On the Modularity Impact of Architectural Assumptions

Dimitri Van Landuyt, Eddy Truyen and Wouter Joosen

IBBT-DistriNet
Celestijnenlaan 200A
B-3001, Belgium
{dimitri.vanlanduyt,eddy.truyen,wouter.joosen}@cs.kuleuven.be

Abstract

In software architecture design, the end product is the combined result of a wide variety of inputs, most of which are provided by the non-technical stakeholders. These include the analysis of the problem domain, the functional and non-functional requirements, the architectural or technical constraints. However, a software architecture is typically also influenced by different, less visible factors such as the architect's prior experience and his creativity. In this paper, we focus on so-called architectural assumptions, which are key premises made by technical stakeholders in the early phases of the software development life-cycle.

Often these assumptions are made silently and not documented explicitly in the description of the architecture. As a result, they introduce a certain degree of rigor in the software product that hinders the evolvability, variability, and reusability of the architectural solution as a whole and its individual building blocks.

Additionally, architectural assumptions in many cases exert a crosscutting influence on the software architecture and its description. This makes it hard to discover them, assess their individual architectural impact, and treat them as first-class architectural elements.

In this position paper, we explore and discuss these modularity problems in specific examples from a patient monitoring system (e-health). Furthermore, we introduce the distinction between problem-space and solution-space architectural assumptions, and we discuss their intrinsic differences.

Categories and Subject Descriptors D2.1 [*Requirements/Specifications*]: Methodologies; D2.11 [*Software Architectures*]: Requirements/Specifications

General Terms Design, Documentation, Management

NEMARA'12, March 27, 2012, Potsdam, Germany.

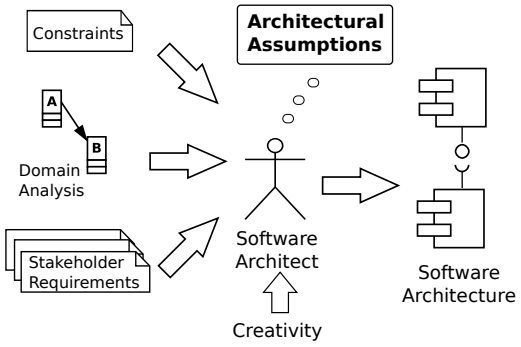

Figure 1. Schematic overview of a top-down architecture creation process.

1. Introduction

Top-down software architecture design starts with the systematic collection, analysis and documentation of the stakeholder requirements (functional and non-functional). Although these stakeholder requirements are the main inputs for the design of a software architecture, many additional factors come into play in the creative process to come up with a full-fledged architectural solution. Fig. 1 represents this graphically. These additional factors include project constraints, the architect's prior experience, and even his creativity. *Architectural assumptions* [16] are key premises made in the early phases of development life-cycle. They are similar to regular requirements, but differ in the sense that they are imposed not by the direct stakeholders of the end-product (e.g. the customer), but by technical stakeholders such as the requirements engineer, the domain analyst, and the software architect.

Not unlike other stakeholder concerns, architectural assumptions manifest themselves in many different forms: some will directly constrain the architecture, others will introduce direct requirements. However, they are typically made silently and motivated only in the heads of the technical stakeholders involved. Although there has been some work focusing on retro-actively discovering architectural assumptions as part of architecture recovery activities (deriving the architecture from existing systems) [16], there is not much work about how to deal with such assumptions

early in the development process (requirements and architecture). Nonetheless, it has been pointed out that they can be a significant contributor to the failures or successes of software-intensive systems [6, 7].

In this paper, we introduce the specific distinction between *problem-space* and *solution-space* architectural assumptions, which depends on the development phase in which they are made. Additionally, we discuss the impact of architectural assumptions on software modularity, and how this potentially introduces modularity problems already at the early stages of the software development life-cycle (requirements engineering and software architecture design).

We illustrate these concepts and problems in a e-health case study that is inspired on a number of research projects [3, 4, 17]. Specifically, we looked at a patient monitoring system for continually and remotely monitoring the health of cardiovascular disease (CVD) patients. Patients enrolled in the patient monitoring system are given a wearable device (e.g. printed on a t-shirt), which holds sensors that continually measure a number of medical parameters such as the patients' heart rate. Next, these sensor data readings (measurements) are packaged and sent to the back-end system. There, advanced risk assessment algorithms (clinical models) are executed to evaluate whether or not the patients' health status is deteriorating and the responsible health professional should be notified.

