Chapter 10
Software Architecture:
Developing Knowledge, Skills, and Experiences

Perla Velasco-Elizondo
Universidad Autónoma de Zacatecas (UAZ), México

ABSTRACT

What is software architecture? A clear and simple definition is that software architecture is about making important design decisions that you want to get right early in the development of a software system because, in the future, they are costly to change. Being a good software architect is not easy. It requires not only a deep technical competency from practicing software architecture design in industry, but also an excellent understanding of the theoretical foundations of software architecture are gained from doing software architecture research. This chapter describes some significant research, development, and education activities that the author has performed during her professional trajectory path to develop knowledge, skills, and experiences around this topic.

INTRODUCTION

What is software architecture? To say it simple: software architecture is about making the design decisions that you want to get right early in the development of a software system, because future changes are costly. Today, software architecture development is necessary as never before; no organization begins a complex software system without a suitable software architecture.

Within the context of the software life cycle (Sommerville, 2011), software architecture is an artifact produced during the design phase. A software architect, or the software architecture design team, is responsible for defining software architecture. Being a good software architect is not an easy matter (Rehman et al., 2018), (Shahbazian, Lee & Medvidovic, 2018). The author considers that, it not only requires deep technical competency which comes from practicing software architecture design in industry; but also a very good understanding of the theoretical foundations of software architecture gained from doing software architecture research.

DOI: 10.4018/978-1-7998-7552-9.ch010

Dr. Velasco-Elizondo finds the topic of software architecture fascinating. This chapter describes some of the significant research, education and, coaching activities she has undertaken during her professional trajectory path to develop knowledge, skills and experiences on this topic. She hopes that this material helps to encourage readers and, particularly, other women to get involved in science, technology and engineering.

This chapter will cover the following sections:

- **Getting it right: software architecture foundations**. This section describes how software architecture foundations are conceived and an example of why preserving them in practice is not always straightforward. It will discuss how to tackle this shortcoming with the proposal of exogenous connectors.
- **Practicing it right: software architecture methods**. In this section, the notion of software architecture lifecycle is introduced. Relevant methods for software architecture development are then briefly discussed within the context of this lifecycle, as well as some limitations related to the difficulty of adopting these methods in practice. Finally, an explanation of why and how technology has to be considered as a first-class design concept in order to tackle one of these limitations will be given.
- **Automating technology selection**. This section presents a software tool, recently developed, which uses information retrieval, natural language processing and sematic web techniques to address the problem of automating NoSQL database technologies search.
- **Software architecture education**. This section describes two educational projects Dr. Velasco-Elizondo has led to promote knowledge and practical experiences on software architecture design and development.
- **Hands on.** Dr. Velasco-Elizondo has had the opportunity to work, as a coach, with practicing software architects and developers helping them to deploy software architecture practices and methods. In this section some of these works will be described.

GETTING IT RIGHT: SOFTWARE ARCHITECTURE FOUNDATIONS

Software architecture has always existed as part of the discipline of Software Engineering. This section describes how software architecture foundations are conceived and gives an example of why preserving them in practice is not always straightforward. The proposed use of exogenous connectors to tackle this shortcoming is also included.

Foundations in Theory

Back when systems were relatively "less complex and small", abstract diagrams were drawn to give stakeholders a better understanding of software designs when describing them. Later, systems went beyond simple algorithms and data structures becoming more complex and larger in size. Therefore, similar in practice to other branches of engineering, more structured diagrams were essential to describe software system designs and communicate regarding aspects such as their main parts and responsibilities, their communication and data model, etc.

Software architecture foundations were built from a high-level model that consists of elements, form, and rationale (Perry, & Wolf, 1992). Elements are first-class constructs representing either computation, data, or connectors. Form is defined in terms of the properties of, and the relationships among, the elements. The rationale provides the underlying basis for the architecture in terms of architectural significant requirements, a.k.a. architectural drivers (Bass, Clements, & Kazman, 2012). In alignment with these foundations, in their seminal book Bass, Clements and Kazman defined software architecture as the set of structures needed for reasoning about the system, which comprises software elements (i.e. computation and data), relations among them (i.e. connectors), and the properties of both (Bass, Clements, & Kazman, 2012). In software architecture design, reasoning is vital as it supports designers in making justifiable decisions (Tang et al., 2008). Thus, all these authors agree that effective reasoning about architectural constructs and their relationships requires a high degree of understanding of these foundations.

Thus, components and connectors have become the basis of many software architecture development approaches. For example, Architecture Description Languages (ADLs) have always defined architectures of software systems in terms of components and connectors connecting them (Shaw & Garlan, 1996). Also, the current version of the Unified Modeling Language (UML) uses connectors to compose components into architectures of software systems.

Foundations in Practice

Despite software architecture foundations and well-specified software architecture design, this nonetheless, tends to erode over time (de Silva & Balasubramaniam, 2012). A number of reasons for design erosion have been identified. One reason relates to the manner in which software architecture is implemented in practice. Software architecture foundations define a component as the principal unit of computation (or data storage) in a system, while connector is the communication mechanism to allow components to interact. However, in existing system implementation approaches, communication originates in components, and connectors are only channels for passing on the control flow to other components. Connectors are mechanism for message passing, which allows components to invoke one another's functionality by method calls (or remote procedure calls), either directly or indirectly, via these channels. Thus components in these approaches mix computation with communication, since in performing their computation they also initiate method calls and manage their returns, via connectors. Consequently, in terms of communication, components implementations are not loosely coupled.

Having components containing very specific communication information hinders their reuse. Software reuse has been defined as "the systematic use of existing software assets to construct new or modified assets" (Mohagheghi & Conradi, 2007). Software reuse is an important topic in Software Engineering as it is widely accepted that it is a means of increasing productivity, saving time, and reducing the cost of software development. Separating computation from communication means that system specific composition details are not in components and therefore components can be reused many times for constructing different systems.

With these shortcomings in mind, this work proposes exogenous connectors to support component composition (Lau & Wang, 2007).

Exogenous Connectors

Exogenous connectors are first-class architectural constructs, which as their name suggests, encapsulate *loci* of communication outside components. By analyzing the control flow required in software systems, a set of useful communication schemes—here defined as specific connector types—has been identified and a catalogue of connectors proposed (Velasco-Elizondo, 2010). The communication schemes are analogous to either control-flow structures that can be found in most programming languages or to behavioral patterns. Behavioral patterns are design solutions that describe common communication schemes among objects. Table 1 contains the connectors in this catalogue and their corresponding descriptions.

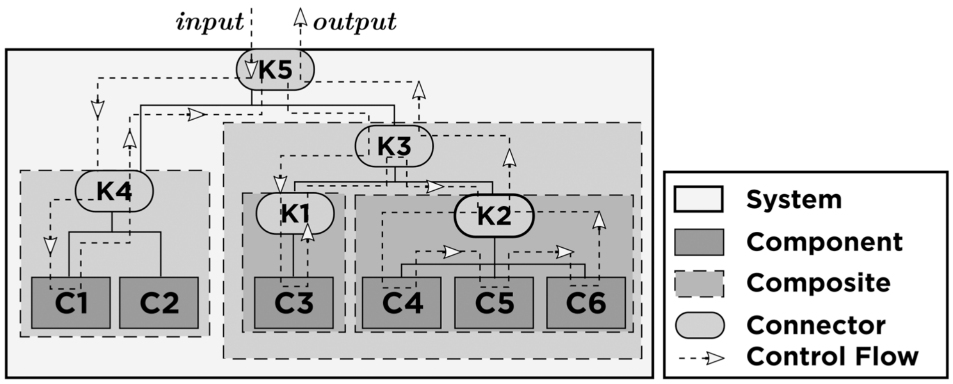
Table 1. A catalogue of exogenous connectors

Name	Description
Sequencer	Provides a composition scheme where the computation in the composed components is executed sequentially one after another.
Pipe	Provides a composition scheme where the computation in the composed components is executed sequentially one after another and the output of an execution is the input of the next one and so forth.
Selector	Provides a composition scheme where the computation in only one of the composed components is executed based on the evaluation of a Boolean expression
Observer	Provides a composition mechanism where once the computation in the ‘‘publisher” component has been performed; the computation in a set of “subscribers” components is executed sequentially.
Chain of responsibility	Provides a composition mechanism where more than one component in a set can handle a request for computation.
Exclusive choice sequencer	Provides a composition mechanism where once the computation in a ‘‘predecessor” component has been performed, the computation of only one component in a set of ‘‘successor” components is executed.
Exclusive choice pipe	A version of the exclusive choice sequencer with internal data communication among the ‘‘predecessor” and in a set of ‘‘successor” components.
Simple merge sequencer	Provides a composition mechanism where once the computation in only one component in a set of ‘‘predecessor” components has been performed, the computation in a set of ‘‘successor” component is executed.
Simple merge pipe	A version of the simple merge sequencer with internal data communication between the ‘‘predecessor” and the ‘‘successor” component.

