattern as a whole. (8: Related Patterns) The relationships among patterns. (9: Known Uses) Known applications of patterns within existing systems. (10: Pros/Cons) Pros and cons of employing patterns.

- Mappings: We identified the impacts of 29 patterns on 40 quality attributes based on a series of expert interviews with twelve senior software architects at different software producing organizations in the Netherlands [11]. Moreover, The mapping between the patterns and the features was investigated with the SLR and the experts.

Decision-Makers, such Software architects and developers, prioritize their requirements based on the MoSCoW prioritization technique [5], and then they send the requirements through the user interface of the DSS to the inference engine. Figure 3 shows the user interface of the DSS.

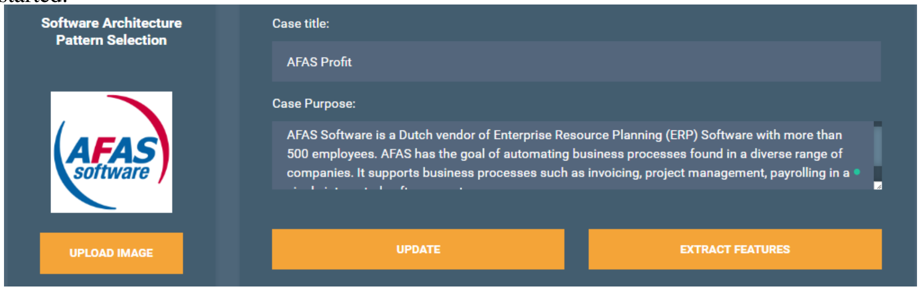
Inference engine: The DSS has an inference engine that receives inputs from the user interface. Next, it excludes all infeasible solutions, those that do not support "Must-Have" features or those that support "Won't-Have" features, and then it ranks the feasible solutions based on the number of "Should-Have" and "Could-Have" features that they support. In other words, requirements with *Must-Have* or *Won't-Have* priorities act as hard-constraints and requirements with *Should-Have* and *Could-Have* priorities act as soft-constraints. The inference engine assigns a non-negative score to each alternative solution based on the well-known Sum of Weights Method [7], and finally, it returns a shortlist of feasible patterns (solutions) to the user interface.

3 A Practical Running Example

The DSS is accessible through the following link: (`https://dss-mcdm.com`). After login to the system, a software architect should select the "Software Architecture Pattern Selection" to create an instance of the decision model.

This section presents a real-world example of the pattern selection process. We asked a software architect at AFAS Software, a software producing organization in the Netherlands, to define their software architecture from a high-level of abstraction; then, we

Fig. 2. The architects describe their case in the context description screen. The tool uses text matching to automatically extract a subset of features from the description to get the architect started.



Case title:

AFAS Profit

Case Purpose:

AFAS Software is a Dutch vendor of Enterprise Resource Planning (ERP) Software with more than 500 employees. AFAS has the goal of automating business processes found in a diverse range of companies. It supports business processes such as invoicing, project management, payrolling in a

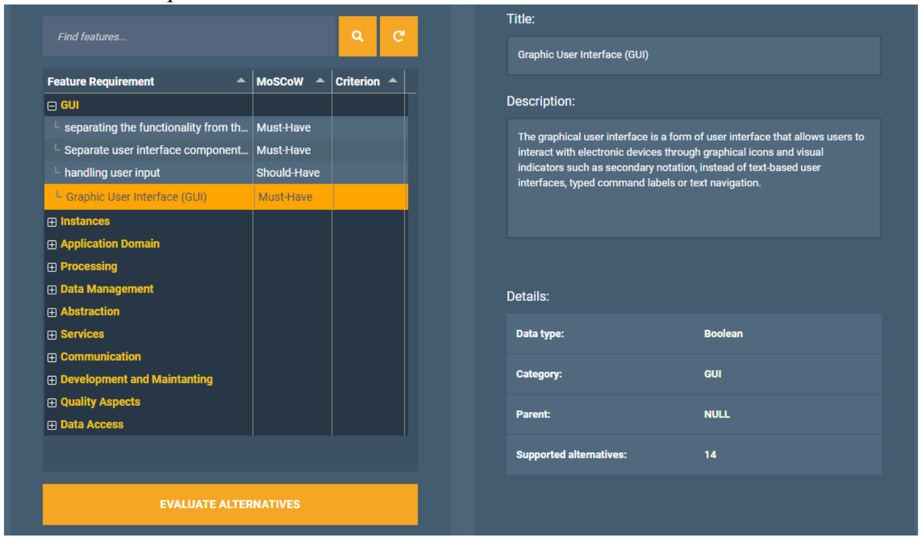
used the DSS and the decision model to capture the architect's concerns and requirements; next, the DSS generated a set of solutions accordingly.

