Decision making in software architecture

Hans van Vliet\textsuperscript{a,b}, Antony Tang\textsuperscript{b}

\textsuperscript{a}VU University Amsterdam, De Boelelaan 1081, Amsterdam 1081 HV, The Netherlands
\textsuperscript{b}Swinburne University of Technology, John Street, Hawthorn, VIC 3122, Australia

\textbf{A R T I C L E   I N F O}

Article history:
Received 15 November 2015
Revised 17 December 2015
Accepted 11 January 2016
Available online xxx

Keywords:
Software architecture
Design decisions

\textbf{A B S T R A C T}

Traditionally, software architecture is seen as the result of the software architecture design process, the solution, usually represented by a set of components and connectors. Recently, the \textit{why} of the solution, the set of design decisions made by the software architect, is complementing or even replacing the solution-oriented definition of software architecture. This in turn leads to the study of the \textit{process} of making these decisions. We outline some research directions that may help us understand and improve the software architecture design process.

© 2016 Elsevier Inc. All rights reserved.

\section*{1. Introduction}

There are a great many definitions of the notion of a Software Architecture. Many of the early definitions focus on the end result of the architecture design process: the solution, the global design, represented as a set of components and connectors (Garlan and Shaw, 1993; Bass et al., 2013). Since about 10 years, the \textit{why} of the solution: the set of design decisions, the rationale for these decisions, and the knowledge captured by those decisions, has become prominent, and to many replaced the solution-based definition of Software Architecture. The first such definitions appeared in Perry and Wolf (1992) and then in Bosch (2004).

We may observe a steady increase in the number of papers that take architectural design decisions as starting point. A systematic literature review of the topic by Tofan et al. (2014) mentions an increase from about 10 per year around 2005 to more than 25 in 2011.

This naturally also brings us from the result of the architectural process, the set of design decisions, to the actual \textit{process} of making these decisions. How do people take design decisions? Is that a rational process, whereby people choose the best option at each stage, or is the process a lot more flimsy? To be more precise about the flimsiness: could software architects possibly be biased or make decisions without careful considerations? A further question is the \textit{what} architects take decisions about. There is an ongoing debate about the intertwining between requirements and architecture. We may argue what these decisions are about: are they about the solution - the architecture, or about the problem - the requirements? Finally, architecting involves a number of decisions. Can we say something about the order in which they are made? Or should be made? Is there anything special about the first decision(s)?

In this paper, we explore these aspects further. In doing so, we draw parallels with decision making in other fields. We sketch research directions we may pursue, to increase our understanding of what makes up the architecture design process, and ultimately improve the architecture designs delivered.

\section*{2. Background}

In this section we provide some background and wider context. We first discuss the history of decision making and rational management in software architecture. Next, we touch upon team issues, a topic not further explored in this paper. Finally, we shortly discuss the notions of architecturally significant decisions and requirements, and the wicked character of software design.

\textbf{Decision making and argumentation} in software design can be traced back to the 70’s and 80’s. Methods such as IBIS, QoC and DRL were proposed then to identify issues and capture design arguments (see Moran and Carroll, 1996). IBIS (Issue Based Information System) guides the identification, structuring, and settling of issues raised by problem-solving groups (Kunz and Rittel, 1970). Subsequently, the understanding of planning and design as a process of argumentation (of the designer with himself or with others) has led to the use of IBIS as a design rationale. The elements of IBIS are issues (or questions that need to be answered), each of which is associated with alternative positions (or possible answers). These in turn are associated with arguments which support or object to a given position (or another argument). In the course of the treatment of issues, new issues come up which are treated

---

* Corresponding author.

E-mail addresses: hans@csc.vu.nl (H. van Vliet), atang@swin.edu.au (A. Tang).

http://dx.doi.org/10.1016/j.jss.2016.01.017

0164-1212/© 2016 Elsevier Inc. All rights reserved.

Please cite this article as: H. van Vliet, A. Tang, Decision making in software architecture, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.01.017
likewise, Jeff Conklin and co-workers adapted the IBIS structure for use in software engineering, creating the gIBIS (graphical IBIS) hypertext system in the late 1980s (Conklin and Begeman, 1988). QoC (Questions, Option, Criteria) is a well-known notation for design space analysis (see MacLean et al., 1991). DRL (Decision Representation Language) has been implemented in a system called SYBIL which provides a graphical user interface for design rationale (Lee, 1991).

