Software Architects in Practice

Vidya Lakshminarayanan, WenQian Liu*, Charles L Chen, Steve Easterbrook*, Dewayne E Perry

Empirical Software Engineering Lab (ESEL)
Electrical and Computer Engineering
The University of Texas at Austin, Austin TX
{vidya,clchen,perry}@ece.utexas.edu

*Software Engineering
Department of Computer Science
University of Toronto, Canada
{wl,sme}@cs.toronto.edu

ABSTRACT

We present the results of a multiple case study of how architects view and address the issues in transforming requirements into architectures in practice. Specifically we report how they view and address issues of requirements, architecture, and the transformation of requirements into architecture. We then summarize the important lessons learned from these practicing architects about this critically important step in creating and evolving software systems.

Categories and Subject Descriptors

D.2 [Software Engineering**]:** D.2.1 Requirements/Specifications, D.2.11 Software Architectures

General Terms

Software Requirements, Software Architecture, Multiple Case Studies, Architecture in Practice

Keywords

Transforming Requirements into Architectures, Architects in Practice, Multiple Case Studies, Experience Report, Managing Requirements, Creating and Evolving Architectures

1. INTRODUCTION

Research in the areas of requirements engineering and software architecture have developed independently for more than a decade [7,8]. However, in the last few years, a number of researchers have begun to investigate the relationships between requirements and architecture, and how to bridge the gap from one to the other [9,10]. In particular, techniques have been proposed for generating software architectures from the requirements [11,12,13,14]. However, there have been no studies to date on how architects currently perform this transformation in practice.

In this paper, we report the results of a multiple case study of experienced software architects. We believe that if we are to provide methods, techniques, and tools to help transform requirements into architectures then we need a detailed understanding of what architects do in practice. All too often, software engineering research pays little attention to what software engineers actually do when they build and evolve software systems.

We take an empirically based approach and use an interview-based case study methodology to carry out our investigations. Case studies are a specific empirical research method used to gain a deep understanding of a particular phenomenon in its real life context. As such, they are characterized by analytical generalizations, rather than statistical generalizations, i.e. they are not to be understood in terms of samples, but in terms of analysis and comparison of cases [4].

Our study involves a series of semi-structured interviews on requirements and architecture topics with multiple subjects. In [6] we describe our process of defining our case study from its preparation to its evidence chain and evidence trail.

So far, we have interviewed fourteen different architects from different domains (such as security, usability, product lines, etc.) in different organizations (ranging from relatively small to extremely large, from specific product focuses to general system and solution providers, etc).

The architects in this study were chosen from software intensive organizations in two general areas: in and near Toronto, and Texas (Austin, Houston and Dallas). Where the subject architects are from the same companies, they either represent different divisions (which are often the size of medium sized companies) or represent different kinds of architects. For example, in one company, we chose a usability architect and a domain specific architect; in another company, the different divisions represent significantly different domains. With a few exceptions, our subject architects have significant experience both in their roles as architects (anywhere from 4 to 15 years) and as software developers; where the subjects have only a few years experience as architects, they do have significant experience as developers (five or more years). For the architects working at large, software intensive organizations, there tend to be mature and well-defined processes to fall back on. For the architects in smaller organizations or where software provides support for non-software intensive domains, experience and domain expertise are the bases for architectural guidance.

This study investigated a number of issues in software architecting: (i) how do software architects manage requirements and make design decisions, particularly those concerning non-

functional requirements; (ii) what do practicing architects understand by the term software architecture, including the critical characteristics of a good architecture and the driving forces of its creation; (iii) how do experienced architects generate an architecture from the requirements and manage the evolution of an existing architecture. In sections 2-4 we provide a distillation of how our subject architects view and address these issues.

In section 2, we summarize how architects view, discover, anticipate, and manage requirements in practice. In section 3, we summarize how architects view software architecture, it's meaning, characteristics, driving forces, manifestation, evaluation, and tool support. In section 4, we summarize how architects view the transformation from requirements to architecture, shaping the architecture, and the effects of the shape of the problem on the shape of the architecture. Finally, we conclude with lessons learned in our multiple case study of architects in practice.

2. REQUIREMENTS

Our first set of results focus on how the architects perceive the requirements. There was a clear consensus on what requirements are - things that are wanted, needed, asked for, or demanded. However, there was less consensus on how they are handled. In this section, we describe how practicing architects view functional and non-functional requirements, discover real requirements, deal with requirements variance, and handle complex and costly requirements.