Case Description - The software architect described AFAS software as follows: *AFAS Software is a Dutch vendor of Enterprise Resource Planning (ERP) Software with more than 500 employees. AFAS has the goal of automating business processes found in a diverse range of companies. It supports business processes such as invoicing, project management, payrolling in a single integrated software system. The current AFAS product, called AFAS Profit, is a traditional client-server application with a relational database for storing and retrieving customers' management data, such as business models and ontologies. AFAS Profit is a complete, integrated ERP system used by more than 10000 small and medium-sized enterprises. For example, Ernst & Young, Kwik-Fit, LeasePlan, Oad Reizen, Sandd, and Wibra, are already employing AFAS Profit to automate their business processes. Fig. 2 shows the description of the decision-making problem in terms of the case title and description; moreover, the logo of the company can be attached to the "case description".*

Case Definition - The software architect defined AFAS Profit as *a web-based solution that is consistent with the user experience of the windows client but feels web-native to customers. AFAS Profit is configurable by customers in their styling to match their logo and business style.* AFAS Profit has the following characteristics: (1) It is a combination of a client or frontend portion that interacts with the user and a server or back-end portion that interacts with the shared resource. The client process contains solution-specific logic and provides the interface between the user and the rest of the application system. The server process acts as a software engine that manages shared resources. (2) All data are centralized on a single server, simplifying security checks, including updates of data and software. (3) It supports a higher degree of flexibility and security, compared to the previous solution. (5) Its performance has increased significantly, compared to the previous solution, as tasks are shared between servers.

The architect stated that *"Functional Correctness", "Resource Utilization", "Configurability", "Accessibility", "Reliability", "Availability", and "Scalability" are the main quality concerns. Additionally, "technology agnostic", "modern web application", and "reusability of the business logic" are the key requirements of AFAS profit.*

Fig. 3. represents how a decision-maker can define the requirements based on the MoSCoW prioritization technique.



Feature Requirement	MoSCoW	Criterion
⊟ GUI		
└ separating the functionality from th...	Must-Have	
└ Separate user interface component...	Must-Have	
└ handling user input	Should-Have	
└ Graphic User Interface (GUI)	Must-Have	
⊞ Instances		
⊞ Application Domain		
⊞ Processing		
⊞ Data Management		
⊞ Abstraction		
⊞ Services		
⊞ Communication		
⊞ Development and Maintanting		
⊞ Quality Aspects		
⊞ Data Access		

Title:

Graphic User Interface (GUI)

Description:

The graphical user interface is a form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation, instead of text-based user interfaces, typed command labels or text navigation.

Details:

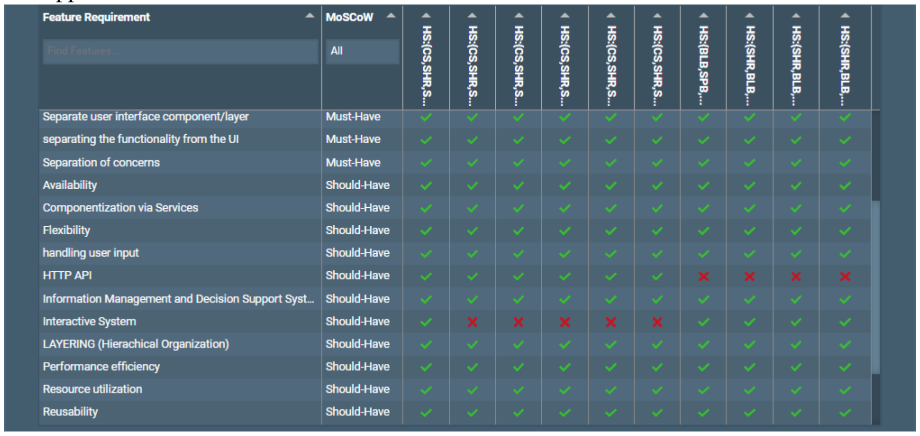
Data type:	Boolean
Category:	GUI
Parent:	NULL
Supported alternatives:	14

