IADIS International Journal on Computer Science and Information Systems
Vol. 7, No.1, pp. 106-119
ISSN: 1646-3692

SELECTING ARCHITECTURAL PATTERNS THROUGH A KNOWLEDGE-BASED APPROACH

Liziane Santos Soares, Roberto Tom Price, Marcelo Soares Pimenta. *Instituto de Informática, Universidade Federal do Rio Grande do Sul. Caixa Postal 15064, 91501-970. Porto Alegre, RS, Brazil.*

José Luis Braga. *Departamento de Informática, Universidade Federal de Viçosa, Campus da UFV, 36570-000, Viçosa-MG, Brazil.*

ABSTRACT

Several architectural patterns are described in books, papers and in repositories. Due to the huge amount of information related to patterns description and their application, it is important to have approaches and tools available to help developers carry on the selection of the patterns more suited to each software project. The reasoning behind architectural decisions, regarding pattern selection, usually exists in the form of tacit knowledge. It is important to make this knowledge explicit by mapping the patterns and the domains where they are most often used. This work proposes a knowledge-based approach to accomplish the architectural patterns selection, according to each project. The approach aims at supporting the construction and maintenance of a knowledge-base to guide the selection. We built an initial base as a suggestion, relying on recommendations of specialists found in the literature. Each team or institution can build its own base or enhance an existing one through the structure offered by the approach. The input for pattern selection consists, basically, of information available in systems requirements. The proposal presented is one step towards the automation of architectural pattern selection during on architectural design.

KEYWORDS

Architectural patterns selection, software architecture, knowledge-based approach.

1. INTRODUCTION

Designing software architecture involves making decisions that have great impact on systems in terms of efficiency, re-usability, maintainability, capacity to evolve, and other quality attributes. Most of the architectural decisions have multiple consequences in the system. Kruchten et al. (2006) highlights that, with the exception of the resulting architectural design, most of the knowledge about architecture is usually not documented, and this knowledge remains inside the architects' minds. The reasoning behind an architectural decision, although essential, usually exists in the form of tacit knowledge. For a certain project, this kind of knowledge may comprise the resulting architecture design as well as the architectural decisions made to build the design, the context of the development and other factors that, together, determine the selection of a particular solution.

Architectural patterns provide predefined structured schemas, including the description of elements, their responsibilities and the rules and guidelines for organizing the relationship between these elements (Buschmann et al. 1996). Several benefits may be obtained through pattern usage. One of them is the encapsulation and reuse of successful solutions used previously. Patterns also contribute for documenting the adoption of solutions, capturing information about their structure and behavior. They provide information about the reasoning related to themselves, the consequences of their application and the motivation for their usage (Harrison et al. 2007).

The selection of architectural patterns comprises several architectural decisions. It shall consider the patterns more appropriate for each system according to its requirements. Another important point is the analysis and exclusion of patterns that would present any kind of conflict among them. Generally, experienced architects are able to make these kinds of decisions, even in an implicit manner. It is important to make this knowledge explicit by mapping the patterns and the domains where they are most often used and analyzing the reasoning behind this mapping (Harrison & Avgeriou 2007). That knowledge can provide support for the task of defining the system architecture for both the developer who has a huge experience in software architecture and for inexperienced developers.

There are several architectural patterns described in books, papers and in repositories (Buschmann et al. 1996; Fowler 2002; Schmidt et al. 2000; Kircher & Jain 2004; Buschmann, Henney & Schmidt 2007b; Buschmann, Henney & Schmidt 2007a; Cunningham n.d.; Group n.d.; Booch n.d.). They are focused on different domains and are generally described in a textual manner. Due to the huge amount of information related to pattern description and decisions to select them, it is also important having techniques, methods and tools available to help architects and developers accomplishing the selection of patterns more suited to each system.

This work proposes a knowledge-based approach to accomplish the architectural patterns selection, according to each project. The approach aims at supporting the construction and maintenance of a knowledge-base to guide the selection. We built an initial base as a suggestion, relying on recommendations of specialists found in the literature. Each team or institution can build its own base or enhance an existing one through the structure offered by the approach. The input for the pattern selection consists, basically, of information available in systems requirements specification.