Figure 1 shows an example of a system architecture with exogenous connectors. It consists of a hierarchy of connectors K1-K5 representing the system's communication, sitting on top of components that provide the computation performed by the system. A connector works as a composition operator that promotes compositionality. That is, when applied to components it yields another component, which is called *composite component*. A composite component can in turn be a subject of further composition. This is illustrated in Figure 1 by the inner dotted boxes.

The control flow in the resulting system is fixed and encapsulated by the corresponding connector structure. A system with exogenous connectors has a set of possible execution paths, but when the system is executed, only one execution path is carried out. The dotted line in Figure 1 shows one possible control flow path for the system represented by the architecture. The execution of the system starts with the connector at the highest level. Thus, the connector K5 invokes the computation in the composite component (containing C1 and C2 components) by calling its top-level connector K4. After that, the connector K5 calls the composite component (containing two composite components: C3 and C4-C6 components). Internally, its top-level connector K3 calls both inner composite components. After this sequence of executions is completed, the control flow is returned back through the hierarchy until reaching the top-level connector K5 that delivers the result of the executions.

Figure 1. An example of a system architecture with exogenous connectors



As it can be implied, exogenous connectors support an algebraic approach to component composition, namely, an approach inspired by algebra where the functionality in components is hierarchically composed into a new one of the same type. The resulting function can be further composed with other functions, yielding a more complex one. More formally, for an arbitrary number of levels (*L*), the connector type hierarchy can be defined in terms of dependent types and polymorphism as follows:

*L*1 º Component ® Result;
*L*2 º *L*1 × … × *L*1 ® Result;
For 2 < *i* £ *n*, *Li* º $L(j_1)$ × … × $L(j_m)$ ® Result, for some *m*;
where j_k Î {1, …, (*i* - 1)} for 1 £ *k* £ *m*,
{ *L*1, *i* = 1
and *L*(*i*) = …
*L*1, *i* = *n*.

Exogenous Connectors in Practice

Since they were proposed, exogenous connectors have been utilized in a variety of systems designs and implementations. Recent examples include IoT end-user smart homes (Arellanes & Lau, 2019) and vehicle control systems (Di Cola et al., 2015). These implementation exercises have all demonstrated the elimination of design erosion caused by not preserving components and connectors as first-class constructs at implementation time. Additionally, the use of exogenous connectors lead to a bottom-up, architecture centered development from reusable constructs. These reusable constructs include both, components and connectors.

Still under investigation is the fact that the use exogenous connectors to design and code system architectures inevitably leads to big hierarchies of connectors. Refactoring techniques for connectors can provide just such a facility.

PRACTICING IT RIGHT: SOFTWARE ARCHITECTURE DEVELOPMENT METHODS

If an important process is difficult, a common approach in many disciplines, including Software Engineering, is to try to systematize that process to ensure predictability, repeatability, and high-quality outcomes. To achieve that, a number of architecture development methods have appeared during the last decade. This section introduces the notion of the software architecture lifecycle. Relevant methods for software architecture development are then briefly discussed within the context of this lifecycle. Also discussed are some limitations related to why these are difficult to adopt in practice. Finally, there is an explanation of why and how technology must be considered as a first-class design concept to tackle one of these limitations.

Software Architecture Lifecycle

Ideally, software architecture development should be carried out within the context of a software architecture lifecycle, which provides a set of stages to follow, with its corresponding activities, to developing it (Cervantes, Velasco-Elizondo, & Kazman, 2013). In general, this lifecycle comprises the following stages, which are depicted in Figure 2:

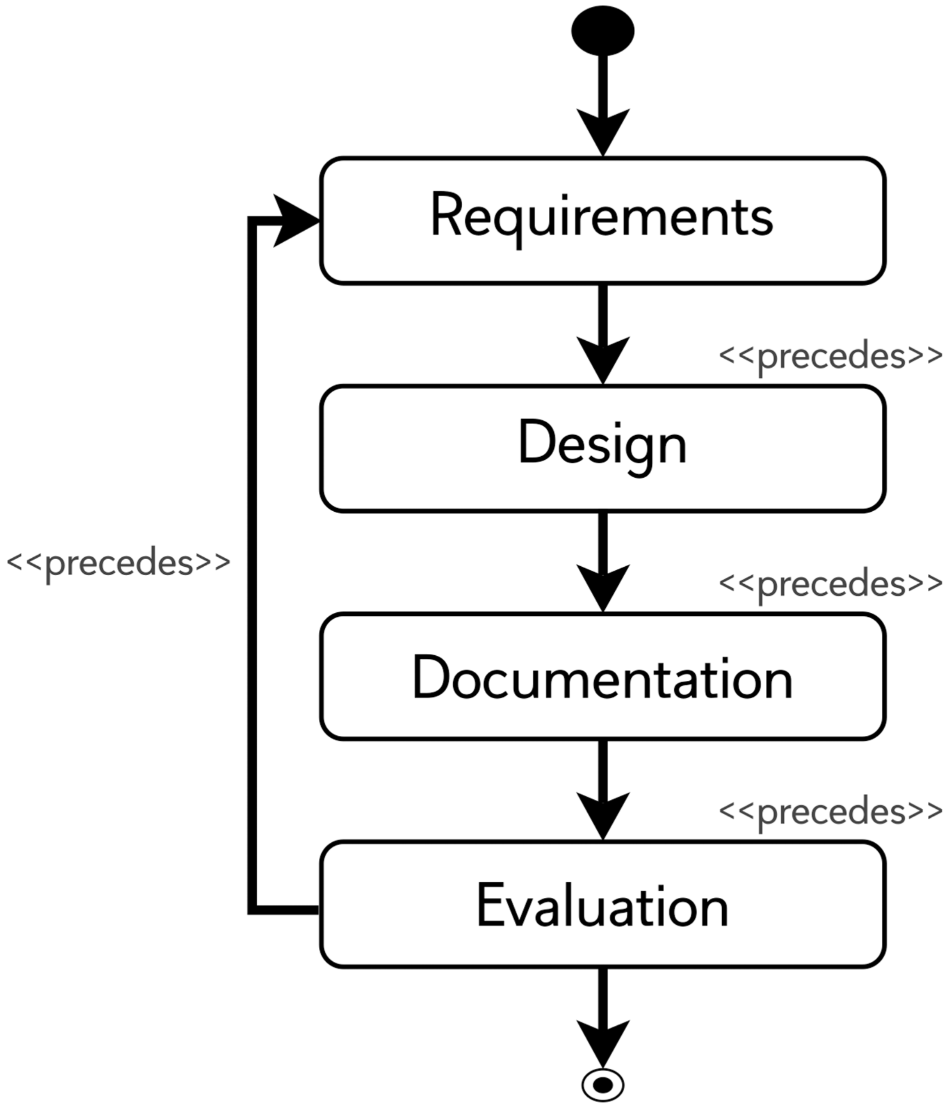
1. **Requirements,** which focuses on identifying and prioritizing architectural requirements.
2. **Design,** which focuses on identifying and selecting the different constructs that compose the architecture and satisfy architectural requirements.
3. **Documentation**, which focuses on creating the documents that describe the different constructs that compose the architecture, in order to communicate it to different system stakeholders.
4. **Evaluation**, which focuses on assessing the software architecture design to determine whether it satisfies the architectural requirements.