Fig. 3 shows the "case definition" of AFAS Profit. The software architect assigned the MoSCoW priorities (Must-Have, Should-Have, Could-Have, and Won't-Have) to the requirements. Note, the data type the features can be either Boolean or Non-Boolean. For instance, "handling user input" is a Boolean feature, which means that a pattern either supports it or not. However, the level of support of "Availability" or "Scalability", as two Non-Boolean features, of a pattern can be "High", "Medium", or "Low".

Case Evaluation - The software architect stated that *AFAS Profit architecture is based on a combination of the "Client-Server", "Publish-Subscribe", and "Layers" patterns. The main rationales behind these design decisions are (1) the frontend can be easily replaced or upgraded, and every module of the business logic, in the back-end, can be reused. (2) The web client communicates over HTTP with the server, so it is possible to choose different technologies for the web client. (3) They can implement a Content Management System (CMS) to make the web client configurable in style and layout. (4) While the data is requested through communication with the server, preventing stale data, the CMS parts are published with some delay, making it possible to cache the style and layout for fast retrieval.*

The inference engine gets the requirements and evaluates the alternative patterns in its knowledge base accordingly. As each pattern supports only a limited set of features, the inference engine has to generate feasible solutions (combinations of patterns). Note, finding a subset of patterns that support all hard-constraints can be formulated as the set cover problem. The DSS uses an algorithm based on the set cover problem to generate several feasible solutions when all patterns in its knowledge base do not support the entire list of hard-constraints of a decision-maker. For instance, Fig. 4 shows that the

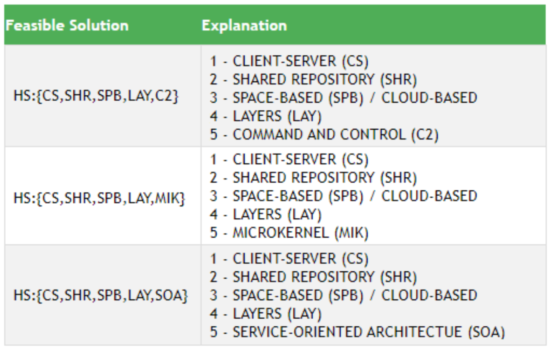
Fig. 4. illustrates part of the case evaluation by the DSS. Ticks (✓) in a row signify that the feature is supported by the corresponding patterns, and crosses (✗) symbolize that the patterns do not support the feature.



Feature Requirement	MoSCoW	HS:{CS,SHR,S...	HS:{CS,SHR,S...	HS:{CS,SHR,S...	HS:{CS,SHR,S...	HS:{CS,SHR,S...	HS:{CS,SHR,S...	HS:{BLB,SPB,...	HS:{SHR,BLB,...	HS:{SHR,BLB,...	HS:{SHR,BLB,...
Find Features...	All										
Separate user interface component/layer	Must-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
separating the functionality from the UI	Must-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Separation of concerns	Must-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Availability	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Componentization via Services	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Flexibility	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
handling user input	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
HTTP API	Should-Have	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗
Information Management and Decision Support Syst...	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Interactive System	Should-Have	✓	✗	✗	✗	✗	✗	✓	✓	✓	✓
LAYERING (Hierachical Organization)	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Performance efficiency	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Resource utilization	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Reusability	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

DSS could not find any patterns that address all the AFAS Profit requirements so that it generated a set of solutions consist of multiple patterns.

Fig. 5. shows top-3 solutions for AFAS Profit.