The remainder of this paper is structured as follows: first, Section 2 introduces and discusses the key distinction between *problem-space* and *solution-space* architectural assumptions. Then, Section 3 presents examples of both and presents the problem statement. Section 4 summarizes the findings and concludes the paper.

2. Problem-space versus Solution-space Architectural Assumptions

Roeller et al. [16] have defined the term *architectural assumption* as a *"general denominator for the forces that drive architectural design decisions"*. Paraphrased, architectural assumptions lead to and influence the key decisions made during architectural design.

In this paper, we distinguish between *problem-space* and *solution-space*[1] architectural assumptions. Problem-space architectural assumptions are assumptions made during requirements engineering or domain analysis[2]. On the other hand, solution-space architectural assumptions are assumptions made during architectural design[3]. Roeller et al. [16] describe: "*Just like it is difficult to distinguish between the 'what' and 'how' in software development, so that one person's requirements is another person's design, it is also difficult to distinguish between assumptions and decisions.*". However, the key difference is the fact that solution-space architectural assumptions represent or lead to key decisions (invariabilities) that shape the architecture, while problem-space architectural assumptions are preliminary: they may still be altered, invalidated, decomposed or merged together later on. The set of problem-space architectural assumptions is the minimal yet necessary set of assumptions that the requirements engineer is forced to make in order to write meaningful requirements.

Architectural assumptions differ fundamentally from the other influences on software architecture sketched earlier in Fig. 1 because they are imposed by technical stakeholders such as the requirements engineer or the software architect. They are commonly used as a technique to converge as quickly as possible to an architectural solution: by making reasonable architectural assumptions, the search space (problem or solution space) is reduced early on.

3. Illustration of Architectural Assumptions in an E-Health System

Below, we illustrate both types of architectural assumptions in the patient monitoring system discussed in Section 1.

Architectural Assumptions in Scenario-based Requirements. The strength but also the weakness of scenario-based requirement elicitation techniques such as quality attribute scenario elicitation [2, 10] and use case engineering [9] is that they force the requirements engineer to be very specific. As a result, he is is often forced to make some implicit early architectural assumptions in order to capture the specifics of the requirement scenario in mind.

For the patient monitoring system, we have relied extensively on quality attribute scenario and use case elicitation to analyze and document the system requirements. In these types of requirements, early architectural assumption commonly represent initial elements of architectural building blocks of the envisaged system. The Performance scenario below provides an example.

- **Source:** new information (update of sensor data readings or questionnaires), by patient, caretaker, GP, trustee, or specialist
- **Stimulus:** patient risk estimation by applying clinical models
- **Artifact:** the (sub-)system responsible for performing risk estimation and issuing notifications
- **Environment:** Normal execution modus
- **Response:** If the system does not meet the deadlines specified below:
  - in normal modus, the subsystem processes the incoming information updates in a first-in, first-out order
  - in overload modus,
    - the subsystem changes the processing order by prioritization according to the patient's risk level (red over yellow over green)
- **Response Measure:**
  - In normal modus, the system goes into overload modus when the throughput > 20 risk level estimations/minute;
  - In overload modus, there is no starvation of risk estimation jobs for patients with a green risk level.

[1] In literature, these are typically called 'architectural assumptions' [11, 16].

[2] In the left peak of the Twin Peaks model [13].

[3] In the right peak of the Twin Peaks model.

This example describes the situation in which too many concurrent requests to process incoming sensor data readings (scheduled for risk estimation) leads to a situation of overload, and prescribes how the system should react in this case. The strength of quality attribute scenario elicitation is the high level of detail and fine granularity in which they are documented. A downside however, is that the requirements engineer authoring such scenarios is forced to make some initial assumptions about the system's architecture. In this example, by referring to "the subsystem processing the incoming information updates", the requirements engineer has made the tacit assumption that there will be in fact one separate subsystem for this, and that it will use a priority queue for the incoming data packages. However reasonable this may seem, such assumption acts as an early reduction of the solution space, and may drive the architect away from alternative solutions.

