

# Software Designers, Are You Biased?

Antony Tang  
VU University Amsterdam  
Department of Computer Science  
Amsterdam, the Netherlands  
atang@cs.vu.nl

## ABSTRACT

Methods of representing and capturing design rationale have been studied in past years. Many meta-models, methods and techniques have been proposed. Are these software engineering methods sufficient to help designers make logical and appropriate design decisions? Studies have shown that people make biased decisions, software designers may also be subjected to such cognitive biases. In this paper, I give an overview of how cognitive biases and reasoning failures may lead to unsound design decisions. I conjecture that in order to improve the overall quality of software design, we as a community need to improve our understanding and teaching of software design reasoning.

## Categories and Subject Descriptors

D.2.1 [Requirements / Specifications], D.2.2 [Design Tools and Techniques].

## General Terms

Design, Human Factors.

## Keywords

Software Design, Decisions, Reasoning, Cognitive Bias.

## 1. INTRODUCTION

Software design was said to be a wicked problem by Rittel and Webber back in the 70's [25]. They suggested that there is no well-defined set of potential solutions. A solution is very much the results of what the designers devise. As such, the skills of a designer are very important and design is a handicraft that depends on this skill. The software industry has put much emphasis on the use of processes, methodologies and techniques to assure the quality of software design. Despite the common adoption of these software engineering processes and methodologies, it is still common to see sub-optimal and non-functioning software design. Why? It is because individual designers can still make bad design decisions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SHARK'11, May 24, 2011, Waikiki, Honolulu, HI, USA  
Copyright 2011 ACM 978-1-4503-0596-9/11/05 ...\$10.00

The software engineering community has recognized that decision making is a key element in designing quality software. We have seen two waves of research activities on this subject. The first wave was in a period between the 70's and 90's. The research focus can be typified as argumentation-based design rationale. Notable methods proposed in this period were IBIS [19], QOC [22] and DRL [21]. However, Shipman III and McCall [13] concluded that neither the argumentation nor the communication perspective of argumentation-based design rationale has been generally successful in practice. The second wave of design rationale research began from around 2003. The focus in this period can be typified as practice and knowledge management oriented. Notable works focus on the practice of software architecture design [4, 6] and the capturing of design decision [35]. Meta-models such as CORE [8] and SOAD [37], tools such as SEURAT [7], ADDSS, Archium, AREL, Knowledge Architect and PAKME were proposed (summarized in [30]).

Doubtless these methods would provide an orientation for a designer to justify his design decisions, but design decision making remains an internal thought process, and it relies on the ability of a person to reason. If the reasoning process fails or the basic information (i.e. we call premises) used for reasoning is untrue, then the resulting design decision is likely to be wrong. Using the research results of other disciplines such as cognitive psychology, I suggest that there are three fundamental causes of design reasoning failures:

- Cognitive Bias – a distortion of judgment in particular situations due to psychological effects and insufficient regards of probability [17]. For example, software designers select inappropriate design solutions because they are more familiar with these design solutions, despite that the solution is ill-suited to solve the problem. A phenomenon we call “I have a hammer, and everything is a nail”.
- Illogical Reasoning – a logical reasoning requires that the basic premises of a decision, in our case design concerns and requirements, should be factual and true; and the arguments and inferences to reach a design conclusion should be sound [15]. Quite often software designers make design decisions without regards for whether the premises are true or the conclusions are reasonable. Sometimes the design problems are ill-defined. When asked to justify his decisions, it is not uncommon to see designers make retrospective justifications.
- Low Quality Premises – in large software projects in which multiple designers make decisions, failures to represent the premises truly, i.e. inaccurate or inadequate premises, can cause incorrect decisions. For instance, if an assumption is

unclear, that could cause a designer to make a decision based on faulty inputs.

Reasoning failures can lead to wrong or sub-optimal design decisions. Therefore it is important that we understand the fundamental causes of reasoning failures in order to find ways to improve the situation.

This paper intends to serve a few purposes. Firstly, it provides an overview of possible causes of design reasoning failures (Section 2). Secondly, it proposes to use a logic framework to support argumentation in design reasoning (Section 3). Thirdly, related work in this area is discussed and techniques are suggested to improve design reasoning (Section 4). This area of research is in its infancy as far as software engineering is concerned. We as a community should consider the challenges and the potentials that it may bring (Section 5).

## 2. DESIGN REASONING FAILURES

In spite of many successful and well designed software systems, there are also many less successful ones. Amongst these two sets, many of the development projects make use of software development processes, methodologies and quality reviews. So software design and implementation with the aid of software engineering processes and methodologies do not guarantee success. Their use is probably a necessary but insufficient condition to developing quality software.