This paper is organized as follows. Section 2 describes the development of the approach structure which includes the construction of a knowledge-base containing elements necessary to accomplish the pattern selection. Section 3 presents a proof-of-concept for the proposed approach, in the form of a Prolog program. An example of its execution is presented as well as how to enhance the knowledge-base. Section 4 discusses related work and Section 5 concludes the paper.

2. APPROACH FOR PATTERNS SELECTION

2.1 Construction of the Knowledge-base

In the literature, there are several repositories of architecture patterns (Buschmann et al. 1996; Fowler 2002; Schmidt et al. 2000; Kircher & Jain 2004; Buschmann, Henney & Schmidt 2007b; Buschmann, Henney & Schmidt 2007a; Cunningham n.d.; Group n.d.; Booch n.d.; Rising 2000). Some of them are related to specific types of system (Schmidt et al. 2000; Buschmann, Henney & Schmidt 2007b; Kircher & Jain 2004) and others have a wider scope (Buschmann et al. 1996; Fowler 2002; Buschmann, Henney & Schmidt 2007a; Cunningham n.d.; Group n.d.; Booch n.d.; Rising 2000). Patterns are generally described by texts using natural language, which is not precise enough to be treated by a computer. The description is organized in sections such as context, problem, structure, dynamics (including possible scenarios), examples, variants and known uses and consequences. Information about the problem to be solved and the pattern's solution are, sometimes, mixed in the sections (Buschmann et al. 1996). In a pattern description, we can mine information about its impact on certain quality attributes. Moreover, we can find information about features of a system for which the pattern is more suitable.

In this work an initial base was built based on several studies that focus on software architecture methods including the use of architectural patterns (Avgeriou & Zdun 2005; Bachmann et al. 2005; Bass et al. 2003; Bosch 2000; Buschmann et al. 1996; Shaw & Garlan 1996; Hofmeister et al. 2005). Among them, we find information about patterns, their properties, the analysis of their application as well as their consequences; contexts where they are applicable and relationships among them. This kind of information allowed the construction of a knowledge-base to guide the selection of architectural patterns according to each system. The knowledge-base initially built regards the set of architectural patterns described in (Buschmann et al. 1996). The set was prioritized because it embraces classic and well known patterns with a broader scope. However, each team or organization can build and enhance its own base using information from its own experience in previous projects, or from experience of specialists.

Patterns may be kept preserving the original sections from their description (listed in the beginning of this section). Although, only their names are relevant for the selection approach itself.

2.2 Non-functional Requirements as Input for the Selection Approach

The architecture is designed to address a set of stakeholder needs, concerned with functional and non-functional requirements. A common concept behind several definitions is that the NFRs describe qualities or characteristics which software should possess and constraints which it should meet. The constraints are related to the system being developed and to the development process (Glinz 2007). The properties and characteristics can be related to quality attributes and other issues such as: appearance, platform, efficiency and accuracy. NFRs are quite often the most significant requirements which an architect is concerned about (Eeles 2006). According to Buschmann et al. (1996), every architectural pattern denotes a relationship between a specific context, a given problem and a solution. The context for architectural patterns application can be extracted from the system requirements, particularly from the non-functional requirements.

System qualities are related to quality attributes that are desired characteristics of software, associated with non-functional requirements (Mylopoulos et al. 1992). Common examples of these attributes are *usability, portability, reliability, security, efficiency* and *maintainability*. The constraints specify in which manner something in the system may be realized (Lawrence Chung & Leite 2009), and are related to structural aspects such as interface, platform, and physical distribution. They may be expressed as features desired for the system. For example, how the interface shall be: interactive with the user or no (in the case of interaction with other system)? How the distribution of the system shall be: stand-alone or distributed? The system shall be designed to be web-based? All these questions are related to desired features for the system.

Information about non-functional requirements of each system, including system qualities and features (constraints), can be found in requirements artifacts such as *Vision, Use Cases* (specifically in the *Special Requirements* section of the Use Case) and *Supplementary Specification* from RUP (Kruchten 2004); or other artifacts with similar purpose. Thus, we define desired quality attributes and features as the input for our approach. The following sections detail the relation between architectural patterns and quality attributes; and between architectural patterns and features. The relations correspond to facts in the knowledge-base.

2.3 Relation between Patterns and Quality Attributes

Over the years, a number of factors constituting characteristics and behavior of the software have been identified and associated with quality attributes (McCall, 1977). An attribute is a quality criterion which can be used to evaluate the performance of a system.