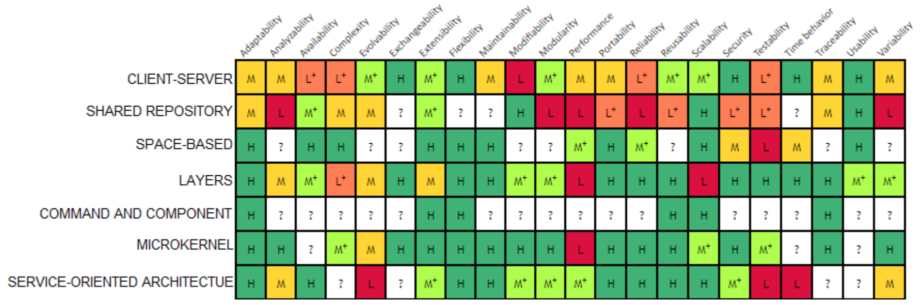


Feasible Solution	Explanation
HS:{CS,SHR,SPB,LAY,C2}	1 - CLIENT-SERVER (CS) 2 - SHARED REPOSITORY (SHR) 3 - SPACE-BASED (SPB) / CLOUD-BASED 4 - LAYERS (LAY) 5 - COMMAND AND CONTROL (C2)
HS:{CS,SHR,SPB,LAY,MIK}	1 - CLIENT-SERVER (CS) 2 - SHARED REPOSITORY (SHR) 3 - SPACE-BASED (SPB) / CLOUD-BASED 4 - LAYERS (LAY) 5 - MICROKERNEL (MIK)
HS:{CS,SHR,SPB,LAY,SOA}	1 - CLIENT-SERVER (CS) 2 - SHARED REPOSITORY (SHR) 3 - SPACE-BASED (SPB) / CLOUD-BASED 4 - LAYERS (LAY) 5 - SERVICE-ORIENTED ARCHITECTUE (SOA)

Patterns tend to be combined to provide greater support for the reusability during the software design process [19]. A pattern can be blended with, connected to, or included in another pattern. For instance, the *Broker* pattern can be connected to the *Client-Server* pattern to form the combined *Client-Server-Broker* pattern [12]. Fig. 5 shows top-3 solutions for AFAS profit. The solutions support all requirements with Must-Have priorities and do not support Won't have requirements (hard-constraints). Note, the DSS generated almost similar solutions that the experienced software architects at

AFAS came up with. Note that the DSS sorts its suggestions based on their scores so that top-3 solutions can be considered the most valuable suggestions.

Fig. 6. show a subset of the mapping between features and patterns used by the DSS to generate solutions for AFAS profit. The primary source of knowledge to build this mapping is the SLR. We employed Fuzzy logic to gain some agreement among the selected studies to calculate the values [11]. Note: High (H), Medium (M), Low (L), Unknown (?).



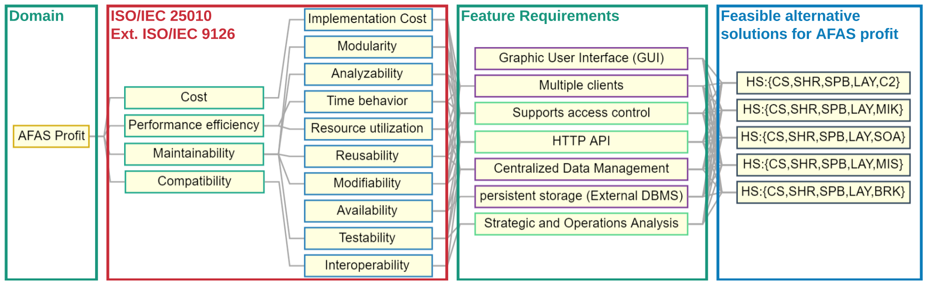
	Adaptability	Analyzability	Availability	Complexity	Evolvability	Exchangeability	Extensibility	Flexibility	Maintainability	Modifiability	Modularity	Performance	Portability	Reliability	Reusability	Scalability	Security	Testability	Time behavior	Traceability	Usability	Variability
CLIENT-SERVER	M	M	L^+	L^+	M^+	H	M^+	H	M	L	M^+	M	M	L^+	M^+	M^+	H	L^+	H	M	H	M
SHARED REPOSITORY	M	L	M^+	M	M	?	M^+	?	?	H	L	L	L^+	L	L^+	H	L^+	L^+	?	M	H	L
SPACE-BASED	H	?	H	H	?	?	H	H	H	?	?	M^+	H	M^+	?	H	M	L	M	?	H	?
LAYERS	H	M	M^+	L^+	M	H	M	H	H	M^+	M^+	L	H	H	H	L	H	H	H	H	M^+	M^+
COMMAND AND COMPONENT	H	?	?	?	?	?	H	H	?	?	?	?	?	?	H	H	?	?	?	H	?	?
MICROKERNEL	H	H	?	M^+	M	H	H	H	H	H	H	L	H	H	H	M^+	H	M^+	?	H	?	H
SERVICE-ORIENTED ARCHITECTUE	H	M	H	?	L	?	M^+	H	H	M^+	M^+	M^+	H	H	H	H	M^+	L	L	?	?	M