In this case study (and others), we observed the following modularity problems:

- In these requirements, problem-space architectural assumptions made tacitly and implicit. They are not easy to derive from the requirement specifications, as there is no explicit difference between what is an essential part of the stakeholder requirement and what was assumed in order to result with a good characterization of those stakeholder requirements.
- Problem-space architectural assumptions are tangled and scattered throughout the requirements body, both over requirements of the same type (e.g. use case) and across different requirement types (between quality attribute scenarios and use cases): they are crosscutting. As a consequence, they might differ slightly per instance which causes vagueness and again, implicitness. Assessing the impact of a single problem-space architectural assumption requires a great amount of mental effort (i.e. iterating over all requirements and manually assessing whether or not it was affected by the silent assumption).

Risk assessment as a cloud service. Fig. 2 presents an excerpt of the patient monitoring system architecture, which is designed with the Attribute-Driven Design (ADD) process [2]. Specifically, Fig. 2 zooms in on the `Decision Support System` (`DSS`). This is the component (mentioned earlier) responsible for processing the incoming data and assessing whether or not the risk level of the patient should change. When this is the case, it issues a warning to the medical supervisor (e.g. the patients' GP).

Early on in architectural development, the architecture team recognized the potential of moving this risk assessment component (the `DSS`) to the cloud. Indeed, by doing so, the scalability, performance and availability requirements can be mitigated more easily, but more importantly, this would create the additional business opportunity to offer risk assessment as a separate service and sell it not only as an enabling service for monitoring patients with cardiovascular disease,

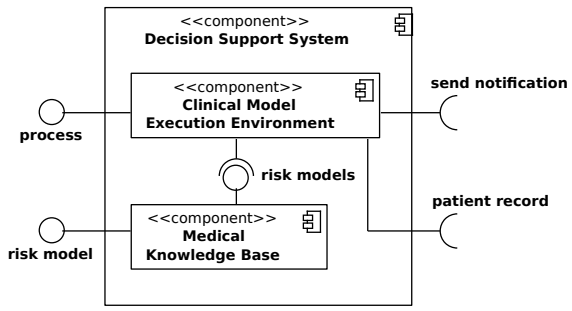

Figure 2. Design of the `Decision Support System`.

but also for different chronic diseases, such as hypertension, asthma, etc.

As this was not an explicit requirement or constraint for the stakeholders of the patient monitoring system, preparing the `DSS` component to evolve easily to a cloud context and to be flexible w.r.t. different risk assessment algorithms has not played a primary role in its design: it was no explicit driver in the ADD process. However, it is clear in retrospect that the awareness of cloud potential has indeed influenced the architectural design:

- **Design for change:** in order to be able to offer this particular service in a different context (e.g. to support risk assessment for a different chronic disease type), one first decomposition decision of the `DSS` involved separating the actual risk model from the application or *execution* of such model. This has lead to the definition of the `Medical Knowledge Base`, and the `Medical Model Execution Environment` respectively. Changing the risk assessment algorithms (also called 'risk models') or updating them is just a matter of adapting the `Medical Knowledge Base`, and this can be done at run-time.
- **Low coupling and stateless design.** This component is designed in such a way that it depends on as few inputs as possible, and remains stateless. Specifically, the inputs are (i) the incoming data readings package, (ii) the patient record (for patient history), and (iii) the means to send out notifications.

We can foresee the following problems with not explicitly modularizing such assumptions in the architectural descriptions:

- As this is no explicit architectural driver, it is not recorded as part of the architectural documentation (the ADD log). Subsequently, there is a risk of diverging later on from the selected architectural solution, for example when requirements change in a later stage.
- Furthermore, the effects of this design solution are not limited to a single subsystem (the `DSS`), but the assumption affected many elements of the architecture. For example, the modules and techniques for representing, sharing and managing medical knowledge are affected by this decision to separate medical knowledge in the `Medical Knowledge Base`. Furthermore, these effects are not lim-

ited to one architectural view, but the influences of the assumption are visible in many of the architectural views. In summary, this particular assumption has had a crosscutting impact on the software architecture of the patient monitoring system.