2.1 On Functional vs. Non-functional

In the literature, a distinction is drawn between functional and non-functional requirements. A typical definition is that functional requirements specify the functions that a system or software component must be capable of performing, while non-functional requirements describe general qualities that the system must possess, or constraints on the acceptable solutions, such as performance requirements, software design constraints, software quality attributes, etc.

Our subjects do not have a uniform opinion about functional and non-functional requirements. Some of them tend to distinguish between functional and non-functional requirements. On the other hand, a few architects suggested that they see no real difference between functional and non-functional requirements and each requirement (if committed) must be accommodated in the design. For example, one subject indicated that security is neither functional nor non-functional; it is simply a serious business and technical requirement that can "*make or break the deal*".

One subject considers that every requirement has both functional and non-functional aspects; depending on one's perspective and the problem context, one aspect may dominate the other. This view is echoed by another subject while discussing security and performance requirements. In security, there are standard functional requirements dealing with interoperability and manageability issues. For example, a VPN client has to implement a chosen protocol in order to communicate with the server; a system administrator needs to be able to safely update the access information using scripts overnight. As for performance, specific requirements are set to prevent performance degradation. For example, one cannot impose more than $^1/_{10}$ of a second latency on the startup of a connection; or one cannot impose more than 3% throughput overhead due to cryptography.

Although there seems to be some disagreement, the differences between them are more cosmetic than fundamental. The underlying themes of our subjects' opinions are as follows:

- Non-functional requirements are frequently used for convenience to refer to high-level properties and abstract phenomena that are desirable.
- To actually deliver non-functional properties, they must be (re)formulated in terms of functions.

2.2 Discovering the Real Requirements

Requirements set the goals and expectations for a system that is to be developed. When the requirements are not specified correctly, producing the right system becomes a challenge. Our subjects expressed that eliciting the requirements for any kind of product is complex because many stakeholders are involved. All of these people have their own views of what is important, along with their own experiences, prejudices, and perspectives of the world. Usually, the requirements are expressed in technical terms based on the current technology, rather than the problems that stakeholders actually need to accomplish. This is because the stakeholder is most likely to communicate a requirement that he is aware of based on his particular view of the world.

The most common technique for discovering the real requirements is for the architects to involve themselves in the requirements gathering task so that they understand what the stakeholders actually need. The architects with their experiences and knowledge will be able educate the stakeholders to broaden their focus and steer them in the right direction. Most, but not all, of our subjects agreed with this view. According to one of our subjects who disagreed with this, architects should communicate with the requirements engineers rather than bypass them and go straight to the customers. According to him, requirements should be left to the requirements engineers who are specially trained to handle them. Also, if the architects are directly involved with the customers there is a possibility that the architecture and requirements may become desynchronized.

Not all the requirements collected are consistent. According to our subjects, interacting with the stakeholders and educating them will help to resolve inconsistencies. One technique mentioned in our interviews for resolving the inconsistencies is brainstorming. The purpose of brainstorming is to use the group effect to generate good ideas and solve problems. The technique focuses on generating as many ideas as possible. The resulting ideas are evaluated, and the one best suited for the stakeholders is chosen. Unfortunately, reality does not always allow architects the opportunity to resolve all inconsistencies. One of our subjects admits that there are times when architects must rely on their own judgment to resolve inconsistencies – this is risky and not recommended; however, sometimes there is no alternative.

2.3 Anticipating Changes

Constant change is a common theme in software development. Changes represent new requirements or re-evaluations of existing requirements. They are usually stated as additional functionality or an incremental problem to solve. One of our subjects indicates that in order to create an architecture that can solve similar problems for different customers and adapt to changing needs,

architects need to take a step back and see the big picture to discover what the true requirements are so that resources can be better planned and more important problems are addressed. He further suggests that "*what is going to change really is related to the domain*" and trying to handle everything without an understanding of where changes are likely to occur is a bad approach in software engineering. One key thing to anticipating changes is to understand the business needs and what the customers are trying to accomplish; otherwise, it will be just a guessing game.

Another subject suggests that to satisfy every customer's request may not be the right thing to do because some requests may not fit in the long-term direction of the industry. He believes that the key to understanding where the industry is going in the long-term is to involve not only the industry community but also the research community. With the knowledge and experience exchange, both communities can benefit.

Professional experience can contribute greatly. Typically, experienced architects will seek to reuse information they have seen and predict the problems and changes that may come in the future. Experience helps architects to identify what is hard and what is an issue.

A number of subjects suggest that using good design principles provides an indirect solution for dealing with changes and anticipating future requirements. Some of the examples described by our subjects are as follows: (i) when the natural cohesion from the requirements is reflected in the design, maintenance and enhancements will fall into alignment and become easier, (ii) use modularity to minimize the effects of changes on the entire system, and (iii) make solutions more generic to provide reusable and extensible frameworks.