In Software Architecture, research into design rationale was rather quiet till after 2000. Burge's work on design rationale goes back to 2000 (Burge and Brown, 2000). She developed SEURAT (Software Engineering Using RATIONale), a system to capture, and check for completeness and consistency, of design rationale (Burge, 2005). De Bruin et al. (2002) discussed the need/advantage of explicitly documenting design decisions. They capture architectural knowledge in a graph structure which is used to document and reason about architectural trade-offs. Tyree and Akerman's paper on architecture decisions is from early 2005, but is based on an existing standard that had been in use for quite a while already (Tyree and Akerman, 2005). The “Software Architecture in Practice” book states that a software architecture manifests the earliest design decisions (Bass et al., 2013). In subsequent years, many other research tools have been created to capture software design decisions (Capilla et al., 2015) as well as relations between design decisions (Kruchten, 2004). Despite these works, there is little investigation on how and why design rationale works, and whether it always helps.

Several researchers investigated the fundamentals of software decision making. Zannier investigated the role of naturalistic and rational decision making (Zannier et al., 2007). They found the structure of the design problem determines which aspects of rational and naturalistic decision making are used. The more structured the design decision, the less a designer considers different options. A workshop was organized to study how software designers design (Petre and Van Der Hoek, 2013). Three pairs of professional software designers designed a solution for the same problem. They talked aloud, the design sessions were videotaped and next analyzed from a variety of perspectives. At a more general level, a systematic literature review reveals that there have been over 250 research publications since 1997 on cognitive and human behavioral aspects in software engineering (Lenberg et al., 2015).

Often, a design is made by a team. Consequently, team issues can influence decision making next to the more individual issues explored in this paper. Group decision making, through biases such as Groupthink, where reaching a consensus becomes the main driver, can influence the design process and its outcome (Smitthi Rekha and Muccini, 2014). A nice variation hereof is the Abilene paradox, where one person makes a proposal, which is then followed by another person, but eventually results in disappointment for both. The first person intends his proposal as an open suggestion, and the second person does not want to be impolite, and consents. But in fact, both persons do not like the idea (Harvey, 1974). To mitigate biases like these, some organizations assign a specific devil's advocate role in architecture teams. When talking about design decisions, we usually mean the set of architecturally significant design decisions. A Software Architecture captures many design decisions, but not all of them are architecturally significant. Consider the famous case of the Mars Climate Orbiter, where one component used English units for its output (inches) while another component expected metric units. As a result, never the twain shall meet. Of course, the choice for which unit to use is an important one, but it is not architectural: the architecture will not be different if another choice is made. We may also state this decision as a requirement as to which metric to use. So, in a similar vein, we may distinguish architecturally significant requirements (ASRs) as a subset of the total set of requirements. Chen et al. (2013) provide a framework for systematically characterizing ASRs, based on an empirical study of practitioners. Bass et al. (2013) also give guidelines on how to identify ASRs.

There are other fields where the intricate, complex nature of design problems has been noticed. One is the design as a wicked problem characterization from Rittel and Webber (1973). This originated in the social planning field, but is now increasingly adopted as a characterization of the software design field (Yeh, 1991; Dutoit et al., 2007; van Vliet, 2008). Properties of wicked problems in social planning are remarkably similar to properties of software design:

- A wicked problem has no definite formulation. Likewise, a software design process can hardly be separated from other processes such as requirements engineering.
- A wicked problem has no stopping rule; there is no criterion that tells us when a solution has been reached.
- The solution to a wicked problem is not true or false. At best, it is good or bad.
- Every wicked problem is a symptom of another problem. Resolving one problem may well result in an entirely different problem elsewhere.

3. Is software architecture decision making a rational process?

Many argue that decision making is not a rational process, and that people stop reasoning as soon as a satisfactory solution is found. This typical behavior is discussed in Section 3.1. A very similar characterization uses the term intuitive or naturalistic decision making, as discussed in Section 3.2. Which type of decision making we use seems situated, as discussed in Section 3.3.