This view is supported by having spent some years developing software and managing software teams. I notice that besides the appropriate use of processes and methodologies, often the right *soft-skills* of the designers seem to be a key success factor. The *soft-skills* are about how a good software designer is willing to gather as much relevant information as he can, listening to users and developers, weighing the relative merits of contradicting solutions, and so on. These soft-skills appear to help software designers succeed. In this section, we consider a soft-skill which is about design reasoning and cognitive biases.

### 2.1 Cognitive Biases

Cognitive bias was first introduced by Kahneman and Tversky [16]. The idea is that human judgments can be distorted by an intuitive bias when the representativeness heuristic is not used correctly. Let us consider an example, someone driving during peak hours blames that he is unlucky because he always stops before a red light, especially when he is in a hurry. If the driver considers the slow traffic movements and the probability of cars stopping before a red light in peak hours, this is hardly surprising. The driver uses subjective probability in his judgment. It shows that human tends to judge without regards to the likelihood of events. Therefore biased thinking generates systematic judgments errors.

This phenomenon is often encountered in software design and software projects. For instance, a programmer thinks that he can complete a program in  $X$  days. But programmers have a tendency to underestimate their efforts. It may be that programmers would like to demonstrate their competency by claiming that they can finish a task early. In fact, delayed completion seems to happen more often. One would think that after a few times, the estimations by a programmer would improve. From observations, I find that it generally takes some time before such cognitive bias

is adjusted. Some software designers also suffer from cognitive biases. Claims such as the following are often encountered in design discussions:

- “I’ll add more people to get the job done”
- “OOP and Java is the programming language of choice”
- “A centralized database approach is the best because it works for us last time”

These claims may be correct under some specific circumstances but often they are taken in at face value and lead to decisions without further evaluation. These claims may not be supported by any facts. Does the software designer know from past experience that adding more people to a project can help speed up a project? Which development phase is applicable to this claim? Is the technical environment a relevant issue to consider? Are there any facts or proofs to support this claim? If these claims are based on intuitions instead of facts, then the subjective view may bias the judgment. There are many kinds of cognitive biases that affect design decisions. Some examples can help illustrate them:

- Egocentric bias – I am always right and I will argue ideas that are not invented or suggested by me.
- Projection bias – a tendency to project that the outcomes will be alright.
- Consistency bias – a false impression of what a person was actually thinking about a past event.
- Last Experience Bias – what happened recently has a stronger influence on upcoming decisions.
- Anchoring – the first impression of a solution that comes to mind anchors, and it may be difficult to adjust or change even when there is evidence to show that the initial solution is inferior [10].

The above list is not exhaustive, it is important that the software community recognizes such biases and their impacts on design.

### 2.2 Illogical Reasoning

Some designers make decisions based on personal preferences and habits. It has been suggested in [11, 12] that there are two distinct cognitive systems underlying reasoning: *System 1* comprises a set of autonomous subsystems that react to situations automatically. They enable us to make quicker decisions with a lesser load on our cognitive reasoning. *System 1* thinking can introduce belief-biased decisions based on intuitions from past experiences; these decisions require little reflection.

*System 2* is a logical system that employs abstract reasoning and hypothetical thinking, such a system requires longer decision time and it requires searching through memories. *System 2* permits hypothetical and deductive thinking. Under this dual process theory, designers are said to use both systems. It seems that, however, designers rely heavily on prior beliefs and intuition rather than logical reasoning.

An explanation is offered by Simon [28]. In his bounded rationality theory, he suggests that there are limits upon the ability of human beings to adapt optimally to complex environments. This is due to the limitation in short-term memory for problem solving [27]. Thus problem solvers, in our case designers, cannot

evaluate all the rational options to find an optimal solution, instead they settle with an acceptable solution.

### 3. LOGICAL DESIGN ARGUMENT

In the last section, we see issues that adversely influence the decision making. These issues are related to getting the basic facts correct or getting the arguments correct. In this section, we explore argumentation with a logic framework. I conjecture that such a logic framework helps to frame a mindset, and it facilitates systematic and explicit logical reasoning in design.