METHODS TO SOFTWARE ARCHITECTURE DEVELOPMENT

For the purpose of this work, a *method* is assumed to be a series of detailed steps—with the corresponding instructions for implementation—to accomplish an end. Table 2 lists a set of well-known methods in software architecture development. It is generally considered that most of them exhibit limitations to effectively adopting them in practice, namely: (i) they do not cover all the stages of the architecture lifecycle, (ii) combining them is not always practical and efficient and (iii) they say very little about technology. Next, these limitations are further explained.

As Table 2 shows, most of the methods, apart from ACDM and RUP, do not cover all the stages of the architecture lifecycle. RUP covers them. However, RUP is a general approach to software develop-

Figure 2. Software architecture life cycle



ment and the guidance that it provides with respect to each of the stages in the architecture lifecycle is very general.

The fact that architecture methods generally focus on a particular stage of the lifecycle requires, when doing software architecture development, selecting more than one method. However, the existence of more than one method to choose from for a particular stage of the lifecycle can complicate the selection. Besides, choosing an appropriate combination of the methods is not all that is required. One must also identify how the selected methods should be, practically and efficiently, used together. This is not trivial because different authors have defined these methods "in isolation". They are defined in terms of different activities, artifacts, and terminology. Thus, once an appropriate combination of methods is chosen, the architect must often modify them to avoid mismatches, omissions and/or repetitions.

Finally, architectural design is about applying a set of design decisions to satisfy architectural requirements and design decisions often involve the use of design concepts. *Design concepts* refer to a body of generally accepted architectural design solutions that can be used to create high quality architectural designs. There is a common agreement that design concepts include *reference architectures*, *deployment patterns*, *architectural patterns,* and *tactics*. Design concepts are abstract, and differ from the concepts that software architects use in their day-to-day work, which mostly come from technologies. Software architecture design methods say very little about technology, despite the fact that it is very critical to the success of and architecture design (Hofmeister et al., 2007).

Next section will describes a proposal to tackle this third limitation.

Table 2. Well-known methods for software architecture development

Name		Software architecture lifecycle staged covered	Description
1	Quality Attribute Workshop (QAW) (Barbacci et al., 2003)	Requirements	It is a facilitated scenario-based method for defining the architectural *quality attributes* involving the stakeholders in a workshop. No other requirements types are considered.
2	Attribute Driven Design (ADD) (Cervantes & Kazman, 2016)	Design	It is an iterative method for designing software architectures in which, the designer decomposes the architecture into greater detail, considering *architectural requirements*.
3	The 4+1 view model (Kruchten, 1995)	Documentation	It is a model used for documenting software architectures, considering multiple, concurrent views from the viewpoint of different stakeholders. The four views of the model are *logical, development, process and physical*. The *use case* view serves as the 'plus one' view.
4	Views and Beyond (V&B) (Clements et al., 2010)	Documentation	It is a method and a collection of techniques for documenting software architectures, considering the following view types giving a different perspective of the structure of the system. The considered view types are: *module, component-and-connector* and the *allocation*.
7	The Software Architecture Analysis Method (SAAM) (Kazman et al., 1996)	Requirements, Evaluation.	It is a method for evaluating software architecture designs to point out the places where it fails to meet its *quality attribute requirements* and show obvious alternative designs that would work better. No other requirements types are considered.
8	Architecture Tradeoff Analysis Method (ATAM) (Clements, Kazman, & Klein, 2002)	Requirements, Evaluation	It is a method for evaluating software architecture designs relative to *quality attribute requirements* in order to expose architectural risks that potentially inhibit the achievement of an organization's business goals. No other requirements types are considered.
9	ARID (Clements et al., 2008)	Evaluation	It is a lightweight method for evaluating partial software architecture designs in its early stages. It combines aspects from ATAM and SAAM in a more tactical level.
9	Architecture Centric Design Method (ACDM) (Lattanze, 2009).	Requirements, Design, Documentation, Evaluation.	It is an iterative approach to software development; its feature is an emphasis on architecture. It say very little about technology.
10	RUP (Kroll, Kruchten, & Booch, 2003)	Requirements, Design, Documentation, Evaluation.	It is an iterative approach to software development. It say very little about technology.

Technology as a First-Class Design Concept and its use in Architectural Design Methods

In (Cervantes, Velasco-Elizondo, & Kazman, 2013) the idea of considering technologies as a first-class design concept was explicitly stated for the first time. A demonstration of how to use in ADD, a well-known method to architecture design, was also presented. Next, a recent case to demonstrate it is used.

Consider the development of an online system to apply for a place at a primary school. Via this system parents/tutors will be able to perform function such as creating an account, finding a school, filling out an application, and submitting an application. According to the software architecture lifecycle depicted in Figure 2, before starting architecture design the development team should have the architectural requirements identified. Architectural requirements include: (*i*) user requirements, which are the functions the users can perform via the software system; (*ii*) quality attributes, which are properties of software systems that are of interest in terms of it being developed or operated, e.g. maintainability, testability, performance, security; and (*iii*) constraints, which are already taken (design) decisions, i.e. mandated technologies, laws and standards that must be complied with.

These architectural requirements are outlined, in general terms, in Table 3.

Table 3. Architectural requirements for a system to apply for a place at a primary school

Architecture Requirements	Description
User requirements	UR1 - Find a school UR2 - Fill out application
Quality attributes	QA1 - Performance. A user performs a school search during peak load; the system processes the query in under 10 seconds. QA2 - Availability. A random event causes a failure to the system during normal operation; the system must be providing services again within 30 seconds of the failure. QA3 - Security. An unknown user from a remote location attempts to access the system during normal operation; the system blocks the access to this user the 99.99% of the times.
Constraints	C1- The system must run in Internet Explorer n+, Firefox 3+, Chrome 6+, and Safari 5+. C2- The system must use information stored in a MariaDB 10+ database.

Architectural patterns are proven good architecture design structures, which can be reused, e.g. layers, pipes and filters, MVC. Tactics are a proven good means to achieve a single quality attribute requirement, e.g. replication, concurrency, authentication. Considering the former, let's suppose that the architect used the ADD method, which defines the below steps, and also used architectural patterns, tactics and technologies.

1. Confirm that information about requirements is sufficient.
2. Choose a system element to decompose.
3. Identify candidate architectural drivers.
4. Choose a design concept that satisfies the drivers.
5. Instantiate architectural elements and allocate responsibilities.
6. Define interfaces for the instantiated elements.
7. Verify and refine requirements and make them constraints for instantiated elements.

8. Repeat these steps for the next element.

Table 4 shows three design iterations of how the design process can be performed when technologies are first-class design concepts in ADD.

Table 4. Three design iterations using technologies as first-class design concepts in ADD

	Drivers (ADD step 3)	**Element to decompose (ADD step 2)**	**Design decisions (ADD step 4)**
1	User requirements Constraints	The system as a whole	Layers pattern
2	QA1 - Performance QA3 – Security	Presentation layer Business logic layer	Increase resource efficiency performance tactic – Cloudfare Detect intrusion security tactic – Cloudfare Detect service denial security tactic - Cloudfare
3	QA2 - Availability		Monitor availability tactic - Zabbix Ping/Eco availability tactic - Zabbix

The first iteration was intended to decompose the entire system (ADD step 2) and the primary concern was to define the overall system structure and allocate responsibilities to elements that the development team could develop independently. The architect decided to create a layered system using the Layers Architecture Pattern; which is a n-tiered pattern where the components are organized in horizontal layers (Clements et al., 2011). Although the layered architecture pattern does not specify the number and types of layers that must exist in the pattern, most layered architectures consist of four standard layers: presentation, business, persistence, and database.