3.1. Bounded rationality and satisficing behavior

There are two (global) ways to look at decision-making: rational and bounded rational. Rational decision making is where we carefully consider all options and then choose the best one at each stage: rationality is optimization, also known as Rational Choice Theory. Many people think this is what we do. Many people hope this is what we do. But there is quite strong evidence that reality is different. In real life, people are irrational, taking decisions in split seconds, and so on. Popular books such as those of Kahneman (2011), Ariely and Jones (2008), or Gladwell (2007) give many examples hereof. The Nobel laureate and AI guru Herbert Simon has coined the term bounded rationality: in decision-making, rationality of individuals is limited by the information they have, the cognitive limitations of their minds, and the finite amount of time they have to make a decision (Simon, 1996).

Another way to look at bounded rationality is that, because decision-makers lack the ability and resources to arrive at the optimal solution, they instead apply their rationality only after having greatly simplified the choices available. Thus the decision-maker is a satisficer, one seeking a satisfactory solution rather than the optimal one. Some evidence that software designers exhibit satisficing behavior is given in (Tang and van Vliet, 2015). In that study, we gave some participants six scenarios, each with a conclusion. The participants were asked to indicate the degree to which they agreed with the conclusion, and indicate issues they saw with the scenarios, issues that influenced their conclusion. Most participants gave few reasons before they made their decisions. When asked if how much they agree or disagree with the conclusions, most participants did not fully commit to their disagreements. We noticed that a few of the participants (4) behaved markedly different from all the others (68). These 4 outliers on average identified twice as many issues as all the others.
We next analyzed these 4 outliers in more detail. We had asked participants to write down their reasons. From these texts, we derive the following. First, these outliers use a lot less analogy. Example analogies that satisfiers used include “recharging would be a problem with smart cards in general”, “a system like this works in the Netherlands, so it is possible to implement this in any country”. The use of analogy simplifies the problems, decreases cognitive load, uses superficial similarities, and may easily overlook context-specific critical issues. The four non-satisfiers in our study only use one analogy, and this analogy is supported by an explanation. They identified many more issues. These reasons could only be found through careful reasoning, since most of the scenarios were unfamiliar to most of the participants. This study shows that few software designers are non-satisfiers and they take a rational approach to decision making. This finding agrees with the Law of Least Effort which states that people tend to minimize cognitive load and tend to use intuition in making decisions (Kahneman, 2011).

3.2. Rational thinking vs intuitive thinking

Some researchers make a distinction between rational thinking and intuitive thinking (rather than bounded rationality), Kahneman uses the terms System 1 and System 2 thinking. System 1 is fast, instinctive and emotional, and evolutionary very old. System 2 is slower, more deliberative, and more logical, and evolutionary more recent. Kahneman (2011) is full of examples and experiments to show that people tend to resort to System 1 thinking in everyday life.

In a similar vein, Klein suggests two types of decision making: naturalistic decision making and rational decision making (Klein, 2009). In naturalistic decision making, rather than quantitatively comparing options, people frequently construct explanations of decisions in the form of stories about possible outcomes. Naturalistic approaches to decision making are more contextually embedded. Naturalistic approaches stress the roles of identity and unconscious emotions in decision making. Decisions in different domains or contexts are related to fundamental differences in subjective evaluations of each outcome, inducing different choice strategies. Decision strategies may include story construction, regret focus, morality focus, choose the favorite, avoid the worst, or other emotional reactions. Rational decision making is based on careful evaluation of problems, options and tradeoffs. Decision makers require knowledge, expertise and facts as basis for analysis and evaluation. It is cognitively demanding and time consuming. Klein argues that we need both decision making styles in different circumstances.

3.3. Which type of decision making to use?

It appears that the use of naturalistic or rational decision making is situated. Zannier et al. found that the structure of a software design problem determines the aspects of naturalistic and rational decision making used (Zannier et al., 2007). The more structured the problem, the more naturalistic decision making was used; the less structured the problem, the more rational decision making was used. Additionally, they also found that designers have satisfactory behavior and do not necessarily always try to explore exhaustively. In an interview, one of their participants said that “... if I'm quite confident that I know of a good solution that does what I need I will just do that and that's a question of experience ... I know that I can do four things here, I don't know where any of those four things will lead me. So let's just see what happens ...” (Zannier and Maurer, 2007).