Pattern descriptions contain information about consequences of their usage (Buschmann et al. 1996). The analysis of the consequences allows discovering the liabilities and strengths of each pattern related to quality attributes. A certain pattern may impact a certain quality attribute positively, negatively or does not present any impact (Harrison & Avgeriou 2007).

This information may be reinforced by the analysis of the impact. Consider, for example, the pattern Layer and its impact over *maintainability*. In this case, the impact is considered to be positive, namely, it contributes for the system maintainability (Buschmann et al. 1996). The structure proposed by the pattern *Layer* complies with the *Common-Closure Principle* (CCP) (Martin 2002) where the classes susceptible to changes for the same reason are put in a same place. It minimizes the effort of releasing and revalidation, which contributes to enhance the maintainability of the system. Similar analysis can be made regarding other patterns and their impact on quality attributes, but this is not in the scope of this work. What is important here is highlighting that patterns present an impact on one or more quality attributes and those impacts can be mined from their description and from literature (Buschmann et al. 1996; Harrison & Avgeriou 2007; Harrison & Avgeriou 2008).

In order to build the knowledge-base, we consider three possible values for the impact: *positive*, *negative* or *neutral*. Table 1 presents a subset of impacts mined from the literature, only to illustrate our discussion. Each pattern may affect more than one quality attribute in different ways. The pattern *MVC (Model View Controller)*, for example, impacts *negatively* the *Maintainability* and *Portability*, whereas impacts *positively* the *Usability*.

Table 1. Impact of patterns over quality attributes.

PATTERN	IMPACT	QUALITY ATTRIBUTE
Layers	Positive	Maintainability
MVC	Negative	Maintainability
Pipes & Filters	Negative	Security
Broker	Positive	Security
Broker	Positive	Usability
Broker	Neutral	Reliability
Layers	Positive	Portability
MVC	Positive	Usability
Broker	Positive	Portability
MVC	Negative	Portability

The relationship between a quality attribute and a pattern consists of the impact of the latter on the former. Each impact constitutes a fact within the knowledge base and is used in the rules for pattern selection. The representation of this relationship gathers the pattern in question, the quality attribute impacted by the pattern and the type of impact (*positive*, *negative*, or *neutral*).

```
Impact (Pattern, QA, Type)
```

Other impacts can be added to the knowledge-base by the team. The quality attributes in the base can be obtained from well-established quality models as those presented in (McCall 1977; Boehm et al. 1978; Grady 1992; Schulmeyer & McManus 1998; ISO/IEC 2001).

2.4 Relation between Patterns and Features of the System

As was said previously, non-functional requirements include constraints specifying in which way something in the system should be realized. For example, how the interface should be: interactive with the user or no (interaction with other system)? How the distribution of the system shall be: stand-alone or distributed? Is the system supposed to be executed on-line? The three questions deal with features we call *interactive*, *distribution*, *web-based*.

A certain pattern may be suitable to be adopted in a project, depending on a desired feature. For example, if a system is intended to be interactive with a graphical interface, the patterns *Model View Controller* (MVC) or *Presentation Abstraction Controller* (PAC) are suitable (Buschmann et al. 1996). If the system is intended to be distributed, the pattern *Broker* can be used (Buschmann et al. 1996).

With this in mind, we assume that a *feature* is related to a certain structural issue of the system as platform, interface, distribution and so on. There may be different possible constraints regarding each issue, thus each feature may be associated to a set of values, where each value corresponds to a possible constraint. For the purpose of structuring our approach, each feature is included in the knowledge-base together with their possible values.

For example, considering the feature *distribution* (related to the system structure and not to the manner by which its versions are released to the users), we have two possible values for it: *stand-alone* or *distributed*, which means that there are two possible constraints for the distribution of a system: it may be *distributed* or may be *stand-alone*.

It is important to highlight that the team can define features and values in the way that best suit them. For example, the feature interactive could have been defined as: Interaction, and the possible values: *user*, *other_systems* and *both*. Table 2 shows a set of features and their respective possible values, supported by information extracted from the literature.

Table 2. A set of features and their possible values.