In iteration 2, the architect decided to use several design concepts—say tactics and technologies—to address performance and security quality attributes. The *increase resource efficiency* tactic suggests looking at the ways algorithms use a resource to process an event and optimizing them to reduce their processing time. The *detect intrusion* tactic suggests monitoring traffic coming into a system against known patterns of attacks. The *detect service denial* is a tactic for detecting a special type of attack in which the system become overwhelmed with a flood of unauthorized access attempts and cannot do anything else (Sangwan, 2014). Cloudfare[1] is a technology that implements all these tactics.

With regard to performance, Cloudfare can automatically determine which resources are static, such as HTML pages, javascript files, stylesheets, images, and videos, and help cache this content at the network edge, which improves website performance. In terms of security, Cloudfare can stop attacks directed at a website by providing security from malicious activity, including denial-of-service attacks, malicious bots, and other nefarious intrusions.

In iteration 3, the architect decided to use the following tactics and technologies to address the availability quality attribute. Zabbix[2] is a monitoring software tool for diverse IT components, including networks, servers, virtual machines, and cloud services. Zabbix provides monitoring metrics, among others network utilization, CPU load and disk space consumption. Zabbix implements the ping/echo and monitor security tactics. The *monitor* tactic uses a process that continuously monitors the status of different elements of the system (Sangwan, 2014). This tactic often uses the ping/echo tactic to periodically ping an element for its status and can receive an echo indicating it was operating normally (Sangwan, 2014).

In this example, design continues along other iterations, making design decisions ranging from selecting architecture patterns and tactics to implementation options provided by technologies.

Important Considerations

As reported (Cervantes, Velasco-Elizondo, & Kazman, 2013) this approach has been applied successfully to several projects at a large company in Mexico City that develops software for government and private customers. More recently, it has been applied for refactoring existing systems and developing new ones at the Secretary of Education of Zacatecas state. For example, the online system to apply for a place at a primary school, described above, was unable to consistently handle 12,000 users at once –which is the expected load in peak time, and exhibited security breaches last time that it was in production. Using this approach the architecture of the system was redesigned and its implementation ran without incidents in 2020.

The proposed approach has been observed to be more realistic in the sense that design activities produce architectures that can be executed—and therefore tested—before they are implemented. However, it is important to consider that technology selection cannot be arbitrary, as it depends on factors such as: the development team's level of knowledge of each technology, the technology type (i.e. commercial or open source), the technology's maturity and level of support from its community, etc.

AUTOMATING TECNOLOGY SELECTION

When software architects design software architectures, they make the decision to promote a set of architectural requirements and aim for an optimal choice in the necessary trade-offs for a particular context. Assuming that software architecture can be specified in a formal manner, and its design can be done systematically via realistic design methods, a basic research question arises: could it be possible to automate software architecture development at least to some degree? The automation of software architecture design would be beneficial not only for increasing the productivity of a software architect, but also in improving the quality of the resulting architectures.

This section presents a recent tool developed, which uses natural language processing and semantic web techniques to the problem of automating technology searches (Esparza, 2019).

WHAT TECHNOLOGY SELECTION REALLY INVOLVES

The process of developing an architecture design starts by considering architectural requirements. Architectural requirements include (Cervantes et al., 2016): (*i*) user requirements, (*ii*) quality and (*iii*) constraints. Among these requirements, quality attributes and constraints are those that shape the architecture the most significantly (Cervantes et al., 2016). The process of creating an architecture design then continues by linking architecturally significant requirements to design decisions; this involves the use of design concepts that satisfy requirements. An example might be using the broker architectural pattern as a design concept to satisfy a scalability quality attribute. Next, these design decisions should be systematically linked to the implementation alternatives through the use of specific technologies, for example, selecting RabbitMQ[3] to implement the broker architectural pattern.

Table 5. Well-known tools to support technology search

	Repository			Query					Results																									
									Basic description				Third party links						Technical details											Quality Attributes				
	Frequent updates	Automatic filling	Multiple sources	License type	Operative System Platform	Product's name	Categories / Subcategories	Compatible technologies	Description	License type / Price	Developer / Official page	Version	Companies using the products	Use cases	Links to other resources	Recommendations of other products	User's comments	Integration with other platforms	GUI images	Code examples	Information's statistics	Score	Product's popularity charts	Size	Platform	Implementation languages	Compatible technologies	APIs	Features and design	Performance	Usability	Availability	Scalability	Security
Softonic	S	S	S	S	S	S	S	-	S	S	S	S	-	-	-	S	S	-	S	-	-	S	-	S	S	-	-	-	-	-	S	-	-	-
Softpedia	S	-	S	S	S	S	S	-	S	S	S	S	-	-	-	S	-	-	S	-	S	S	-	S	S	-	-	-	-	-	-	-	-	-
Wolfram Alpha	S	P	-	S	S	S	-	-	S	S	S	-	-	-	-	-	-	-	-	-	S	-	-	-	-	S	-	-	-	-	-	-	-	-
AlternativeTo	S	-	-	-	S	S	S	-	S	S	S	-	-	-	-	S	S	-	S	-	S	S	-	-	S	-	-	-	-	-	-	-	-	-
StackShare	S	P	S	-	-	S	S	-	S	-	S	-	S	-	-	S	S	S	-	-	-	S	-	-	-	-	-	-	-	-	-	-	-	-
G2 Crowd	S	-	S	-	-	S	S	-	S	S	S	-	-	-	-	S	S	-	S	-	-	S	-	-	-	-	-	-	-	-	-	S	S	S
StackOverflow	S	-	-	-	-	S	-	-	S	S	S	S	-	P	S	S	-	-	-	P	S	S	-	-	S	S	S	S	S	-	-	-	-	-
DBEngine	S	-	S	-	-	S	S	-	S	S	S	S	-	P	P	-	-	-	-	-	-	S	S	-	S	S	S	S	S	-	-	-	-	-
S = supported; P = partially supported; - = non supported;																																		

Technology selection involves five main factors to consider: (*i*) deciding which technologies are available to realize the decisions made; (*ii*) determining whether the available tools to support this technology choice (e.g. IDEs, simulators, testing tools, and so on) are adequate for development to proceed; (*iii*) determining the extent of internal familiarity and external support available for the technology (e.g. courses, tutorials, examples and so on); (*iv*) determining the side effects of choosing a technology, such as a required coordination model or constrained resources; and (v) determining whether a new technology is compatible with the existing stack (Cervantes et al., 2013).

As a result, technology selection is a multicriteria decision-making problem. Furthermore, given the steadily growing number of technologies, deciding which options are available to implement the selected design decisions requires being aware of the range of existing. This can be particularly challenging for inexperienced software architects.

The problem of selecting a software technology can be defined in terms of two sub-problems as follows:

1. **Technology Search**: Finding technologies that allow the implementation of specific design decisions.
2. **Technology Selection**: Selecting a technology to use from among those available.

The following section elaborates on aspects of the first sub-problem: technology search.

EXISTING SUPPORT FOR TECHNOLOGY SEARCH

Table 5 compares a set of well-known tools to support technology search. This comparison is next briefly explained.

The "Repository" section describes the way in which the tools structure and maintain information about technology products in a repository. "Frequent Update" refers to the frequency of content updates. "Automatic filling" refers to whether the information for each technology is added manually or automatically. "Multiple sources" indicates whether the technology information is obtained from more than one source.

The "Query" section groups aspects related to the facility of formulating a query considering factors such as "Type of License", "Operating System Platform", "Name of the (Technology) Product", "Category", and "Compatible Technologies".