The benefit of anticipating changes does not come free or without challenges. First, it is a time consuming task. There may be cases when this investment is not warranted. This is especially true for products with shorter anticipated life spans and in cases where the time to market for the first release is a dominant factor.

"Disruptive technologies", unanticipated technologies that become dominant in a given industry and radically change this industry, provide a special challenge. One way of dealing with such technologies is to design the system to be adaptable to mitigate the risk of unforeseeable changes.

Another challenge is being able to correctly interpret the given requirements in the absence of contextual information. In such cases, making incremental changes and integrating back into the product is appropriate. It also helps in uncovering any hidden assumptions.

If there is a 'middle man' (such as marketing) filtering and interpreting the requirements before passing them on, the real problem may be obscured from the architect. In order to mitigate this problem, one subject believes that an architect needs to "*be analyzing a number of problems in aggregate*" instead of focusing on a given single instance before committing to a solution.

One of the ways to manage change is to lay out a roadmap to help customers move on from one version of the system to the next. By having a clear plan for upgrades and changes, the problem of dealing with multiple versions with different configurations can be eliminated.

2.4 Managing Requirements

The production of high-quality software is a major concern for today's software industry. Delivering software on time and within budget, and satisfying all of its requirements pose significant technical challenges for researchers, managers, and practitioners. Every client wants a product that can do everything that might conceivably be needed. Such a product would take an unacceptably long time to build and cost far more than the client considers reasonable.

Software architects use a number of strategies to manage complex and costly requirements. Understanding the rationale and the business problem of the given requirements by asking the 'why' and 'what' questions is critical. Answers to these questions provide both deeper insight into the real problem and more options for solving them. During these back and forth question-and-answer sessions, the negotiation of the requirements is actually taking place to produce acceptable requirements that can be signed off by both sides.

Conflicts, unrealistic 'sales promises' and impossible requirements must all be negotiated with the customer. In some cases, a compromise can be achieved, but in others, 'no' is the only answer. Under such circumstances, prioritization helps to determine which requirements are the key and which are subordinate. These priorities can be based on the cost/benefit analysis or areas of particular concern to the stakeholders. There may still be a case that there are too many requirements all with the highest priority where an "*80/20 rule*" could be applied, that is take 80% and leave the rest behind with the consent of the stakeholders. Problem decomposition is frequently used to manage complexity. In most problems that involve automating human activity, decomposition is not very hard to accomplish because humans will not typically do anything too complicated to be modeled. The reason for decomposition is to make implementation easier. The refinement methods and the techniques for decomposition are based on the architect's experiences and domain knowledge.

Since there are no specific techniques or methods to handle complex requirements, it is important that the architects perform post mortem analysis so that they can analyze their mistakes and come up with better methods to handle such requirements. However, such post mortem analysis is usually not conducted well in practice because of a lack of tool support to trace back to the requirements.

So far, we have discussed how architects uncover real requirements, manage different requirements, and anticipate changes. In the next section, we will describe our subjects' views on software architecture.

3. SOFTWARE ARCHITECTURE

In the literature, software architecture is generally discussed in terms of a collection of interfaces and components, and it is mainly used as a communication vehicle to balance different interests. In this section, we report on how architects understand the meaning of software architecture, its characteristics and driving forces, and how the architecture manifests itself in the final system. Furthermore, we describe the techniques used by our subjects to evaluate the architecture and to do their jobs as architects.

3.1 Meaning of Architecture

The software architecture of a large software system is mainly comprised of the major components of the system and their interconnections. It can also be defined as a collection of components and interfaces, which ties together the different requirements such as business requirements and technical requirements in a loosely coupled fashion. Moreover, it is commonly held that multiple views are required to gain a proper understanding of the various aspects of a software architecture, as seen for example in Kruchten's 4+1 views.[1] [3]

According to one subject, architecture is an eight faceted diamond that can be described in different ways by different stakeholders. For example, from an administrative point of view, the architecture is described using new templates; but from a product development point of view, the architecture may not be fully described or may be described like a deployment view. However, independent of how many and which views are needed, the critical question is how much work has to be done upfront to describe the architecture before moving on to the low-level design work.

One communicational use of architecture bridges the gulf between the architect and project management. Clearly, the architecture is of critical use in planning the development of the system, and its structure can facilitate or hinder the development interval. A well-documented architecture facilitates a shorter development interval.