Ball suggested that design might be viewed as predominantly top down and structured in nature, and with opportunistic venturing into the depths of a sub-problem when there are doubts about a feasibility issue (Ball et al., 1997). Guindon explained that designers often deviated from a top-down approach and ventured into low-level details in an opportunistic manner (Guindon, 1990). Parnas and Clements suggested that we cannot have a totally top-down approach and be totally rational about it, but a rational design can be falsified by documenting what options are available and why a particular design option is chosen (Parnas and Clements, 1985).

There is a place for both intuition and careful reasoning, but, especially in software design, one has to be careful when resorting to analogy and intuition. All too often, the context is different from what one is used to. When faced with an unfamiliar context, there can be many details that a designer needs to know, and making naturalistic or intuitive decisions can be risky.

4. Is the software architect biased?

Following the argument that software architecture decision making is not entirely rational, and certain intuitions and naturalistic decisions may be influenced by emotions and other cognitive factors, we need to examine the influences of cognitive biases.

Cognitive bias is the general term, introduced by Kahneman and Tversky (1972), to denote humans inability to reason in a rational way. They are cognition or mental behaviors that prejudice decision quality in a significant number of decisions for a significant number of people (Arnott, 2006). Kahneman, Tversky and their colleagues demonstrated several replicable ways in which human judgments and decisions differ from rational choice theory. They explained these differences in terms of heuristics; rules which are simple for the brain to compute but introduce systematic errors. For instance the availability heuristic (also termed recall), when the ease with which something comes to mind is used to indicate how often (or how recently) it has been encountered. These experiments grew into the heuristics and biases research program which spread beyond academic psychology into other disciplines including medicine and political science. It was a major factor in the emergence of behavioral economics, earning Kahneman a Nobel Prize in 2002.

4.1. Anchoring bias

Let us discuss an obvious type of bias, anchoring. Decisions are made in a context. You choose this context, but you often do so kind of automatically. Once you have chosen a context, you make decisions from there. The context chosen serves as an anchor, so to say. An example of biasing we often use in class is ranking your teacher. If one first asks for the last digit of one’s phone number, and next to grade the teacher, there is a 0.5 correlation between the two. The first piece of information serves as an anchor, relative to which further decisions are made. If this works for grading, would it work for software architects too? The answer is yes. We have been teaching software architecture for a number of years, and each year we ask students to make a first guess at their solution in week 1. In week 2, we then discuss the solutions, and we try to figure out the why of their solution. We inevitably have a sizable number of students with a SOA solution, since there is a SOA class in the period just before the software architecture class.

A second example of anchoring bias in software development was mentioned by a Dutch architect, Robert Deckers.1 He had to devise a software architecture for a copier, right after finishing university in 1992. He had heard all about OO, so he designed a beautiful OO architecture for a copier, with major components for handling originals, doing the printing, finishing (stapling etc). Once he

---

1 Personal communication
had done this he realized there is a need for detecting and locating paper jams, giving users proper instructions, hold processing when the printer is out of toner, and so on. In reality, well over 75% of the code is error handling. A printer really is an error handler, and a good design reflects this dominant aspect. Anchoring the design on using an OO framework was probably ill-considered.

4.2. Framing bias

Another type of bias, framing, is discussed in a software engineering context in Mohanani et al. (2014). Framing is the tendency to give different responses to problems that have surface differences, but are really formally equivalent. In Mohanani et al. (2014), participants were asked to make a design for a particular system. One group got a description which used the words requirements specification, the other group got the same assignment, but now the requirements were phrased less compulsory, less thou shalt like, more as a list of ideas. The resulting designs were next compared. It turned out that the second group came up with many more original ideas. They were less fixated on satisfying the requirements. By using the mere words requirements specification, people are inclined to think that satisfying those is the only thing that matters. This of course relates to other issues around requirements engineering and its relation to architecture: from MOSCOW types of requirements categorization (Must haves, Should haves, Could haves, Won’t haves) to questions whether the 99.9999% availability is really needed.

4.3. Confirmation bias

Stacy and MacMillan (1995) described how biases show up in day-to-day software engineering activities. They described a case of confirmation bias in which software testers are four times more likely to choose positive test cases than to choose negative test cases, implying that they are more inclined to show that the software works. Tang (2011) described confirmation biases he observed in the software industry. For instance, a project manager said “I’ll add more people to get the job done”. A project manager wanted to complete a job and as such made himself believe, i.e. a confirmation bias, that he could accomplish the task by adding people.