FEATURE	POSSIBLE VALUES
adaptable	[yes, no]
with_nondeterministic_solution	[yes, no]
real_time	[yes, no]
distribution	[distributed, stand_alone]
embedded	[yes, no]
interactive	[yes, no]
gui_based	[yes, no]
web_based	[yes, no]

The feature *adaptable* refers to the characteristic of the system being able to adapt and change itself in certain points. Namely, the value *yes* for *adaptable* indicates that the system has a structure that allows changing its behavior or structure *a priori*, without the necessity of an explicit software maintainace and a release of a new version. Systems with computational reflection fall in this case. The feature *with_nondeterministic_solution* is related to a system whose problem does not present a deterministic solution. In general, a system that involves heuristic computation or any support for artificial intelligence presents value *yes* for that feature. The feature *real time* indicates if the system presents restrictions on response time during the processing (*yes*) or not (*no*).

The feature *interactive* specifies whether the system should be interactive with the user (*yes*) or not (*no*). *Gui(graphical user interface)_based* is the feature that indicates whether the system interface with the user should be graphical (*yes*) or not (*no* - for example, an interface that uses a command prompt). These last two features are intimately linked, since *interactive* is *no*, *gui_based* will be also. In addition, some patterns are applicable only if both are yes such as the *Model View Controller* and *Presentation Abstraction Control*.

Distribution refers to the way the system will be executed. The system can be executed centrally from a computer (*stand_alone*) or in a decentralized manner through two or more computers connected via a network (*distributed*). The feature *web_based* is related to systems that are on a server, accessible through a Web browser. In general, these systems include code in languages supported by Web browsers.

Each system can present a specific configuration of values for the features. Consider, for example, a dedicated system to control a dispositive including mechanical parts. It consists of an embedded system and probably presents restrictions on the response time for processing. Besides, nowadays that type of system tends to offer human interfaces. A possible configuration for a system like this may be [adaptable=*no*, with_non_deterministic_solution=*no*, embedded=*yes*, real_time=*yes,* interactive=*yes* gui_based=*no,* distribution=*stand-alone*, web_based=*no*]. If we consider an e-commerce system, a possible configuration may be [adaptable=*no*, with_non_deterministic_solution=*no*, embedded=*no*, real_time=*no,* interactive=*yes*, gui_based=*yes,* distribution=*distributed*, web_based=*yes*].

Each feature can be associated to a set of possible values. The features and the possible values constitute constants in the knowledge-base. The representation of a feature that has a certain value constitutes a fact in the base.

```
Feature (Value)
```

A particular value for a feature may indicates a certain pattern more applicable to the application context. Considering a specific project, only one value can be attributed for each feature. Other features and their possible values can be added to the base by the team or institution. Not only the features but also their values are used in the rules for pattern selection.

2.5 Pattern Selection Rules

In the previous sections we set the grounds for the construction of our knowledge-base (2.1). Then, we discussed about software architecture and non-functional requirements and we specified two kinds of elements to be the input for our approach: quality attributes and features for each system (2.2). Next we described and illustrated the relation between the input elements and architectural patterns (2.3 and 2.4).

Our next step consists in formulating the types of rules in the base. They are built aiming at representing recommendations found in the literature and also those extracted from specialists and teams. They involve the elements discussed previously, like quality attributes (QA), features and their values, the impact of patterns over quality attributes and the suitability of patterns according to the features.

(i) Recommendations of a pattern based on its impact over quality attributes

```
QA Λ Impact (Pattern, QA, positive) → Pattern
```

Suppose the system requirements specify *Security* as a desired QA. Patterns that help achieving this requirement include *Layer* or *Broker* (Harrison and Avgeriou, 2007; Harrison and Avgeriou, 2008). Each pattern presents an impact on one or more quality attributes, being this impact positive, negative or null. This knowledge can be used in order to compose rules where depending on a desired QA, a set of patterns, which affect positively the QA, are selected. For example:

security Λ Impact (broker, security, positive) → *broker*

(ii) Recommendation of a pattern based on the values for the features

```
Feature(Value) Λ … Λ Feature(Value) → Pattern
```

As previously said, each system presents a configuration of values for the features. This type of rule represents recommendations about patterns that are more suitable according to a specific value for a feature. For example, in the case of a system that is intended to be distributed, pattern Broker is suitable. In the case of a system that is intended to be *Interactive (yes)* and *gui_based (yes),* patterns such as MVC or PAC are suitable (Buschmann et al., 1996).