Some of our subjects believe that the architecture is based on the problem it solves, but how well it models the problem varies. The architecture does define the solution to the problem being solved, but by looking at the architecture, sometimes it is difficult to find the exact problem. It is sometimes the case that the architecture does not have anything to do with the problem being solved because architecture may be analyzed independently from the processes and decisions. In other words, someone outside of the domain may be able to investigate a system and ultimately understand the architecture regardless of the problem being solved. Furthermore, the architecture is the first step in the construction of the solution, and they agree that the architecture is in the solution space rather than in the problem space. [1]

From the data collected, the architect's views on software architecture can be summarized as follows:

- Multiple views are needed to satisfactorily understand and explain a software architecture.
- It is a technical need that is used as a way of communicating the technical information about the system to the various stakeholders. It guides them in their tasks, whether it is creating white papers about the system or implementing the system.
- It can be viewed as a framework. It always exists in the system whether it is documented or not. However, it is important to note that having a documented architecture facilitates faster development.
- It is a collection of interfaces and components. In a large software system, it can be viewed as the major parts of the system and their interconnections.
- Domain and background knowledge can have a profound effect on how architects view architecture. For example, according to one of our subjects, a logical view of the system, and not necessarily a specific implementation, constitutes architecture. However, another subject expressed that he usually creates a proof of concept implementation of the architecture.

3.2 Characteristics

Software architecture is a structure that encodes the organization and interactions of components. It contains several parts that may even include things that are unfamiliar to the architect. These parts may have high visibility within the system and may be the central piece that is used by lots of other components. It is important that the architect identifies these risky areas and does a careful design. In theory, it is ideal that the system goals and rationale be captured within the architecture. This is important because as the next person comes along and modifies the system it is required that he does not break the fundamental assertions about the architecture or the implementation. However, some of our subjects believe that it can do more harm than good to capture the rationale at the very early stages because people will tend to treat the commentaries as the actual system specifications. The key to the disagreement on rationale is when to capture it and how much to capture. Overall, everyone agrees that it should be captured. It is best to capture it at the end of the design stage or even the implementation stage. Reasons for this include reducing confusion and unnecessary communications. It is also important to note that only the key elements are captured so that the fundamental decisions are preserved and the maintenance of this information does not become a burden during system evolution.

A majority of our subjects agreed that a good architecture is a critical factor in the success of a system's development. Certain critical characteristics of the architecture define its goodness. Software architecture represents a common high-level abstraction of a system that most if not all of the system's stakeholders can use as a basis for creating mutual understanding, forming consensus, and communicating with each other. As stated by Conway's Law, *"Organizations which design systems are constrained to produce systems which are copies of the communication structures of these organizations"*. In other words, the organization of the system is the same as the organization of the people building it. It is well understood that a large system cannot be built without a large number of people and a large team of people need to communicate among themselves and take up responsibilities for different parts of the system. Therefore, the system starts echoing the same structure and reflects the architecture of the groups.

A common theme among our subjects is that the architecture *is mainly used as a communication vehicle.* However, communication can be done in a broad range and is not limited to only the delivered architectural documents. It can be done through e-mail, presentations, white papers, etc. Whatever the medium,

[1] These five views are 1) the logical view, 2) the process view, 3) the physical view, 4) the development view, and 5) the use case or scenario view.

architecture is widely used for communication purposes.[2] Having an architecture gives stakeholders confidence towards the satisfaction of their goals. According to one of our subjects, the bottom line is to *"enable everyone with a vested interest to do what they need to do"*. For example, a user is concerned with whether she can get the desired end product and a developer is interested in meeting the deadlines of the deliverables. The architecture provides information on all of these aspects.

Furthermore, the architecture can be used to verify that the problem has been understood correctly. Using the architecture an architect can uncover misunderstandings of the requirements. This process of verification may lead to a new set of improved requirements.

One of the desired purposes of the architecture is to save other people's time. The architect needs to develop an architecture in such a way that it reduces the development time, maintenance time, time needed to develop the user manual, etc.

Our subjects expressed that there are several required properties of the architecture. (i) The architecture needs to be open and pluggable. This is because there is always advancement in the technology and it is important to leave enough room so that modification of different components does not affect the rest of the architecture. (ii) The architecture should be adaptive in nature. It should be designed in such a way that when the changes occur the existing system should not be simply discarded. In other words, the architecture needs to be robust and not brittle because different enhancements can cause total reorganization and restructuring of brittle architectures. (iii) It is also important that the architecture is simple and elegant, but at the same time, it is not desirable that the architecture adheres to a specific style at the cost of functionality or purpose. (iv) Abstraction is needed for the architecture and it should capture the important aspects but leave the details open. This property is necessary to avoid over constraining the architecture.