4. Conclusion

This paper has discussed the modularity of architectural assumptions, i.e. early (preliminary or real) decisions that affect the architectural solution, but are not motivated directly from requirements, constraints or analysis coming from non-technical stakeholders. We have introduced the key distinction between problem-space and solution-space architectural assumptions, in accordance with the distinction between preliminary, reversible assumptions and actual architectural decisions.

We have discussed and illustrated the main modularity problems in the current state-of-practice and -art of dealing with such assumptions: (i) they are made silently and implicitly, and therefore they are difficult to discover or recover [11, 16], and (ii) they have a profound, sometimes subtle, but often crosscutting impact on the development artifacts involved. Therefore, it is hard to assess their individual architectural impact, and treat them as first-class architectural elements.

There is a potential and promising overlap between the problems targeted in early aspects research and the problem statement of this paper, which in essence targets the limited modularity of architectural assumptions on the one hand, and on the other hand the impact of these assumptions on software modularity in general. In the case of problem-space architectural assumptions, many existing Aspect-Oriented Requirements Engineering (AORE) techniques already focus on discovering and documenting the crosscutting influences across requirements of different types [12, 15, 18]. However, to our knowledge, none make the key distinction between reversible assumptions and actual stakeholder requirements, and provide support for reversing such assumptions. In the case of solution-space architectural assumptions, Aspect-Oriented Architecture Description Languages (AO-ADLs) typically offer advanced composition mechanisms to represent crosscutting concerns, both within specific architectural views [5], and across views [1, 8, 14]. However, these all focus on documenting the final architecture, and not on documenting and managing the individual decisions, influences and assumptions that play an important role in the design process.

References

[1] Colin Atkinson and Thomas Kühne. Aspect-oriented development with stratified frameworks. *IEEE Software*, 20(1):81–89, 2003.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, second edition, 2003.

[3] BraveHealth. Patient centric approach for an integrated, adaptive, context-aware remote diagnosis and management of cardiovascular diseases. `http://distrinet.cs.kuleuven.be/research/projects/showProject.do?projectID=Bravehealth`.

[4] E-hip. E-health information platforms. `http://distrinet.cs.kuleuven.be/research/projects/showProject.do?projectID=E-HIP`.

[5] L. Fuentes, N. Gamez, M. Pinto, and J. A. Valenzuela. Using connectors to model crosscutting influences in software architecture. In *Software Architecture*, volume 4758 of *LNCS*. Springer, 2007.

[6] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Softw.*, 12:17–26, November 1995.

[7] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is still so hard. *IEEE Softw.*, 26:66–69, July 2009.

[8] John Grundy. Multi-perspective specification, design and implementation of software components using aspects, 2000.

[9] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[10] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13:47–55, 1996.

[11] Patricia Lago and Hans van Vliet. Explicit assumptions enrich architectural models. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 206–214, New York, NY, USA, 2005. ACM.

[12] Ana Moreira, Awais Rashid, and João Araújo. Multi-dimensional separation of concerns in requirements engineering. In *RE*, pages 285–296, 2005.

[13] Bashar Nuseibeh. Weaving together requirements and architectures. *Computer*, 34:115–117, March 2001.

[14] Steven Op de beeck, Marko van Dooren, Bert Lagaisse, and Wouter Joosen. Multi-view refinement of ao-connectors in distributed software systems. In *AOSD '12: Proceedings of the 11th international conference on Aspect-oriented software development*, 2012.

[15] Awais Rashid, Ana Moreira, and João Araújo. Modularisation and composition of aspectual requirements. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 11–20, New York, NY, USA, 2003. ACM.

[16] Ronny Roeller, Patricia Lago, and Hans van Vliet. Recovering architectural assumptions. *J. Syst. Softw.*, 79:552–573, April 2006.

[17] Share4Health. Healthcare professional's collaboration space. `http://distrinet.cs.kuleuven.be/research/projects/showProject.do?projectID=Share4Health`.

[18] Dimitri Van Landuyt, Steven Op de beeck, Eddy Truyen, and Wouter Joosen. Domain-driven discovery of stable abstractions for pointcut interfaces. In *LNCS Transactions on Aspect-Oriented Software Development*, volume 9, December 2011.