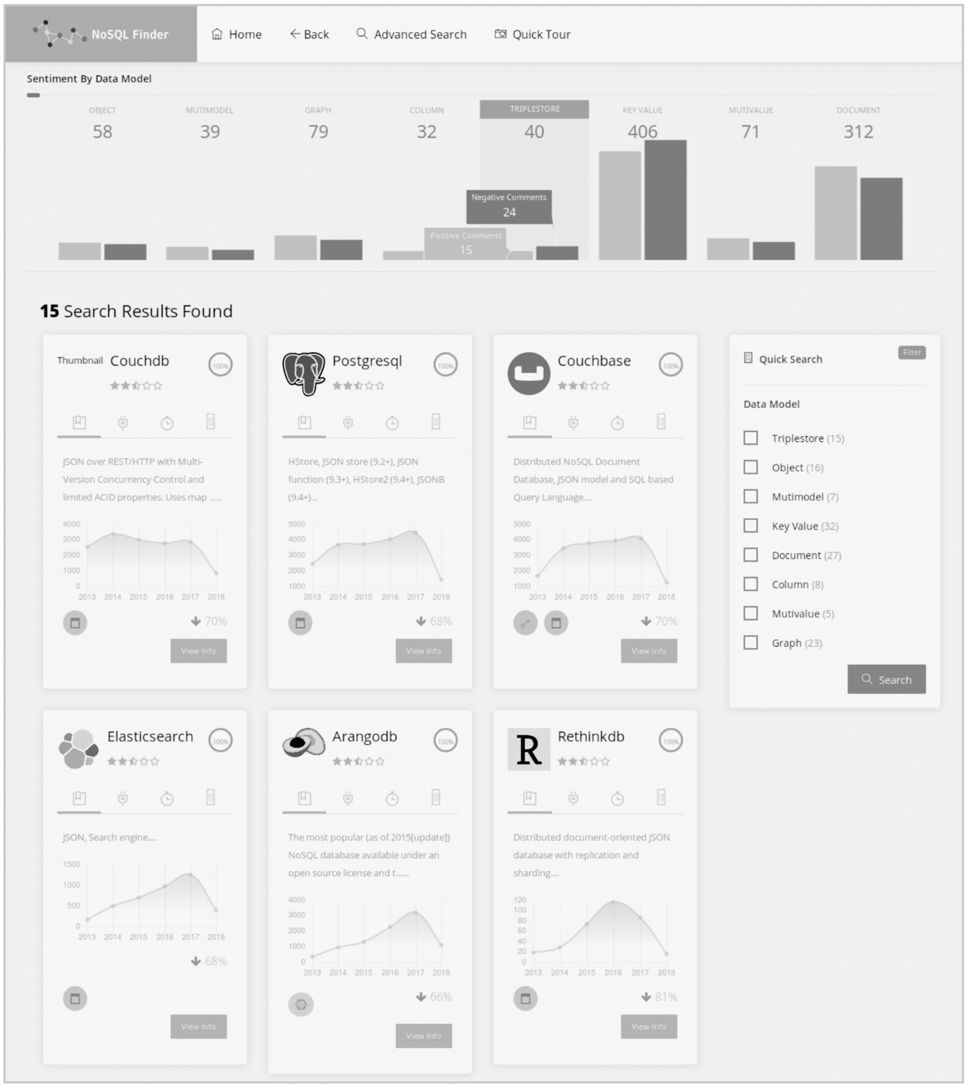
The "Results" section is about the aspects related to the query's results. The aspects are grouped into four categories. The first one, "Basic description" includes general information about the technology. The second category, "Third party links" refers to general information elements from third parties that complement the basic description. The third section, "Technical details", contains technical information elements. Finally, the "Quality attributes" section includes the quality attributes of the technology.

Three letters are used in Table 5 to describe the level of support for information in the sections previously described, namely S = supported; P = partially supported; - = non supported. It is therefore possible to state the following findings.

The support for formulating queries is limited. Most tools support queries based on the name of the technology and the category to which it belongs. The remainder of the factors are not well supported in the reviewed tools.

The resulting information is incomplete. Most of the tools cover all the requirements in the "Basic description" section. However, the necessary elements of the "Links to third parties" section are included

Figure 3. A screenshot of the result of a query in NoSQLFinder

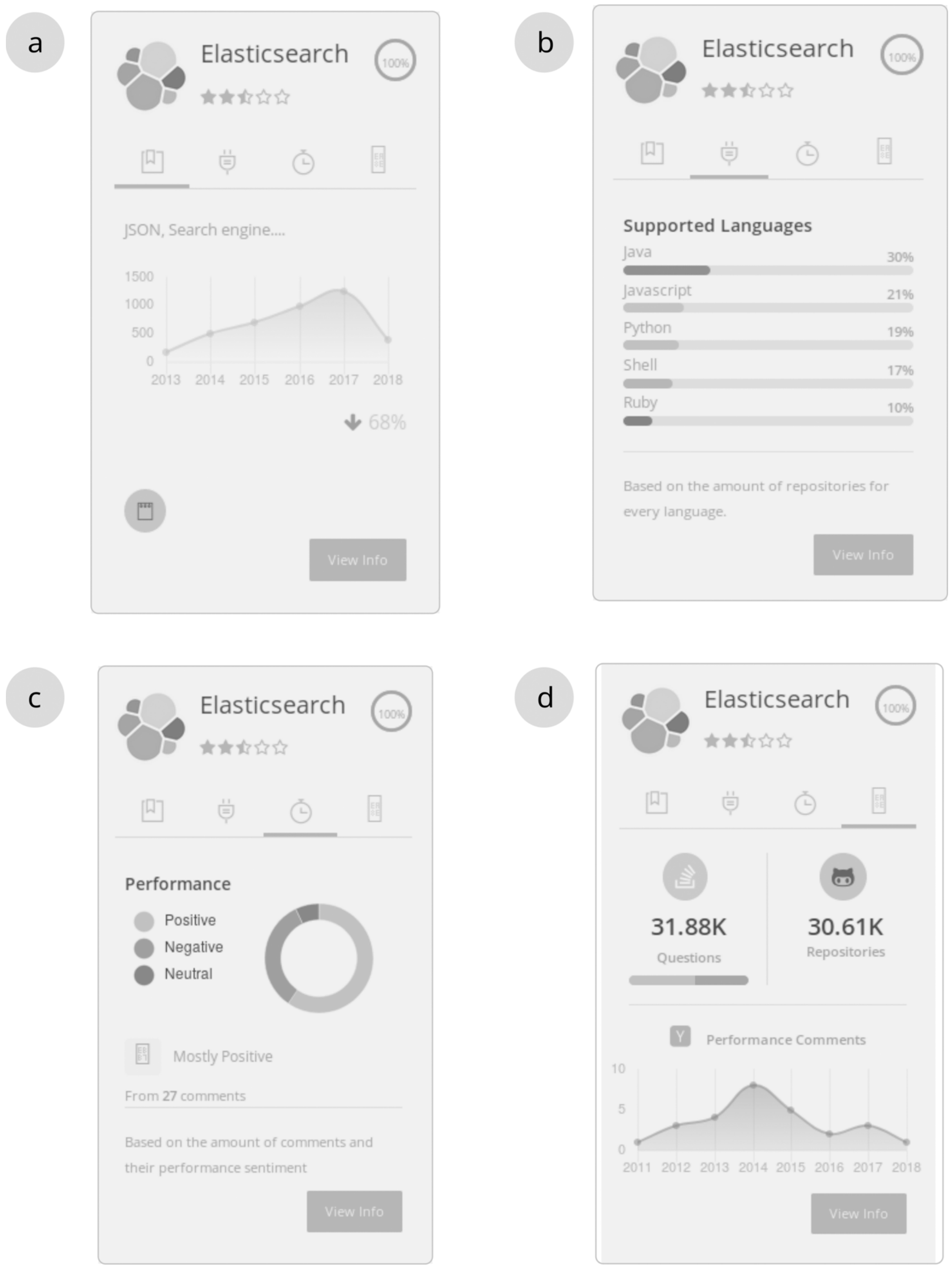


in very few tools. With regards to the “Technical details” section, only two of the tools support most of the elements. Finally, information about quality attributes is limited.