3.3 Driving Forces

Multiple decisions drive the development of the architecture. Most of these decisions are made for sound technical reasons. For example, one such reason is the ability to satisfy customer requirements. In fact, sometimes a particular requirement may shape the architecture of the system if there is only one way to fulfill it, as can be the case when dealing with security requirements where compromising can mean leaving the system open to attackers. For example, once you have decided on a security design goal to impose a specific work factor on the attacker *"you basically have to hit that mark or do better; you can't […] really continuously tune your level of security"*. In other words, you either succeed in preventing the attack or you don't. Security is not a tunable property like performance.

Another technical reason is the maintainability of the system. If you do not have the right structure, it is difficult to adapt the solution to future needs, and maintenance may become an issue. Ideally, changes and enhancements to the system should be confined to a small part of a system to prevent the effects from the change from affecting the entire system. *"So if you can isolate that new stuff, or at least isolate the enhancement to within a subsystem or even a component within the subsystem, that's the ideal."*

In addition to the technical reasons, some architectural decisions are made for reasons such as business alliances. For example, a company may dictate that a specific tool be used simply because it has a contract with other company to use its tool. They can also be made for the reasons like personal gain. For example, a project manager may insist on using a certain new technology for the sole purpose that he can include in his resume about his experience with the new technology even if it is inappropriate for the project itself. These decisions can create serious problems by over constraining the architecture and forcing the developers to use an inappropriate framework.

On the other hand, projects are not always started by an upper-level management decisions; they can 'sneak-in' as a "skunkworks project", a project that is started by developers on their own initiative. This may be a good idea though they may not be clearly profitable. Nevertheless, they decide to make a prototype, which eventually evolves into a real project. In such a case, the evolution process is the driving force in the creation of the architecture.

3.4 Manifestation

It is debatable whether and how software architecture manifests itself in the end product. Our subjects feel that the architecture does manifest itself, at least partially, in the end product. The design patterns and naming conventions tend to be clearly visible. Also, the decomposition of the architecture into subsystems and components can be seen in the code. *"A lot of times, you'll see things mentioned in the architecture still mentioned at the product level."* In fact, one of our subjects suggests that *"if you can't see it in the code, then you probably got chaos going on."* However, the more detailed parts of the architecture and the intermediate steps that led to its creation (such as rationale, negotiations, and processes) are usually not visible. These aspects may be kept in documentation but are, in general, hard to maintain.

3.5 Evaluation

Software architecture is a foundation for building successful software-intensive systems. Evaluating the architecture before implementation can reduce costs through early detection of errors and problems. In addition, evaluations also reduce the risks of disaster projects. It helps to increase the understanding of the system and to clarify and prioritize the requirements. In order to evaluate architecture, there needs to be a set of specific criteria for measuring goodness. However, there is no universal set of criteria for determining goodness in practice. Our subjects discussed the following criteria that they have used: (i) a good architecture should be extensible, adaptable, elegant, and represent an abstraction of the problem, (ii) it needs to be marketable since there is no use in having an architecture that nobody cares about, and (iii) it should have clean well-defined interfaces.

It should be noted that the most important criterion for goodness can vary from situation to situation. What is critical for one case may not apply to another. In general, an architecture has to be implementable, support good performance, and be reliable.

[2] This use of architecture as a communication mechanism is consistent with the results of Sim's study of the social aspects of architecture [5].

Once the criteria for determining the goodness of the architecture are established, different techniques can be applied for evaluating the architecture. Metrics can be used to detect problems quickly before doing a serious review. If there are well-established metrics with expected values, then deviations from those values could indicate problems. For areas that are immature and unstable, there may not be a simple, easily defined measurement. Because the domain is changing so rapidly, it is impossible to create a practical ontology that would even come close to covering everything. In such cases, the only real technique for evaluating the architecture is to think about it carefully.

3.6 Tool Support

There are several tools and techniques that the architects use for constructing architectures, understanding requirements, and doing their jobs as architects. The tools currently used in practice include requirements management systems, code analysis software, and logic programs. Tools for capturing the rationale behind the architecture and linking the requirements with the architecture are still not very well developed and trying to document this still requires a huge amount of effort. However, such tools would be nice to have because it is important that this information is not lost. Not all available tools are readily adopted. For example, tools that are overly detailed for the task or are difficult to use will not be adopted. It is evident that tools can make the job easier, but they are not a silver bullet for solving the problem of radical design. This is because it is not possible to create a tool to reason about all the issues that an architect would normally need to consider.