Interactive (yes) Λ GUI-based (yes) → *mvc*
Distribution(distributed) → *broker*

(iii) No recommendation of a pattern based on the values for the features

```
Feature(Value) Λ … Λ Feature(Value) → ¬ Pattern
```

Rules can be created for patterns that are not suitable according to certain values of features. For instance, if the system is intended to be stand-alone *Distribution(stand-alone)* the pattern *Broker* is not suitable:

Distribution(stand-alone) > → ¬ *broker*

Here, as we are building an initial base, the rules consider features listed in Table 2. As the base is enhanced with other features and their possible values, rules involving them should be included. The processing of rules considers, firstly, rules of type (i) to obtain an intermediary list of patterns. The rules of type (ii) and (iii) are processed as subsequent steps to adding more appropriate patterns or eliminating those that are incompatible according to the values. After processing all the rules, the approach for pattern selection generates a list of candidate patterns to be applied to the architectural project.

2.6 Elements of the Knowledge-Base

This section describes the conceptual model containing the main concepts associated with the patterns selection rules. Figure 1 shows the classes representing the elements of the base.

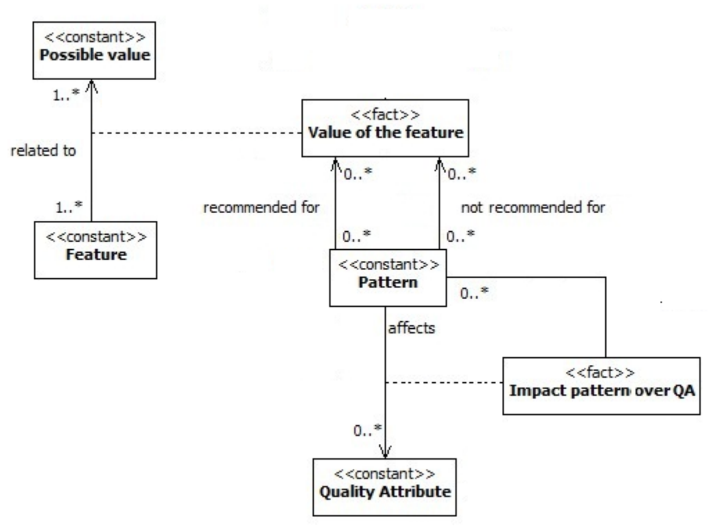

Figure 1. Metamodel for knowledge-base elements.

In the metamodel, classes are stereotyped with the type of role performed by each element in the logical language. The classes *Feature*, *Possible Value*, *Pattern* and *Quality Attribute* consist of constants in the logical language. The other classes consist of facts in logical language:

- *Impact pattern over QA*: this fact is related to the relationship described in Section 2.3, between patterns and quality attribute (`Impact (Pattern, QA, Type)`).
- *Value of the feature:* this fact is related to the relationship described in Section 2.4, between features and values (`Feature (Value)`). It associates a feature to one of its possible values.

The rules for pattern selection are derived from the relationships among the classes: *Pattern*, *Impact pattern over QA, Value of the feature.*

3. PROTOTYPE IMPLEMENTATION

As a proof-of-concept for our approach, we built a prototype in Prolog. The key idea is translating the elements discussed until here into Prolog clauses. The SWIProlog[1] inference engine is used for rule processing, generating a list o suggested patterns. We decided to implement the rules in Prolog to benefit from a language and an inference engine widely used for developing knowledge-based systems.

Our knowledge-base includes quality attributes, patterns, features of the systems (and their possible values), impact of patterns over quality attributes, recommendations of patterns more suitable or not according to features. Table 3 shows how those elements are represented inside the Prolog program. The rules described in Section 2.5 are translated into Prolog according to Table 4, using elements of Table 3. The program input consists of a text file, including a list of quality attributes and the values for the features defined in the base. Each supplied value for a certain feature must be one of the possible values defined in the base. The text file must be in conformance with a predefined format and is specific for each system architecture design. After processing the rules, the final result is a list of candidate architectural patterns to be applied to the system architecture.

Table 3. Prolog representation of elements in the knowledge-base.