NOSQLFINDER

NoSQLFinder is a functional prototype tool that helps architects to perform technology searches. NoSQLFinder uses information from various information sources, such as AlternativeTo, StackOverflow, DBpedia. Due to scope, in this prototype the focus was on supporting the search of NoSQL databases while taking into account many of the aspects described in Table 5. The design and implementation of

Figure 4. A screenshot of a database card in NoSQLFinder



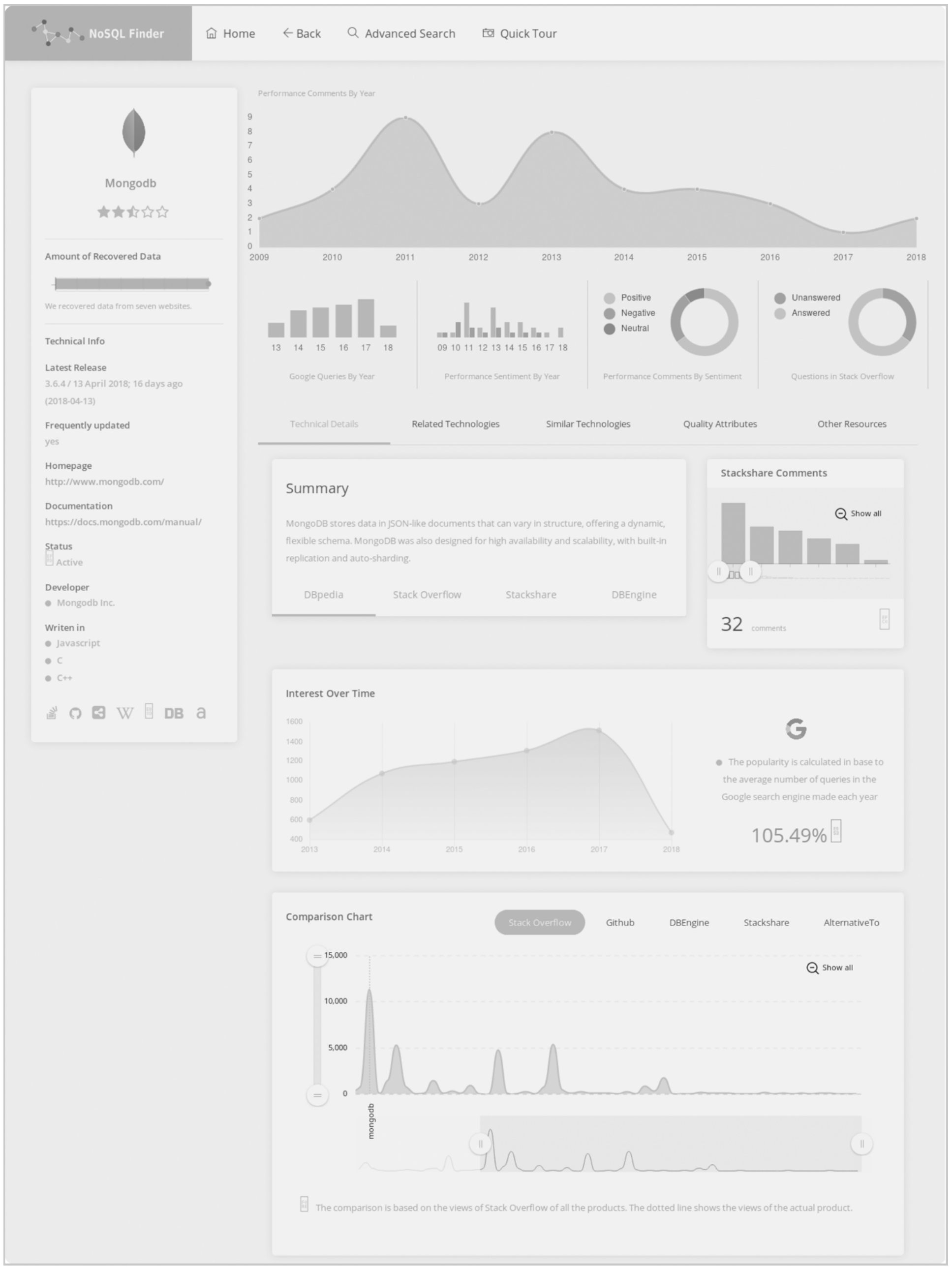
Based on the amount of repositories for every language.

Based on the amount of comments and their performance sentiment

NoSQLFinder involves the use of data extraction, linked data, semantic Web and sentiment analysis techniques. NoSQLFinder is reachable via http://dataalchemy.xyz

Figure 3 shows a screenshot of the result of a query in NoSQLFinder. Each technology is shown as a card containing the following information sections: general popularity, supported programming languages, feedback from other users about its performance, and popularity among sites frequently visited by developers like StackOverflow and Github, see Figure 4 a, b, c, and d respectively.

Figure 5. A screenshot of part of the dashboard with information for a database in NoSQLFinder



Mongodb

Amount of Recovered Data

We recovered data from seven websites.

Technical Info

Latest Release
3.6.4 / 13 April 2018; 16 days ago
(2018-04-13)

Frequently updated
yes

Homepage
http://www.mongodb.com/

Documentation
https://docs.mongodb.com/manual/

Status
Active

Developer
- Mongodb Inc.

Writen in
- Javascript
- C
- C++

Summary

MongoDB stores data in JSON-like documents that can vary in structure, offering a dynamic, flexible schema. MongoDB was also designed for high availability and scalability, with built-in replication and auto-sharding.

DBpedia | Stack Overflow | Stackshare | DBEngine

32 comments

The popularity is calculated in base to the average number of queries in the Google search engine made each year

105.49%

The comparison is based on the views of Stack Overflow of all the products. The dotted line shows the views of the actual product.

When a user selects a specific card, all the information about the corresponding database is displayed on a dashboard. The dashboard is organized into four sections that represent the categories of information elements in Table 5, namely “Basic description”, “Third party links”, “Technical details”, “Quality attributes”. Figure 5 shows a screenshot of part of this dashboard.

FUTURE WORK

The design and implementation of NoSQLFinder has been a great learning experience. Firstly, it was an opportunity to learn various semantic Web techniques for collecting and integrating data. Secondly, various sentiment analysis techniques in order to determine and measure the degree of user satisfaction with respect to a quality attribute from text in natural language, were learned.

We plan to generate a more robust version of the tool for searching technologies. To this end, it was planned to explore the following lines of future work: support for more quality attributes, the use of machine-learning techniques to better evaluate what people say about technologies' quality attribute, and the inclusion of other types of technologies besides NoSQL databases.

SOFTWARE ARCHITECTURE EDUCATION

Software architecture development requires not only deep technical experience, but also understanding of the body of knowledge concerning software architecture, as expressed in terms of design concepts and software architecture development methods. Mastering software architecture development requires combining both experience and knowledge to ensure high quality outcomes when developing architecture.

This section describes two educational projects created to promote software architecture knowledge and experiences.

Summer School on Software Engineering

Dr. Velasco-Elizondo is an experienced teacher. Since 2012, the she has taught software architecture to students in under- and post-graduate programs in Software Engineering and has contributed to the definition of software architecture courses in these academic programs. Dr. Velasco-Elizondo put this experience to further use in 2012, when she founded one of the few Summer Schools on Software Engineering in Mexico, which she still run each summer. In each edition of the Summer School the topic of software architecture has been present in some form.

In the Summer Schools on Software Engineering, participating students represent private and public institutions located in various states of Mexico. Attendees have lauded the School as "an excellent forum to learn exciting topics that are related to way software systems are designed and implemented". They also welcomed the opportunity to "meet with colleagues and share experiences".