4.4. Other cognitive biases

Arnott (2006) gave a list of 37 cognitive biases in software development, and found that decision makers suffer from these biases. He also found that the use of a debiasing method can help decision makers to recognize their own biases and make corrections.

From our excursion into rational/irrational design decisions, we may conclude that software design is not as easy, straightforward, and rational as we may think. Biases do play a role. And we probably cannot fully prevent them from occurring; they are simply too human.

5. What do we make decisions about?

In the traditional software engineering school, there was a rather strict distinction between requirements and design/architecture. First there are requirements. These are then cut in stone, and then the design/architecture is devised. In a more realistic setting, it is being realized that requirements do change, and in many cases are unclear and need refinements as design progresses, so some sort of iteration is required. But we may question whether the distinction between the two is all that clear?

The Twin Peaks model (Nuseibeh, 2001) draws attention to the synergetic relationship between requirements and architectural design. It emphasizes the need to progressively discover, refine and specify requirements while concurrently exploring alternative architectural decisions. But the general idea then still is that requirements and architecture are considered as different things, where requirements originate with end users and the business, while the architecture is decided upon by the architect, and one merely hops back and forth between the two. We argue they are more intricately connected. In many cases, when the architect makes a decision, this leads to further issues to be dealt with. While dealing with those issues, she is not only defining the solution, but she is also shaping the problem to be solved. When the business is involved in these discussions, business and architect together define both problem and solution. De Boer and van Vliet (2009) argue there is no fundamental distinction between architecturally significant decisions and architecturally significant requirements. They are two sides of the same coin.

In the more general design field (e.g. design of physical artifacts, building architecture, and the like) there is a well-known model known as problem-solution co-evolution (Dorst and Cross, 2001). Here, creative design is the development and refinement of the formulation of the problem and parts of the solution together. Not only is the design a series of design decisions, but each design decision may also lead to further requirements that need to be dealt with. So, defining a solution and defining the problem go hand in hand, they proceed at the same time, rather than one preceding the other. We not only make decisions about which solution to choose, we also make decisions about which problem to solve.

Fig. 1 is an example from a workshop on software design, where different teams were asked to design the same system (i.e. a traffic-simulation system). The designers were asked to think aloud, everything was videotaped, and given to researchers to analyze. We were participants of the workshop, and analyzed the videos. We studied the intertwining of problem exploration and solution generation. Fig. 2 shows the co-evolving and intertwining activities on design exploration, problem identification and solution development of one such team. The horizontal axis denotes time. This picture captures approx. 1.5 h of design activity. The vertical axis denotes the number of problem/solution type statements per minute. What strikes in this picture is the alternation between problem and solution utterances. Decisions are made as the designers explored what problems to tackle, what new requirements were uncovered and what solution options to consider and pick (Tang et al., 2010).

6. What about the first decision?

If problems and solutions co-evolve, and if a design decision may lead to new requirements, then the order in which decisions are taken may have an impact on the result (or, rather, the problem that is being solved). In particular, the first decisions may be particularly important.

In the traffic-simulation study mentioned above, Team A very early on decided the design was an MVC problem. They focused on data structure, modeling, representation. They did so for about 40 min. Another team, Team M, took a different design approach. This team of designers first explored the design space, and they made decisions on how to approach the design problems (see their
explorations in Fig. 2). After their initial explorations, they classified the design problem as one of a canvas to draw the simulation.

The first decision made by Team M led them to follow a path that was based on the MVC solution mindset. However, such an early commitment left out many other design considerations at that point. With MVC anchored as the framework, the designers were restrained to fitting the rest of their solutions based on it. It led to a lot of context switching between design issues (Tang et al., 2010).

This early framing of a design problem is known as “design fixation”. It is a broad term to denote the early commitment to a particular representation of the design problem or a solution to that problem. Design fixation effects have been described for many fields (Crilly, 2015; Gero, 2011), including software development, and seems to be influenced by such factors as experience and prior expertise. These first decisions tend to lead designers to a certain design path, which is afterwards hard to get away from. Tracz (1979) for instance observed that software developers fail to see the shortest solution to a problem because of a fixation to one approach to solving a problem. Hassard et al. (2009) interviewed user interface designers, who reported heavy use of analogies in the early stages of their designs, which were later modified extensively to fit the problem, rather than searching for a fresh direction.