#### 3.1 Basic Form of an Argument

The aim of logic is to develop a system of methods and principles that we may use as criteria to evaluate arguments [15]. An argument consists of one or more premises that claim to support a certain conclusion. Let us consider this example:

- The system shall disseminate information through the Internet (premise 1)
- Any users who has access to the Internet and a common browser shall be able to view the data provided by the system (premise 2)
- There shall be no other ways to access the system (premise 3)
- Therefore users will access the system through its web pages (conclusion)

The premises have dictated the conditions such that there are no other choices but to use standard web pages. The conclusion is a logical deduction from the given requirements. Assuming that these facts are true, there can be only one conclusion as there are no other possible access methods. For instance, if we delete *premise 3*, a condition is relaxed. The requirements in premise 1 and 2 only specify Internet access, but it does not specify whether any other access method is allowed. It is ambiguous about that, so a designer could choose to add a thick client implementation on top of web pages.

There are two kinds of arguments, *deductive argument* and *inductive argument*. A *deductive argument* is an argument in which the conclusion is claimed to be impossible to be false if the premises are true and the argument is valid. The conclusion necessarily follows from the premises.

An *inductive argument* is an argument in which it is claimed that the conclusion is *improbable* to be false if the premises are true and the argument is strong. The conclusion is not an absolute certainty but a probability. Inductive reasoning is about reaching a conclusion through generalization of known experiences. An example of inductive reasoning is:

- We have 10 model X disk arrays and none has failed in the last 2 years
- We will install another model X disk array
- This new disk array will probably not fail for the next 2 years

The general form of an argument, deductive or inductive, is shown in Figure 1. It comprises of three parts: (a) premises which are statements to support the inference; (b) a logical argument based on the given premises; (c) a conclusion. A conclusion of a deductive argument can be sound or unsound, depending on the

validity of the argument. On the other hand, a conclusion from an inductive argument can be cogent or uncogent, depending on the strength of the argument.

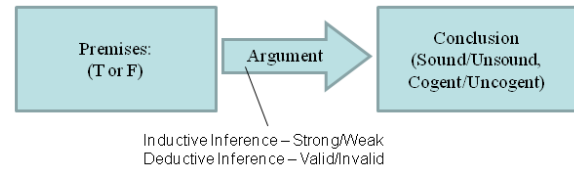


Figure 1. General Form of an Argument

In deductive reasoning, not all conclusions are sound. Conclusions can be unsound for a number of reasons, either because the arguments are invalid or the premises are false. Table 1 shows examples of valid and invalid deductive arguments<sup>1</sup>. If any of the premises used in the argument is false, then the conclusion will be unsound. For instance, if we define any programming language that works on the Internet is a web program then AJAX is one of them. However, in row four of Table 1, it is false to claim that all web programs are user interfaces (U/I). AJAX is a web program but it is not concerned with displaying contents. Therefore it is not a U/I.

Table 1. Deductive Arguments - Software Examples

	Valid Arguments	Invalid Arguments
True Premises, True Conclusion	All U/I design must follow corporate U/I design standards. The web pages are U/I. Therefore, the web pages design must follow corporate U/I design standards. [sound]	All U/I design must follow corporate U/I design standards. The web pages follow corporate U/I design standards. Therefore, the web pages are U/I. [unsound]
True Premises, False Conclusion	None exist	All system functions must satisfy performance criteria. The system configuration is a system function. Therefore, it must satisfy performance criteria. [unsound]
False Premises, True Conclusion	All web programs have a U/I. My AJAX Program is a web program. Therefore My AJAX Program has a user interface. [unsound]	All web programs are U/I. My HTML Program is a U/I. Therefore My HTML Program is a web program. [unsound]
False Premises, False Conclusion	All web programs are U/I. My AJAX Program is a web program. Therefore My AJAX Program is a U/I. [unsound]	All web programs are U/I. My AJAX Program is a U/I. Therefore My AJAX Program is a web program. [unsound]

In an inductive argument, an argument can be cogent if it is a strong argument. In other words, the probability that a conclusion will be right is higher. However, if the premises are false or the argument is weak, then the conclusion is uncogent. Table 2 shows different possible scenarios. Let us examine row 1 in Table 2. A strong argument is the one that has little room for doubt, if the performance of an application is satisfactory on a certain machine, and the performance is characterized by certain data volume and

<sup>1</sup> The structure of Table 1 and 2 comes from [14].

usage; then we can infer that another application with similar characteristics will have satisfactory performance also.

The argument in row 1 (right column) of Table 2 is weak. Although it is true that technically SAP and Oracle can run on the UNIX operating system, and it may be true that this SAP application should run on UNIX, the choice of an operating platform for this SAP application requires many other considerations. Ignoring these considerations (or premises) in the argument weakens this argument.