The DSS Reports - In the knowledge extraction phase for building the decision model, we observed multiple inconsistencies regarding the impacts of patterns on quality attributes. Some studies reported adverse impacts of a particular pattern on a quality attribute. For instance, *efficiency* can be considered as both strength and liability of the *Pipes and Filters* pattern. We applied fuzzy logic to aggregate the extracted knowledge regarding the potential impacts of patterns on quality attributes. In the implementation of the score calculation (trade-off) phase of the DSS, the impact values range from -2 to 2+. Accordingly, the patterns with more liabilities score lower than those that have more strengths. Note, quantifying the impact of a particular pattern on the quality attributes is complicated because quality attributes are system-wide capabilities. Generally, they cannot be evaluated entirely until the whole system can be evaluated. The DSS evaluates alternative solutions according to decision-makers' quality concerns. Fig 6 shows the impacts of the single solutions for AFAS profit on a subset of quality attributes.
Fig. 7 illustrates a decision structure based on AFAS profit requirements. The DSS automatically generates such decision structures according to the requirements of decision-makers. The first level of the decision structure (Domain) indicates the goal of the decision-making process. The second level denotes the relevant quality attributes that impact the prioritized requirements, which are signified in the third level (requirements). The last level (Feasible Solutions) shows a list of feasible patterns for the decision domain.

4 Related Work

In the SLR [11], we reviewed selected *232* high-quality primary studies for performing the knowledge extraction process. The knowledge base of the SLR, including the pri-

Fig. 7. shows part of the decision structure for the AFAS profit that was generated by the DSS. The domain of the decision-making process is "Finding the best fitting set of patterns for AFAS profit". The qualities are based on the ISO/IEC 25010 [13] quality model. The software architect (decision-maker) defined the feature requirements. The DSS suggested feasible alternative solutions for AFAS profit (last level). Note, the mapping between the qualities and the features was based on domain experts' knowledge; moreover, the relationships among features and patterns were determined based on the SLR [11].



mary studies and extracted knowledge, is available as a technical report on the following web page: http://swapslr.com. We realized that researchers introduced a variety of tools and MCDM techniques to address the pattern selection problem. Notably, there are few tools available for software architects. Architecting is a knowledge-intensive practice, so it can be hard to find the best way to support architects with the right knowledge at the right time. A subset of tools for supporting software architects with their design decisions are presented as follows: *Archium* (www.archium.io) is a visualization tool that produces a view on the functional dependencies between architectural design decisions. It is not an automatic pattern detection or selection, but visualizing the dependencies can help software architects identify such patterns. *ArchReco* (www.cs.ucy.ac.cy/ sielis) provides a design environment that software architects can draw diagrams with predefined shapes that exist in a palette. The description of the shapes is part of a contextual element set that ArchReco's processes suggest the most suitable context-based recommended design patterns. Such Design Patterns are retrieved from several data sources and filtered according to the contextual information that is processed when software architects request recommendations.*Sirius* (www.obeodesigner.com/en/product/sirius) is a tool that enables software architects to graphically design complex systems while keeping the corresponding data consistent (architecture, component properties, etc.). *AKB* (www.se.jku.at/akb-knowledge-sharing) is an implementation and extension of the Architecture Haiku concept, a one-page design description. AKB supports software architects with capturing and sharing of architectural knowledge based on architecture profiles.

The DSS enables software architects to document their drawings and design rationales. We implemented a design studio based on the Unified Modeling Language concepts to store design decisions while the decision-making process. The main difference between the DSS and such tools is that it supports software architects with their decision-making process. In other words, the DSS provides a discussion and negotiation platform to enable software architects to make group decisions. Furthermore, the DSS can be used over the full life-cycle and can co-evolve its advice based on evolving requirements. Software architects can prioritize their functional requirements and quality concerns using the MoSCoW prioritization technique through the user interface of the DSS. Then, the DSS generates a set of feasible solutions that address the requirements.