In addition to the tools, there are different general and specific approaches that the architects use for constructing the architectures. The first approach is to control the complexity by decomposing the problem into things that are of the right size to think about. The second approach is to apply reuse in terms of both reusing past solutions and planning for future reuse. Even in cases where it is not possible to apply reuse directly, it is best to break down the problem into familiar pieces and then apply reuse to these pieces. The third approach is using iterative development or prototyping to make sure that the solution is feasible before committing to it.

Specific methods that the architects use for constructing the architecture are as follows: (i) the use of design and architecture patterns; (ii) the use of standard frameworks such as J2EE ; (iii) the use of specialized models built as references for the architecture (for example in security, the architecture is based on the use of threat models.); and (iv) using Use Case analysis [1] to model higher levels of abstraction. One subject specifically indicated that asking 'why' questions allows him to uncover higher levels of abstraction and brings an understanding of the big picture. In turn, he was able to have a broader range of options to solve the problem. In this sense, use cases were a useful tool to him in ensuring the requirements are met; *"you have to run through the use cases rigorously and make sure that we are meeting all the sets of requirements"*.

We now discuss how architects transform requirements into architectures.

4. REQUIREMENTS TO ARCHITECTURE

The requirements and architecture of any software system are interdependent. Little guidance and few methods are available to refine a set of software requirements into an architecture satisfying those requirements. We have collected data from our subjects on two subtopics – how architecture is shaped and its relation to the problem structure.

4.1 Shaping the Architecture

The goal of the requirements phase of software development is to decide precisely what to build and how to document the results. The architecture is the first artifact in the development process that addresses the requirements of the system. In designing and building software systems of any complexity, understanding requirements and using them to make informed architectural decisions is crucial to project success.

Requirements take different forms and they vary from organization to organization. Typically, the requirements capture some definition of the product. For example in one company, they are captured in a product content document and are used as input to the system design document. The next step for this company is then to transform these requirements into a technical architectural design document and then produce a component design document that includes details for the implementation of each component.

According to our subjects, it is important to consider all requirements, whether functional or non-functional, from the beginning. However, it is generally more difficult to obtain the details of non-functional requirements upfront. Some of our subjects claim that the current trend in industry is to deal with functional requirements earlier than non-functional requirements because setting the functional requirements in the initial architecture frame is much easier. Furthermore, customers are usually better at conveying functional requirements than non-functional requirements. But even then their input may be entirely misleading *"some people know exactly what they want until they get it"*.

Nonetheless, non-functional requirements are critical to the formulation of software architecture and can significantly influence the shape of the architecture. One subject expressed that in some applications, non-functional requirements are an integral part of the system rather than an add-on as an afterthought. For example, in x-ray machines, reliability is the most fundamental requirement. In other cases where non-functional requirements are close to the implementation level, what architecture leaves out can be as important as what it includes. As one of our subjects notes, the architecture should not be overly constraining on details such as how to perform disk I/O, since the key factors which influence performance in that case would be very low-level implementation decisions that are best left up to the developers.

As the system evolves, changes in functional and non-functional requirements have a variety of effects on the architecture. According to one of our subjects, changes to functional requirements are easy to handle because they tend to be contained in a smaller set of components rather than affecting the entire system. In contrast, changes to non-functional requirements can be very disruptive and can cause serious problems to the architecture because non-functional requirements tend to be requirements that are pervasive throughout the entire system.

When transforming requirements into architectures, it is tempting to rush headlong into the trap of thinking about the solution too soon. Software development problems are about the world outside the computer - the real environment in which the system must operate - and demand consideration of the surrounding characteristics, relationships, and context. According to our subjects, as a first step it is important that the architect shape the problem by providing the context around it, which would in turn help the business and sales people understand what the customer actually requires.

The next step is then to understand this problem shape and come up with a shape for the solution. The architects tend to draw upon previous experiences to shape the solution. Reusing past solutions is a natural tendency, and analyzing the requirements thoroughly is one of the ways to determine how to reuse familiar existing solutions. Even in cases where it is not possible to reuse past solutions and a new solution must be developed, architects still have a tendency to break down the new solution into pieces they are familiar with, i.e. pieces that are similar to ones they have used in the past.

If the domain is entirely new and unfamiliar to the architect and the development team, one method for constructing an architecture that is easily understood is to use a metaphor that is familiar and represents a similar type of problem, although it may be from a completely different domain. For example, one of our subjects discussed a project that involved creating software controls for a robotic tape deck system. In order to communicate effectively about the control system, he structured the architecture using a metaphor of passengers (the tapes) on a mass transit system (the robotic arms) in order to move from one terminal (tape deck) to the next.