**Table 2. Inductive Arguments - Software Examples**

	Strong Arguments	Weak Arguments
True Premises True Conclusion	All apps. of similar data volumes and usage have satisfactory performance on this machine. This application has similar data volumes and usage to those apps. Therefore, this application probably has satisfactory performance on this machine. [cogent]	Oracle runs on UNIX. SAP uses Oracle. Therefore this SAP application should run on UNIX. [uncogent]
True Premises False Conclusion	None exist	Some banking solutions require 24X7 uptime. Therefore the CRM system requires 24X7. [uncogent]
False Premises True Conclusion	All software systems were written in OO language. Therefore, our next system will be developed in a OO language. [uncogent]	All modern systems are developed in 4GL. Therefore we need to select a graphical user interface tool. [uncogent]
False Premises False Conclusion	All past requirement specs were complete and flawless. Therefore our next requirement spec. will also be complete and flawless. [uncogent]	All past requirement specs were complete and flawless. Therefore our programs will also be complete and flawless. [uncogent]

### 3.2 Quality Premises

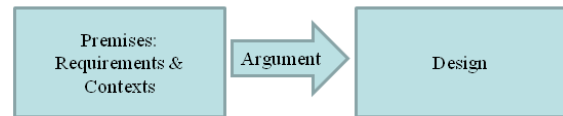
Reasoning depends on the quality of the premises. In software engineering, the basic premises are typically the requirements, quality attributes, characteristics of designed components, system environments and other design contexts (Figure 2). If this information is inadequate or inaccurate, the arguments and the conclusions designer reach would be doubtful.

Since the validity of the final design outcomes depends on the quality of the premises, we will need to establish some criteria to evaluate them. Adequacy and accuracy are two main criteria, but some other criteria are also important:

- Accuracy – designers should be prudent to ensure that the premise is true and accurate, and not a personal belief. For instance, a statement such as “*functional programming is the best programming language*” is a belief. To make it truthful, this statement must be substantiated.
- Adequacy – incomplete or missing premises can lead to invalid/uncogent conclusions. Consider this argument, “*The software service is instantiated once and the process remains in the memory, so the performance will be good*”. There is a premise that is missing in this argument, and that is how many

users would use this service simultaneously. Without such a premise, the conclusion is uncogent.

- Clarity – room for making assumptions about the premise. To avoid unclear premise, designers should explicate and anticipate any assumptions that may affect decision making. There are two suggestions to deal with clarity issues: (a) if an assumption may arise because of the lack of knowledge about something, then either elaborate or make a note where more information can be obtained; (b) if an assumption arises because there are unknowns, then state the risk of the unknown or suggest ways to clarify the unknowns.
- Relevance – designers should provide some context to explain a premise. Consider this example, “*the operators need to have a simple user interface because they enter 200 transactions per hour, efficiency is key*”. This information provides relevant information to a designer. A designer is aware of the number of transactions and the reason for having efficiency. If this relevant information is not available, a designer can design a simple data entry interface but this user interface is not necessarily efficient.
- Fairness – the relative priority of all the premises in an argument can be important. An unfair evaluation of their relative priority can distort the judgment. Consider this example, “*The corporate design principle dictates that we should always implement server-based programs; although your web page requires complex validation, it still needs to be programmed on the server-side*”. In this example, the relative priority of the server-side programming, a design principle, is implicitly higher than the requirement of web-page performance and usability. To obtain a suitable conclusion, stakeholders and designers must understand their relative priority, especially in tradeoff situations.



**Figure 2. Arguments in Software Design Decision**

The criteria for evaluating the quality of premises are critical to design reasoning. Software engineers and designers should learn and assimilate them into their daily practice. In the software industry, it is common to see requirements specifications and software designs that leave opportunities to be misinterpreted. A real-life system example that I encountered is a modifiability requirement in a system. The person who specified did not elaborate what a modifiable system means. There is little documentation on which parts of the system can be modified and under what circumstances software would need modification. The reviewers of the specifications did not ask for clarifications, neither did the software suppliers. Eventually the supplier designed the system which they thought was modifiable, but this did not match users’ expectations. The system failed because of mismatched assumptions.

### 3.3 Inter-related Design Decisions

A software system is often built by many people. Communicating premises accurately between designers in this environment is a

challenge. In a large system, many decisions are made and these decisions are often interrelated. One decision can impact on the other design decisions in a domino effect. This situation creates two issues:

Firstly, important information that is required for design reasoning may be overlooked by designers, i.e. “*falling through the cracks*”. That means a designer does not find all the necessary premises for design reasoning.

Secondly, in a large design team where many decisions are made by many designers at different times and in different geographic locations, a decision which is relevant to another designer may not be communicated across. This would violate the adequacy criteria.