ELEMENT OF THE BASE	REPRESENTATION IN PROLOG	EXAMPLE
Quality attribute	Constant	*usability*
Pattern	Constant	*mvc*
Feature	Constant	*distribution*
Possible value for a feature	Constant	*distributed* *stand_alone*
Value of a feature	Predicate	*feature(distribution, distributed)*
Impact of a pattern over an attribute	Predicate	*impacts(mvc,usability, positive)*
Suitable pattern	Predicate	*apply(mvc)*
Not suitable pattern	Predicate	*notapply(broker)*

[1] http://www.swi-prolog.org/

Table 4. Translating rules to Prolog.

RULES	EXAMPLES OF REPRESENTATION IN PROLOG
(i) `QA Λ Impact (Pattern, QA, positive)> → Pattern`	*bagof(Pattern, impacts(Pattern, security, positive), L)*
(ii) `Feature(Value) Λ … Λ Feature(Value) → Pattern`	*apply(broker):-feature(distribution,yes).* *apply(mvc):-feature(interactive,yes), feature(gui_based,yes) .*
(iii) `Feature(Value) Λ … Λ Feature(Value) → ¬ Pattern`	*notapply(broker):-feature(distribution,stand_alone)*

A Program Execution Example

For the purpose of exemplifying our approach, we consider an on-line system for academic control to be used by students and professors of a university. In this case, the input file is shown in Figure1(a). According to a pre-defined format, in the first part of the file, each line contains a quality attribute. Zero or more quality attributes can ne listed in the file, however, it is important to remember that quality attributes can be conflicting with each other (Boehm & In 1996), thus a very high number of attributes can make the selection of patterns inefficient. The second part begins with a line containing the expression **features** and each line contains a feature and its respective value (space separated).

In order to execute the Prolog program we use the predicate *patternSelection/1*, for which the entry is the name (and the path if needed) of the input file. The program execution results in the list of patterns shown in Figure1(b).

```
Security
usability
reliability
*features*
distribution distributed
embbeded no
real_time no
interactive yes
gui_based yes
web_based yes
adaptable no
non_deterministic no
```

Figure 1(a). Input file for our case study.

```
SUGGESTED PATTERNS:
pac
mvc
broker
layer
```

Figure 1 (b). Result of the program execution.

3.1 Augmenting the Knowledge-Base

During the presentation of our approach, we suggested an initial base built from recommendations of specialist found in the literature. However, each team or institution can build its own base from information of previous projects or other sources. In order to enhance or to build the knowledge-base, the user needs to deal only with a few elements established by the approach. The remainder of the program is already structured to accomplish the rules processing using the elements. Table 5 presents what is necessary to add in each case.

Table 5. New elements representation in Prolog.

NEW ELEMENT	REPRESENTATION IN PROLOG
Pattern	These elements are constants in the program and can just be used in new facts and rules.
Quality attribute	
Feature	
Possible value	
Value for a feature	*feature(Feature, Value).*
Impact of a pattern over a quality attribute	*impacts(Pattern, Quality_attribute, Impact).*
Suitable pattern	These elements are used directly in the rules (ii) and (iii) through the predicates *apply/1* and *notapply/1.*
Not suitable pattern	
Rules of type (i) (Table 4)	For this rule, the addition of *impacts of the patterns over each attribute* (as mentioned above in this table) is sufficient.
Rules of type (ii) (Table 4)	*apply(Pattern):- feature(Feature, Value), ... , feature(Feature, Value).*
Rules of type (iii) (Table 4)	*notapply(Pattern):- feature(Feature, Value), ... , feature(Feature, Value).*

4. RELATED WORK

Jansen & Bosch (2005) proposes an approach for defining the relationship between design decisions and software architecture through a metamodel. The information about the decisions, on which the architecture is based, is not lost and can be represented in a graphical way. The framework proposed in (Babar et al. 2006; Babar & Ian Gorton 2007) includes a data model that characterizes architectural knowledge from constructs, their attributes and relationships. The framework stores scenarios and their description, as well as patterns and their properties. The framework allows the search of patterns and scenarios through some fields. In these cases, we do not observe any kind of inference or automatic selection over the stored knowledge. The architectural experience of the developer keeps being the most important role.

Birukou (2010) realizes a survey about approaches for pattern search and pattern selection. He defines important problems related to the search and selection of patterns, and analises existing approaches according to the problems. Besides he defines properties under which he classifies the revised approaches. Part of the issues treated by the author is directly related to our approach for pattern selection.