In the 2019 edition of the School three courses were offered: An introduction to NOSQL databases, Agile Techniques for Risks Management and Colored Backlogs with Scrum. Scrum is an agile project management framework that can be used for software development projects with the goal of incrementally delivering a software product by supplying working requirements every 2-4 weeks. In Scrum, a backlog is a list of tasks and/or requirements that need to be completed. In the third course it was learned and practiced a technique for ensuring that architectural requirements get addressed and prioritized applying the concept of color in a Scrum backlog as follows:

- Green – functional user requirements.
- Yellow – Architectural infrastructure to support the quality requirements.
- Red – Defects that are identified and need to be addressed.

- Black – Technical debt that builds up as the product is built and key decisions are deferred or poor work done. Technical debt is a term in software development that reflects the implied cost of additional rework caused by choosing a fast and dirty solution instead of using a better approach that would take longer.

Colored Backlogs is a technique proposed by Dr. Philippe Kruchten.

Software Architecture Book

Within the context of Software Engineering, the development of software architecture deals with structuring a software system to satisfy architectural requirements, namely, user functional requirements, quality attributes and constraints. This technological moment requires people to interact with many software systems that increasingly have more complex quality attribute requirements such as performance and availability. This increasing complexity is why software architecture development is an important topic.

Taking into account their experience as specialized teachers of Software Engineering, as well as the years of collaboration and consulting with the software development industry, Dr. Velasco-Elizondo and their colleagues identified the lack of an introductory book of software architecture in Spanish. They therefore decided to write this book, which is titled “Software Arquitectura: Conceptos y Ciclo de Desarrollo”, published in 2015 (Cervantes, Velasco-Elizondo & Castro, 2016).

The book places an important emphasis on software architecture foundations but also provides practical examples that allow these foundations to be related to real practice. Practitioners, and master and undergraduate students interested in design and development of software systems are the main prospective readers of this book. The book has seven chapters, covering the following topics.

Chapter 1 presents the introduction to the topic of software architecture by introducing the basic concepts and the notion of a software architecture development lifecycle defining these stages: requirements, design, documentation, and evaluation.

Chapter 2 discusses the requirements stage. To provide an initial context, the term “requirement” as used in the field of Software Engineering is described. Then the focus is on the specific aspects of the requirements stage, including fundamental concepts, activities, and a review of well-known methods to systematically perform the activities necessary to this stage.

Chapter 3 is about the design stage. This chapter begins by describing the concept and levels of design in general and then lays out the architecture design process. Next is a discussion of the design principles and concepts that are used as part of the architecture design process. A review of some well-known methods to design architectures is then presented.

Chapter 4 explores the documentation stage, initially establishing the importance of documenting software architectures. There is a particular focus on how to create documentation to effectively communicate a software architecture design to different stakeholders. Initially, the meaning of sufficient documentation is discussed in a general way, followed by an exploration of what this concept means in software architecture and why documentation is important. Relevant concepts, notations, as well as methods that can be used to support activities at this stage are also included. The chapter concludes with a list of useful recommendations for generating good quality architectural documentation.

Chapter 5 is about the evaluation stage. Once a software architecture is designed and documented, it is possible to start constructing the system. However, before doing so, it is vital to ensure that the software architecture is correct and satisfies the architectural requirements. Evaluating software architectures enables the detection of breaches and risks, which can otherwise cause of delays or serious problems when implementing software architectures. This chapter begins by discussing the concept of evaluation in general terms, and then the moves to what this means for software architectures. The chapter also includes the description of some relevant methods used to evaluate software architectures.

Chapter 6 discusses the influence of software architecture in the implementation of a software system. The work of the architect is often considered to be finished when an architecture design has been defined, evaluated, and satisfactorily meets the architectural requirements. However, the lack of follow-up during the implementation of the system is often the cause when the architecture's design is not properly implemented.

Chapter 7 describes how to perform the stages of the software architecture development lifecycle in projects that use agile methods. Within the context of Software Engineering, agile methods are a set of widely used lightweight methods currently used to support systems development relying on self-organizing, interdisciplinary, and highly flexible teams. In this chapter the content is described around Scrum, a very popular agile method today.

Importantly, the topics in each chapter are explained via a case study, which is included in full as an appendix of the book. Furthermore, the book provides a set of questions and as well as references to other information sources so that readers can reinforce and build on their knowledge of the material presented in this book.

Hands on

Since 2008 Dr. Velasco-Elizondo has had the chance to work, as a coach, with practicing software architects and developers helping them to deploy software architecture practices and methods. Dr. Velasco-Elizondo strongly believes in lean and agile principles and she had also being able to apply them within the context of software architecture design.

In this section some of these experiences are described.

SmartFlow MVP

Internet of Things (IoT) is the technology enabling the communication of a variety of devices via the Internet in order to exchange data. IoT is composed of network of sensors, actuators, and devices that allow the development of new systems and services to monitor and control process or services. Dr. Velasco-Elizondo had the opportunity to contribute to the design and implementation of a minimum viable product (MVP) of a system of this kind for the mining industry domain. Specifically, she served as the Scrum Master of the Scrum Team developing the SmartFlow MVP. A minimum viable product (MVP) is a version of a product with just enough features to satisfy early customers and provide feedback for future product development.

SmartFlow MVP was a system that used RFID tags to support real-time exchange of data with a series of RFID readers through radio waves. RFID tags are a type of tracking devices that use smart barcodes in order to identify items. RFID is short for radio frequency identification. Any entity of interest circulating in a mine tunnel, i.e. miner, vehicle, had a RFID tag attached. An RFID transmitted its location and other relevant data to RFID readers. RFID readers sent the received data to a software system that processed and displayed it for different applications, i.e. access control, collision control, and evacuation control.

The use of Scrum for the development of the SmartFlow MVP was favored. Considering the agile nature of Scrum, it was considered that the Scrum Team were much more likely to get the MVP shipped on time and on budget. In this project Dr. Velasco-Elizondo and the team had the opportunity to deploy a set of practices to make agile architecting feasible. The following practices were integrated in a customized version of Scrum for this project: (1) visualizing requirements using the colored backlog technique (2) developing an architectural spike in a Sprint 0 (3) visualizing technical debt using the colored backlog techniques and (4) keeping the architecture documentation updated. An architecture spike is an experiment to reduce risk and to improve the technical understanding of the system to build and the technology choices to build it.

Visualizing architectural requirements using a colored backlog allowed the Scrum Team to better plan its releases and be aware of architectural requirements. Developing an architectural spike helped the Scrum Team to early determine whether the defined architecture was proper to support the architectural requirements; particularly quality attributes. Visualizing technical debt using a colored backlog helped the Scrum Team not only to be aware of it, but also to prioritize it in sprint planning just like normal requirements. Finally, as Simon Brown advises the focus was given to describing in the architecture documentation only the aspects that Scrum Team could not get from the code (Brown, 2014).

Refactoring an School Enrollment System

Working as a Principal Technical Assistant at the Secretary of Education of Zacatecas state, has giving Dr. Velasco-Elizondo the chance to contribute in the refactoring work a development team performed to pay technical debt of a school enrollment system. Parents and tutors to apply for a place at a primary school in Zacatecas state using this system.

In short, refactoring is a technical term in Software Engineering to refer of the activity of modifying the internal structure of code of a system without changing its behavior. Refactoring is important to maintain the long-term quality of a system code. If there is technical debt and developers do not perform some refactoring on a regular basis, technical debt will creep into the system. In this project it was made sure to have good reasons for doing it. In this case id was necessary to improve performance, scalability, security and test coverage. These quality requirements were understood as architectural and therefore, it leads to the process to select design concepts to realize them; and then select technologies to implement them.

Besides following a systematic and quality attribute driven approach to refactoring and learning new technologies, another good thing this project generated was the design and delivering of a course that helped the development team "to link theory to practice". That is, to better understand the undelaying design decisions and design concepts and the technology choices to implement them.

CONCLUSION

It is common to hear discussions of academic–practitioner relationships that focus on the gap between academics and practitioners, between rigor and applicability, between theory and practice or similar terms. The gap between academics and practitioners has been of concern for decades. For these reasons Dr. Velasco-Elizondo is very satisfied with the opportunities a have taken to develop what she wants to become: a software architecture academic-practitioner. Thus, in this chapter the author has described some significant research, education and coaching activities the author has performed during her professional career to develop knowledge, skills and experiences related to this desire.