4.2 Effect of the Problem Structure on Architecture

Because of the complexity of the requirements and the problem space, there is the danger of misunderstanding them. These misunderstandings could lead to development failures. To avoid such failures it is important to have a clear and good understanding of the problem structure and then develop the architecture based on this understanding.

Domain analysis is considered essential by our subjects in designing high-quality software systems. If carried out properly, domain analysis can help designers understand the requirements, identify the fundamental abstractions, verify the design, and drive the implementation. In order to deliver what the customer actually needs and understand the relationships between the different requirements, it is important that the architect has a good and thorough understanding of the problem domain. For example, one of our subjects credits the success of architecting a project to his having lived the problem and thus understanding the domain.

However, according to our subjects, it is very difficult (or at least, very unusual) for an architect to have both sound technical skills and deep domain knowledge. This is because part of the job of an architect is dealing with the business aspects while the other part is dealing with technical matters. It is often very difficult to find a person who can do both sides of the job. Architects tend to be generalists, having broad knowledge rather than specific, deep knowledge.. Therefore, in order to understand the problem deeply and to transform the business requirements into a design solution, it is important to have a multi-disciplinary team consisting of requirements engineers who have in depth knowledge of the domain and architects who have sound technical skills.

In systems where security is critical, in addition to the logical (or if you will, intellectual) aspects, the physical aspects play an important role. It is therefore important that the architects look at the entire system and not just at a particular set of technologies. It is also crucial that architects detect and analyze any assumptions made about the system and its environment, which are not explicitly stated. Failure to consider hidden assumptions can be disastrous; according to one of our subjects, a protocol for cell phones had to be reworked because it had rested on the flawed hidden assumption that attackers could not put up anything which could act as a cell phone tower. In other words, architects need to have a broad knowledge of the entire domain and be aware of the underlying assumptions of the system and its environment in order to come up with the right solutions.

It is not always possible to have a team with domain experts. In such circumstances, the following methods help in coping with the lack of domain knowledge: (i) to be agile and ready for change; (ii) to hire a marketing firm who would help in understanding the problem; or (iii) to interact directly with real customers. Despite the difficulty in obtaining the domain knowledge, it is a worthwhile activity as it can result in more generalized solutions, which will have immense benefits in the future.

5. LESSONS LEARNED

We summarize lessons learned from the interview data as well as our own remedies in the following eleven lessons.

Lesson 1 *Architects tend to be generalists rather than specialists and draw on a wide-ranging background to provide the overall structure and interdependencies of the architecture.*

Lesson 2 *Regardless of whether or not our subjects made a distinction between functional and non-functional requirements, all of them agreed that both need to be considered and reflected in the architecture.*

Lesson 3 *Change is inevitable in software developments, so anticipating changes can provide significant benefits by reducing the amount of effort needed to implement changes later on. Although there are not many formal approaches for doing this, there are heuristics that are used in practice. Professional experience plays a large role in identifying which parts of the system are likely to change.*

Lesson 4 *Research needs to address issues of how to collect contextual information along with the requirements and how to mitigate the effects of having a 'middle man' filtering the requirements.*

Lesson 5 *Architects need to ensure that they are working with reasonable, consistent requirements. If there are problems with the requirements, then they should either send the requirements back to the requirements engineers for rework, or they should be more involved in dealing with the customers directly to negotiate the requirements directly. There is some disagreement over which method is better; however, most of our subjects favored the latter approach.*

Lesson 6 *Software architecture is multifaceted; it can be described in different ways by different people. The most common definition is that it is a collection of interfaces and components whose main purpose is to be used as vehicle for communication with the various stakeholders. This communication gives confidence to the stakeholders that their goals are being addressed and verifies that the problem is correctly understood.*

Lesson 7 *The architectural decisions that drive the development of the architecture should only be made on the basis of sound technical reasons, such as satisfying customer requirements, providing better maintainability, etc. However, sometimes, things other than technical reasons, such as business alliances and personal gain, play a role as well.*

Lesson 8 *Evaluating the architecture reduces the cost of the project by detecting errors and problems earlier rather than later. There are several criteria for determining the goodness of architecture. Architecture should be extensible, adaptable, and elegant and have good abstraction; it also needs to be marketable and have well-defined interfaces.*

Lesson 9 *Tools and techniques like requirements management systems, code analysis software and logic programs are adopted by the architects to understand the requirements and to do their jobs as architects. It should be noted that these tools must be easy to use; if a tool is too complicated or burdensome, then it will not be adopted.*