In (Wang et al. 2005) patterns are selected to fit the non-functional requirements of the architectural design. A non-functional requirements framework (NFR) is used to fetch a preliminary list of prioritized patterns that can be appropriated to the system. The applicability of those patterns is then analyzed and determined using the NFR. In our approach, the applicability of patterns is based on recommendations. In (Hang et al. 2007) a method is proposed for pattern selection using a pattern clustering analysis algorithm and a collaborative filtering recommendation algorithm. This method deals with requirement analysis patterns for e-Business applications whereas our approach has a wider focus, including patterns used to build software architecture. Weiss & Birukou (2007) propose a multi-agent system for recommending patterns. The system supports conventional information retrieval and case-based reasoning methods for realizing the recommendations, which are based in past user actions in the system.

Other studies are focused on the development of software architecture based on the use of architectural patterns (Booch n.d.; Bass et al. 2003; Bosch 2000; Buschmann et al. 1996; Shaw & Garlan 1996; Hofmeister et al. 2005). Among them we find descriptions of patterns, including their properties, the analysis of their application as well as their consequences, contexts where they are applicable and some relationships among them. These studies do usually not involve any support to an automatic selection of patterns. In these cases, the architect will be the one who decides what are the ones more appropriated for the context of certain project. The architect shall have the capability of recognizing the most appropriate pattern or patterns for each situation, according to their description. From these studies we can capture several rules that relate features and quality attributes of the system to applicable patterns.

5. CONCLUSION

This work proposes a knowledge-base approach to select architectural patterns during the architecture design. An initial knowledge-base was built as a suggestion, based on recommendations of specialists found in the literature. As a proof-of-concept, the proposed approach is implemented in Prolog. In the case where a team wants to build its own base or enhance an existing one, it is necessary just to add some kind of elements to the remainder of the program, according to what is established by the approach.

Currently, the drawback of the approach is that the construction of the input file and the adding of new elements into the base is made manually which is error prone. Thus our next step is integrating the Prolog program with a graphical interface in order to automate these tasks. Besides, the interface will allow storing patterns, features, quality attributes, projects and their respective information. As a future work, we will focus on identify more specific aspects of the architecture in order to accomplish a more fine grained selection of patterns.

The proposed approach aims at offering support for developers, mainly to those that are not specialists in software architecture. It allows them to rely on experience from specialists, previous projects and other sources, stored in a knowledge-base. The proposal is one step towards the automation of the architectural pattern selection.

ACKNOWLEDGEMENT

This work was partially funded by CNPq - Conselho Nacional de Desenvolvimento Científico – Brazil.

REFERENCES

Avgeriou, P. & Zdun, U., 2005. Architectural Patterns Revisited – A Pattern Language. In *10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*. Irsee, Germany.

Babar, M.A., Gorton, I & Kitchenham, B., 2006. A framework for supporting architecture knowledge and rationale management. In A. H. Dutoit et al., eds. *Rationale Management in Software Engineering*. Springer, pp. 237-254.

Babar, M.A. & Gorton, Ian, 2007. A Tool for Managing Software Architecture Knowledge. *Second Workshop on Sharing and Reusing Architectural Knowledge Architecture Rationale and Design Intent SHARKADI07 ICSE Workshops 2007*, 4(2), pp.11-11.

Bachmann, F. et al., 2005. Designing software architectures to achieve quality attribute requirements. *IEE ProcSoftw*, 152(4), pp.153–165.

Bass, L., Clements, P. & Kazman, R., 2003. *Software Architecture in Practice*, Addison-Wesley.

Birukou, A., 2010. A survey of existing approaches for pattern search and selection. In P. Avgeriou & M. Weiss, eds. *European Conference on Pattern Languages of Programs, EuroPLoP '10, Irsee Monastery, Bavaria, Germany, July 7-11, 2010*. Bavaria, Germany: ACM, p. 2.

Boehm, B. et al., 1978. *Characteristics of Software Quality*, TRW Series of Software Technology.

Boehm, B. & In, H., 1996. Identifying quality-requirement conflicts. *IEEE Software*, 13(2), pp.25-35.

Booch, G., Handbook of Software Architecture. Available at: http://www.handbookofsoftwarearchitecture.com.