Often researchers tend think that they already have enough to do with our research. In her experience, “working outside an office” helps not only to develop business awareness but also to strengthen soft skills, networking, self-awareness, and practical ways to apply research and demonstrate its potential impact. The connected world of today means that software is everywhere; thus, software architecture design is everywhere too. How can a software architect come up with the simplest possible design to make sure a software system fits into the long-term goals? Well, it demands a potent blend of thoughtful design with constant experimentation. This is exactly how people learn to tackle complexity and therefore, everything else in life.

Dr. Velasco-Elizondo hopes the material communicated in this chapter will inspire readers, specifically female readers. Do you know that women did very first jobs in Software Engineering history? Ada Lovelace was the first computer programmer in history. Grace Hopper developed the first computer compiler and made programming easier for all of us. Margaret Hamilton developed the on-board flight software for the lunar lander of the Apollo 11 mission. How, then, did was developed the myth that girls do not belong in Software Engineering? Although the effort to involve women in these fields has been rising in recent years, stereotypes still exist and are preventing women to have equal opportunities to access to education and work in STEM. These stereotypes are flat wrong. It does not make sense to think that talent, ideas, efforts should be wasted simply because they come from a woman, rather than a man. In most software development projects, different decisions are taken. (Bhat et al., 2020) provides a getting-started guide for prospective researchers who are entering the investigation phase of research on architectural decision-making.

Keep always an unfulfilled desire that is waiting to be fulfilled, when one is not hungry enough, one never will find the motivation to do anything. Never be satisfied, be curious to learn more and more. Be willing to keep trying the things people say cannot be done. In other words, and as many important people have said, Dr. Velasco-Elizondo encourages you to “stay hungry, stay foolish”.

REFERENCES

Arellanes, D., & Lau, K. K. (2019). Workflow Variability for Autonomic IoT Systems. In *Proceedings of the International Conference on Autonomic Computing*, (pp. 24-30). Umea, Sweden: IEEE.

Barbacci, M., Ellison, R. J., Lattanze, A. J., Stafford, J. A., Weinstock, C. B., & Wood, W. G. (2003). *Quality Attribute Workshops (QAWs), Third Edition.* Technical Report CMU/SEI-2003-TR-016, Software Engineering Institute, Carnegie Mellon University.

Bass, L., Clements, P. C., & Kazman, R. (2012). *Software Architecture in Practice*. Addison-Wesley Professional.

Bhat, M., Shumaiev, K., Hohenstein, U., Biesdorf, A., & Matthes, F. (2020). The Evolution of Architectural Decision Making as a Key Focus Area of Software Architecture Research: A Semi-Systematic Literature Study. In *Proceedings of the IEEE International Conference on Software Architecture*, (pp. 69-80). IEEE.

Brown, S. (2014). *Software Architecture for Developers - Technical leadership by coding, coaching, collaboration, architecture sketching and just enough up front design*. Leanpub.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture Volume 1: A System of Patterns.* Wiley Publishing.

Cervantes, H., & Kazman, R. (2016). *Designing Software Architectures: A Practical Approach*. Addison-Wesley Professional.

Cervantes, H., Velasco-Elizondo, P., & Castro, L. (2016). *Arquitectura de software: conceptos y ciclo de desarrollo*. Cengage Learning.

Cervantes, H., Velasco-Elizondo, P., & Kazman, R. (2013). A Principled Way to Use Frameworks in Architecture Design. *IEEE Software*, *30*(2), 46–53. doi:10.1109/MS.2012.175

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Reed, L., & Nord, R. (2011). *Documenting Software Architectures: Views and Beyond.* Addison-Wesley Professional.

Clements, P., Kazman, R., & Klein, M. (2002). *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional.

Clements, P., Kazman, R., & Klein, M. (2008). *Evaluating Software Architectures Methods and Case Studies*. Addison-Wesley.

de Silva, L., & Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software*, *85*(1), 132–151. doi:10.1016/j.jss.2011.07.036

Di Cola, S., Lau, K.-K., Tran, C., & Qian, C. (2015). An MDE tool for defining software product families with explicit variation points. In *Proceedings of the International Conference on Software Product Line*, (pp. 355–360). New York, NY: Association for Computing Machinery. 10.1145/2791060.2791090

Esparza, M. (2019). *NoSQLFinder: un buscador de bases de datos NoSQL basado en técnicas de web semántica y análisis de sentimientos* (BSc Thesis).

Hofmeister, C., Kruchten, P. B., Nord, R., Obbink, H., Ran, A., & America, P. (2007). A general model of software architecture design derived from five industrial approaches. *Journal of Systems and Software*, *80*(1), 106–126. doi:10.1016/j.jss.2006.05.024

Kazman, R., Abowd, G., Bass, L., & Clements, P. (1996). Scenario-Based Analysis of Software Architecture. *IEEE Software*, *13*(6), 47–55. doi:10.1109/52.542294

Kroll, P., Kruchten, P. B., & Booch, G. (2003). The rational unified process made easy. Addison-Wesley Professional.

Kruchten, P. B. (1995). The 4+1 View Model of Architecture. *IEEE Software*, *6*(12), 42–50. doi:10.1109/52.469759

Lattanze, A. J. (2009). *Architecting Software Intensive Systems: A practitioners guide*. CRC Press.

Lau, K., & Wang, Z. (2007). Software Component Models. *IEEE Transactions on Software Engineering*, *33*(10), 709–724. doi:10.1109/TSE.2007.70726

Mohagheghi, P., & Conradi, R. (2007). Quality, productivity and economic benefits of software reuse: A review of industrial studies. *Empirical Software Engineering*, *12*(5), 471–516. doi:10.100710664-007-9040-x

Perry, D. E., & Wolf, A. L. (1992). Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, *17*(4), 40–52. doi:10.1145/141874.141884

Rajkumar, R., Lee, I., Sha, L., & Stankovic, J. (2010). Cyber-physical systems: the next computing revolution. In *Proceedings of the design automation conference*, (pp 731–736). IEEE.

Rehman, I., Mirakhorli, M., Nagappan, M., Uulu, A. A., & Thornton, M. (2018). Roles and impacts of hands-on software architects in five industrial case studies. In *Proceedings of International Conference on Software Engineering*, (pp 117–127). Association for Computing Machinery. 10.1145/3180155.3180234

Sangwan, R. S. (2014). *Software and Systems Architecture in Action*. Auerbach Publications. doi:10.1201/b17575

Shahbazian, A., Lee, Y. K., Brun, Y., & Medvidovic, N. (2018). Making well-informed software design decisions. In *Proceedings of the 40th International Conference on Software Engineering*, (pp. 262–263). Association for Computing Machinery. 10.1145/3183440.3194961

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

Sobrevilla, G., Hernández, J., Velasco-Elizondo, P., & Soriano, S. (2017). Aplicando Scrum y Prácticas de Ingeniería de Software para la Mejora Continua del Desarrollo de un Sistema Ciber-Físico. *ReCIBE*, *6*(1), 1–15.

Sommerville, I. (2011). *Software engineering*. Pearson.

Tang, A., Tran, M. H., Han, J., & Vliet, H. V. (2008). Design Reasoning Improves Software Design Quality. In S. Becker, F. Plasil, & R. Reussner (Eds.), Lecture Notes in Computer Science: Vol. 5281. *Quality of Software Architectures, Models and Architectures* (pp. 28–42). Springer. doi:10.1007/978-3-540-87879-7_2

Velasco-Elizondo, P., & Lau, K. (2010). A catalogue of component connectors to support development with reuse. *Journal of Systems and Software*, *83*(7), 1165–1178. doi:10.1016/j.jss.2010.01.008

ENDNOTES

1 See https://www.cloudflare.com/
2 See https://www.zabbix.com/
3 See https://www.rabbitmq.com/