5 Evaluation

We carried out a study with 24 software architects and developers in the Netherlands to assess the user acceptance of the decision support system and the decision model based on the Technology Acceptance Model. Firstly, we formed 12 groups of two individuals according to their expertise and the companies that they were working with. Next, we introduced the decision model within the DSS portal and presented some of its applications. Then, we assigned the problem definitions of two real-world software architectures to the groups and asked them to design two solutions for the problems. The groups used the decision model within the DSS platform to help them with (1) defining the requirements based on the MoSCoW prioritization technique, and (2) finding the best fitting set of patterns. The group sessions lasted between 45 to 60 minutes. At the end of the sessions, we ask all of the participants to fill out a TAM-based questionnaire; Next, we collected their feedback and opinion about the decision model. The participants highlighted that the decision model, in terms of reusable knowledge regarding the patterns, was a useful tool that can support them to explore more patterns while designing real-world software architectures. They asserted that the decision model assists them in finding liabilities and strength of patterns, their features, and potential application domains that they have employed in.

The DSS assists software architects in the requirements elicitation activity by offering a list of essential features of patterns. Moreover, software architects have different perspectives on their requirements in different phases of the Software Development Life-Cycle. They might want to consider generic domain features in the early phases of the life-cycle, whereas they are interested in more technical features as their development process matures. Therefore, the DSS might come up with various solutions for a software architect in different phases of its software development life-cycle. As the choices of a decision-maker are stored in the DSS knowledge base, it does not cost a significant amount of time to rerun the decision-making process. In a typical scenario, an architect will tweak her decisions and values to assess her choices have on the desired set of patterns. Software architects sometimes have to select a particular set of patterns because of legacy technology choices. Sometimes vendor lock-in makes a customer dependent on a vendor for products and services, unable to use another vendor without substantial switching costs. An example of a pattern that has been trending in recent years is the *Microservices* pattern (see [11]). *Microservices* advantages can tempt architects to consider it as a hammer and convert every design decision into a nail.

Patterns and quality attributes are not independent and have significant interaction with each other. Such interactions can be observed as trade-offs between quality attributes. Software architects need to select and employ an optimal set of patterns to satisfy quality concerns. For instance, some studies assert that *Reusability* is a strength and *Scala-*

bility is a liability of the *Layers* pattern (see [11]). If an architect is looking for both qualities, she has two options: choose another (set of) pattern(s) or use *tactics* to improve *Scalability*. System quality is best exposed in production, independent of whether system quality has been made explicit. We recall that well-known authors, such as Wiegers and Beatty [21], classify quality attributes as external (exposed at the run time/in production, e.g., performance) and internal (exposed at design time, e.g., modifiability). If architects do not think about performance, the system will still expose its performance in the field. The knowledge around the quality of a system under design is hard to gather without *in the field* experiences; however, experience with similar patterns in other systems provides invaluable insight into the inherent qualities of a new system. The DSS recommends patterns that exhibit similar quality behaviors when purely implemented (without tactics) in different systems and that this knowledge can be used by architects to make informed design decisions. We consider it future work to further explore these relationships between patterns and the way in which these communicating properties are best communicated to architects, having to choose from a set of complex solutions.
The tool has been designed using the .Net framework. While it has been optimized somewhat, the tool will sometimes still perform slowly, with end-user wait times of around 5 seconds, which is workable, but not ideal. One of the challenges is the solution space: for recommending solutions (combinations of patterns), the problem's search space is huge, consisting of 29 patterns and 188 features. For instance, for a solution with three patterns, the problem's search space is found to contain $\sim 29 \times 28 \times 27 \times 188$ possible problem states.