Lesson 10 *The first step in transforming requirements into architecture is to shape the problem by understanding the context around it. The next step is to use the shape of the problem, along with reusable parts from the past, to shape the solution. If there are no reusable parts, one approach is to use a well-understood metaphor, using a similar type of problem to guide the shaping of the solution for the current problem.*

Lesson 11 *Domain knowledge is essential to constructing a good architecture that addresses the problem and making sure that it does not rest on faulty assumptions.*

6. CONCLUSIONS AND FUTURE WORK

We have summarized some of the critical results of our interviews and drawn what we consider to be important lessons from this part of our multiple case study. There are still lessons to be obtained from our architects beyond what we have provided here. We expect these further results to be made available in workshop, conference and journal publications. We will also make available appropriately sanitized technical reports summarizing the results of our interviews and providing salient quotes [15].

In [6], we discussed two important things: 1) our chain of evidence; and 2) various validity issues with the then current state of our case studies. Our main concern then was the concentration of architects from one international company. Since then we have expanded the representation in terms both of companies and of domains. Thus, we feel that the external validity of our multiple case study has been strengthened significantly.

One of the frustrating things about our case studies, however, has been the lack of significant details about the specific mechanisms and intellectual tools for transforming requirements into architectures. We have generic details for such things as design and architecture patterns, references to architectural styles, the use of threat models, etc., but little in the way of specific details. We hope to remedy that, at least in part, with one or more participant observer studies where one or more of our students work in partnership with architects long enough to see what happens in practice.

The current results of our work and progress can be found and downloaded from the project website [15].

7. ACKOWLEDGEMENTS

We thank all of our anonymous interviewees and their companies for their participation and contributions. This research is supported in part by NSF CISE Grant CCR-0306613 and IBM CAS Fellowship. We also thank our co-researcher Deepika Mahajan who arranged the interviews of two of our architects and assisted with our transcriptions.

8. REFERENCES

[1] Cockburn, A., *"Writing Effective Use Cases"*, 2001, Boston: Addison-Wesley.

[2] M. Jackson and P. Zave, "Deriving specifications from requirements: An example"**,** *ICSE* May 2005.

[3] P.B. Krutchen, "The 4+1 View Model of Architecture"**,** *IEEE Software*, November 1995.

[4] Robert K. Yin, *"Case Study Research: Design and Methods"*, 3/e. Thousand Oaks, CA: Sage Publications, 2002.

[5] Susan Elliott Sim, "A small Social History of Software Architecture", *Proceedings of the 13th International Workshop on Program Comprehension*, May 2005.

[6] W. Liu, C. L. Chen, V.Lakshminarayanan, D.E. Perry, "A Design for Evidence-based Software Architecture Research", *Workshop on REBSE'2005*, ICSE May 2005.

[7] Bashar Nuseibeh and Steve Easterbrook, "Requirements engineering: a roadmap", *Future of Software Engineering*, 22nd International Conference on Software Engineering, Limerick Ireland, June 2000, pp 35-46.

[8] David Garlan, "Software architecture: a roadmap", *Future of Software Engineering*, 22nd International Conference on Software Engineering, Limerick Ireland, June 2000, pp 91-101.

[9] J. Castro and J. Kramer, editors, *The First International Workshop on From Software Requirements to Architecture (STRAW'01)*. International Conference on Software Engineering 2001. Toronto 2001,

[10] D. M. Berry, R. Kazman, and R. Wieringa, editors, *The Second International Software Requirements to Architecture Workshop (STRAW'03)*, International Conference on Software Engineering 2003, Portland OR, May 2003.

[11] M. Brandozzi and D. E. Perry, "From Goal-Oriented Requirements to Architectural Prescriptions: The Preskiptor Process", in [9].

[12] A. van Lamsweerde, "From System Goals to Software Architecture", In M. Bernardo and P. Inverardi, editors, *Formal Methods for Software Architectures*, pp 25-43, 2003.

[13] Divya Jani, Damien Vanderveken and Dewayne E Perry. "Deriving Architectural Specifications from KAOS Specifications: A Research Case Study", *European Workshop on Software Architecture* 2005, Pisa Italy, June 2005.

[14] Damien Vanderveken, Axel van Lamsweerde, Dewayne E Perry, and Christophe Ponsard, "Deriving Architectural Descriptions from Goal-Oriented Requirements Models", September 2005

[15] Dewayne E Perry. NSF Grant CCR-0306613 Project Web Site "Transforming Requirement Specifications into Architectural Prescriptions", http://www.ece.utexas.edu/~perry/work/projects/nsf-r2a/ [29 October 2005]