Bosch, J., 2000. *Design & Use of Software Architectures*, London: Addison-Wesley Professional.

Buschmann, F. et al., 1996. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns* J. W. & Sons, ed., England.

Buschmann, F., Henney, K. & Schmidt, D., 2007a. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*, Wiley.

Buschmann, F., Henney, K. & Schmidt, D., 2007b. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*, Wiley.

Chung, Lawrence & Leite, J.C.S. do P., 2009. On Non-Functional Requirements in Software Engineering. *Lecture Notes in Computer Science*, 5600, pp.363-379.

Cunningham, W., Portland Pattern Repository. Available at: http://c2.com/ppr/.

Eeles, P., 2006. What is a software architecture? *IBMN*. Available at: http://www.ibm.com/developerworks/rational/library/feb06/eeles/.

Fowler, M., 2002. *Patterns of Enterprise Application Architecture*, Addison-Wesley.

Glinz, M., 2007. On Non-Functional Requirements. *15th IEEE International Requirements Engineering Conference (RE'07)*, pp.21-26.

Grady, R.B., 1992. *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall.

Group, H., Patterns Library. Available at: http://hillside.net/patterns/.

Hang, G.S. et al., 2007. A Requirement Analysis Pattern Selection Method for E-Business Project Situation. In *Proceedings of the IEEE International Conference on e-Business Engineering. ICEBE '07*. Washington, DC, USA: IEEE Computer Society, pp. 347–350.

Harrison, N.B. & Avgeriou, P., 2008. Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation. *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pp.147-156.

Harrison, N.B. & Avgeriou, P., 2007. Leveraging Architecture Patterns to Satisfy Quality Attributes. *Lecture Notes in Computer Science*, 4758, pp.263-270.

Harrison, N.B., Avgeriou, P. & Zdun, U., 2007. Using Patterns to Capture Architectural Decisions. *IEEE Software*, 24(4), pp.38-45.

Hofmeister, C. et al., 2005. Generalizing a Model of Software Architecture Design from Five Industrial Approaches. *5th Working IEEEIFIP Conference on Software Architecture WICSA05*, pp.77-88.

ISO/IEC, 2001. *Software engineering - Product quality - Part 1: Quality model, ISO/IEC 9126-1:2001*,

Jansen, A. & Bosch, J., 2005. Software Architecture as a Set of Architectural Design Decisions. *5th Working IEEEIFIP Conference on Software Architecture WICSA05*, pp.109-120.

Kircher, M. & Jain, P., 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*, Wiley.

Kruchten, P., 2004. *The Rational Unified Process: an Introduction* 3rd ed., Addison-Wesley.

Kruchten, P., Lago, P. & Van Vliet, H., 2006. Building Up and Reasoning About Architectural Knowledge. In Christine Hofmeister, I. Crnkovic, & R. Reussner, eds. *Quality of Software Architectures*. Springer Berlin Heidelberg, pp. 43 - 58.

Martin, R.C., 2002. *Agile Software Development, Principles, Patterns, and Practices*, Prentice Hall.

McCall, J.A., 1977. *Factors in Software Quality, RADC-TR- 77-369, 13441-5700*, Nova York.

Mylopoulos, J., Chung, Lawrence & Nixon, B., 1992. Representing and using nonfunctional requirements: a process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6), pp.483-497.

Rising, L., 2000. *The Pattern Almanac 2000*, Addison Wesley.

Schmidt, D. et al., 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*, Wiley.

Schulmeyer, G.G. & McManus, J.I., 1998. *Handbook of Software Quality Assurance*, Prentice Hall PTR.

Shaw, M. & Garlan, D., 1996. *Software Architecture: Perspectives on an Emerging Discipline* M Shaw & D Garlan, eds., Prentice Hall.

Wang, J., Song, Y.T. & Chung, L., 2005. No TitleFrom software architecture to design patterns: A case study of an NFR approach. In *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and ACIS International Workshop on Self-Assembling Wireless Networks.* Washington, DC, USA,: Society, IEEE Computer, pp. 170-177.

Weiss, M. & Birukou, A., 2007. Building a pattern repository: Benefitting from the open, lightweight, and participative nature of wikis. In *Workshop on Wikis for Software Engineering at ACM WikiSym, 2007 International Symposium on Wikis (WikiSym)*. Montreal, Quebec, pp. 21-23.