6 Conclusion

In this tool paper, we present a DSS besides a decision model for architectural pattern selection. The DSS suggests feasible patterns for particular cases based on the quality concerns and functional requirements of decision-makers. The DSS[3] is accessible through the following link: (https://dss-mcdm.com). We consider it future work to ensure that the knowledge base remains up to date, for instance, through a wiki-mechanism. Thus, software architects can consider the DSS as a source of knowledge and reliable assistance while making decisions regarding the best-fitting set of patterns for their software architectures. Additionally, we should enhance the DSS with a learning module that improves its learnability aspect in the future.
It is presently impossible to assess which patterns are compatible and frequently used in combination, even though practically all systems implement more than one pattern. The knowledge base of the DSS contains individual patterns that solve particular parts of a design problem. The inference engine uses an algorithm based on the set cover problem to generate several feasible solutions when all patterns in its knowledge base do not support the entire list of hard-constraints of a decision-maker.
In our studies, we have dealt with different kinds of architectures, with a slight bias towards enterprise resource planning systems. We consider it as future work to apply the tool to problems in other domains, such as Internet of Things, gaming, or media systems.

[3] Please watch a demo video of the DSS through this link: `https://youtu.be/AhfGYpwpJSQ`

References

1. P. Avgeriou and U. Zdun. Architectural patterns revisited-a pattern language. *European Conference on Pattern Languages of Programs*, 2005.
2. J. Bosch. Software architecture: The next step. In *European Workshop on Software Architecture*, pages 194–199. Springer, 2004.
3. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-oriented software architecture-a system of patterns. *Advances in software engineering and knowledge engineering*, 1:1–487, 1996.
4. P. Clements, R. Kazman, M. Klein, et al. *Evaluating software architectures*. Tsinghua University Press Beijing, 2003.
5. DSDM Consortium. The DSDM agile project framework handbook. *Ashford, Kent*, 2014.
6. A. H. Dutoit, R. McCall, I. Mistrík, and B. Paech. *Rationale management in software engineering*. Springer Science & Business Media, 2007.
7. S. Farshidi, S. Jansen, R. De Jong, and S. Brinkkemper. A decision support system for cloud service provider selection problems in software producing organizations. In *2018 IEEE 20th Conference on Business Informatics (CBI)*, volume 1, pages 139–148. IEEE, 2018.
8. S. Farshidi, S. Jansen, R. de Jong, and S. Brinkkemper. A decision support system for software technology selection. *Journal of Decision Systems*, 2018.
9. S. Farshidi, S. Jansen, R. De Jong, and S. Brinkkemper. Multiple criteria decision support in requirements negotiation. In *the 23rd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2018)*, volume 2075, pages 100–107, 2018.
10. S. Farshidi, S. Jansen, S. España, and J. Verkleij. Decision support for blockchain platform selection: Three industry case studies. *IEEE Trans. on Engineering Management*, 2020.
11. S. Farshidi, S. Jansen, and J. M. van der Werf. Capturing software architecture knowledge for pattern-driven design. *Journal of Systems and Software*, 2020.
12. N. B. Harrison and P. Avgeriou. How do architecture patterns and tactics interact? a model and annotation. *Journal of Systems and Software*, 83(10):1735–1758, 2010.
13. ISO. IEC25010: 2011 systems and software quality requirements and evaluation (SQuaRE). *International Organization for Standardization*, 34:2910, 2011.
14. P. Lago and P. Avgeriou. First workshop on sharing and reusing architectural knowledge. *ACM SIGSOFT Software Engineering Notes*, 31(5):32–36, 2006.
15. M. Majumder. Multi criteria decision making. In *Impact of urbanization on water shortage in face of climatic aberrations*, pages 35–47. Springer, 2015.
16. D. J. Power. Decision support systems: a historical overview. In *Handbook on decision support systems 1*, pages 121–140. Springer, 2008.
17. N. Rozanski and E. Woods. *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2012.
18. A. Tang, P. Liang, and H. Van Vliet. Software architecture documentation: The road ahead. In *the 9th Working IEEE Conference on Software Architecture*, pages 252–255. IEEE, 2011.
19. M. T. T. That, S. Sadou, F. Oquendo, and I. Borne. Composition-centered architectural pattern description language. In *European Conference on Software Architecture*, pages 1–16. Springer, 2013.
20. H. Wang. Intelligent agent-assisted decision support systems: integration of knowledge discovery and knowledge analysis. *Expert Systems with Applications*, 12(3):323–335, 1997.
21. K. Wiegers and J. Beatty. *Software requirements*. Pearson Education, 2013.
22. O. Zimmermann. Architectural decisions as reusable design assets. *IEEE software*, 28(1):64–69, 2010.