7. Conclusion

The main purpose of this paper is to point out that there are fundamental issues that influence software architecture design. These issues are concerned with making design decisions. They include bounded rationality, all sorts of cognitive biases, the decisions about which problem we are trying to solve, and the role of the first decisions being taken. Although these issues are not the design artifacts we usually focus on in software engineering, they can adversely affect the quality of software architecture design. The list of issues we discuss is not exhaustive. They derive from our experiences as software architect, our involvement in a workshop studying professional software design (Petre and van Der Hoek, 2013) and our previous research in several projects involving software architecture topics.

The first two topics we discussed, bounded rationality and biases, have been heavily researched in many fields, from policy making to planning and economics. The consensus is that one cannot prevent them from happening. We believe the same is true for software architecture design. But we may try to point designers to these traps as early as possible, and equip them with supporting thinking tools and techniques. We suggest a few research areas to explore to mitigate the fundamental issues:

Please cite this article as: H. van Vliet, A. Tang, Decision making in software architecture, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.01.017
• Paying explicit attention to decisions in architecture assessment, as for instance done in van Heesch et al. (2014).
• Using checklists that aim to reveal biases, as in Kahneman (2011).
• Asking reflective questions during architecture design such as (Razavian et al., 2015).

Software architecture decision making research can profit from results obtained in other domains, but has to take into account specific aspects of our field, such as the amount, complexity and connectivity of decisions, the multitude of stakeholders involved, and the inevitable evolution of requirements. The extent to which these influence tools and techniques is as yet unknown.

The last two topics discussed, decisions about which problem to solve, and the role of the first decisions therein, have been mostly researched within other engineering domains. In particular, the role of design fixation has been extensively studied there, for instance possible differences in design fixation between experts and beginners, and ways to detect design fixation early. We propose tailored research, along similar lines, in the software architecture design field. Again, specific properties of our field may well influence the results.

We do not have a design process recipe that can guarantee success. It may be overly optimistic to assume that any prescriptive design methodology can guarantee success. However, further research into various aspects of decision making, such as cognitive and social aspects, can improve our understanding of the success factors. This understanding may result in methods and tools that better equip designers to reflect on their own thinking process and practice software design wisely. Empirical research into the adequacy of these methods and tools is needed. Researching the different circumstances empirically of how decision making elements work can facilitate their application and, hopefully, improve software architecture design.

Acknowledgments

We gratefully acknowledge constructive comments to an earlier version of this paper by Barbara Paech and Tom-Michael Hesse. We also acknowledge the insightful comments made by the anonymous reviewers of this article.

References

Kahneman, D., Tversky, A., 1972. Subjective probability: a judgment of representa-
Klein, G., 2009. Streetlights and Shadows: Searching for the Keys to Adaptive Deci-
Tang, A., 2011. Software designers, are you biased? Proceedings of the 6th Interna-
Tang, A., 2011. Software designers are you biased? Proceedings of the 6th Interna-

Please cite this article as: H. van Vliet, A. Tang, Decision making in software architecture, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.01.017
Hans van Vliet is Professor in Software Engineering at the VU University Amsterdam, The Netherlands, since 1986. He got his PhD from the University of Amsterdam. His research interests include software architecture, knowledge management in software development, global software development, and empirical software engineering. Before joining the VU University, he worked as a researcher at the Centrum voor Wiskunde en Informatica (CWI, Amsterdam). He spent a year as a visiting researcher at the IBM Almaden Research Center in San Jose, California. He is the author of “Software Engineering: Principles and Practice”, published by Wiley (3rd Edition, 2008). He is a member of IFIP Working Group 2.10 on software architecture, and the Editor in Chief of the Journal of Systems and Software.

Antony Tang is Associate Professor in Swinburne University of Technology, Australia. He received a PhD degree in Information Technology from Swinburne in 2007. Prior to being a researcher, he had spent many years designing and developing software systems. His main research interests are software architecture design reasoning, software development processes, software architecture knowledge management.

Please cite this article as: H. van Vliet, A. Tang, Decision making in software architecture, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.01.017