When decisions are interrelated, designers not only have to make good individual design decisions, they also have to ensure that these design decisions are compatible with others’ decisions. The current industry practice of communicating decisions is inadequate. This issue is concerned with the adequacy criteria. A study conducted recently has shown the different scenarios of this issue [31].

## 4. IMPROVING DESIGN REASONING PRACTICE

Previous research works into the capturing and representation of design rationale have contributed to software engineering practice. I conjecture that the understanding of design reasoning and how designers think [1] will advance the practice further. This understanding of how to think in design will allow us to adapt our working procedures more dynamically according to the design problem we face. In this section, I examine what we currently know about software design reasoning and then suggest some techniques that may help design reasoning.

### 4.1 Software Design Reasoning Research

There are some works on software design reasoning. The topics include design planning, problem structuring, design assumptions and design constraints.

**Problem structuring.** In a study of 25 software designers, Zannier et. al [36] have found that software design is about problem structuring. They have found that if a design problem is well-structured, designers tend to use a rational approach; if a problem is ill-structured, designers tend to use a naturalistic approach, i.e. situation assessment with an evaluation of a single option. This result is similar to what we found by observing two pairs of software designers. More time that is spent on problem structuring helps the effectiveness of design [29]. A possible explanation is the structuring of design problems provides some systems and disciplines to allow designers to arrange and focus on certain aspects of their reasoning thoughts in an orderly way.

A study has found that the strategy of planning may change from breadth-first to depth-first depending on the complexity of the problem. It argues that problem structuring is situation dependent [3]. In another study, Baker & Hoek have found that designers often consider two design subjects simultaneously and repeatedly return to high level subjects [2].

**Problem-Solution Co-evolution.** We have found by observing professional software designers that formulating design problems

are as important as providing solutions. Designers who investigate the problem space more thoroughly use an investigative approach to design. This appears to facilitate design reasoning. On the other hand, a solution-driven design approach tends to anchor on a preconceived idea without challenging why this idea is suitable, or why another option may be better [29]. Such anchoring would limit the exploration of the problem space and the solution space.

**Critical Conversations.** McCall [23] suggests that a process of ideation (i.e. generating design options) and evaluation (i.e. evaluating the pros and cons) in a feedback loop helps to create new ideas. He suggests that this reflection or feedback can occur in two forms: design discourse and actual implementation experience. In essence, the paper suggests that a reflection of design rationale from previous experience can help reasoning in the future. This is one way of acquiring procedural knowledge by reflecting on past designs.

**Explicit Assumptions.** In this study, Lago and Vliet [20] have found that tacit assumptions hamper software design evolution. Assumption is an important element in design reasoning, knowing the assumptions in requirements and design is essential. The paper has classified assumptions by technical assumptions, organizational assumptions and managerial assumptions. They suggest to clearly identify design features that may be affected by the assumptions, and define the dependencies between assumptions. The first suggestion challenges assumptions that affect software design. This idea can be framed by our design argument model (Figure 2). The latter suggestion is about assumptions that can influence multiple decisions in a inter-related decisions (see Section 3.3).

**Properties of Design Constraints.** Berg et. al defines design constraint as a limiting condition that a design concern imposes upon the outcomes [5]. In searching for a solution in an infinitely large solution space, design constraints would help to reduce this space by limiting solutions to those that satisfy the constraints. Design constraints can come from system requirements as well as the system environment. Constraints may conflict with each other, and when this happens some requirements may need to be compromised. Some constraints influence the entire design, e.g. a project deadline. Finally, constraints can morph as a design is developed, meaning that a design constraint can lead to other constraints further down the chain of designs.

**Architecture analysis, synthesis and evaluation.** In a survey of students, Heesch & Avgeriou [14] have found that in analyzing requirements, not all students analyze or prioritize important requirements when designing. When synthesizing solutions, students were not aware of the constraints impose on other decisions. During design, students checked that all requirements were covered by at least one solution, but they did not consider interrelated decisions. When evaluating an architecture design, students did not do tradeoff analysis and they appeared to be biased towards an initial vision of a design.

### 4.2 Design Reasoning Techniques

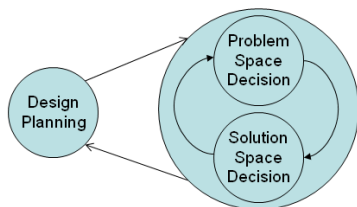
Let us for now assume that a designer is conscious of biases and tries to avoid them. This conscious reflection may motivate the practice of logical inferences and reasoning. This is one way to improve. Alternatively, perhaps some techniques and methods targeted at addressing specific biases may also help.

In a course at VU University Amsterdam, we have taught design reasoning techniques. Students are coached to use reasoning techniques in their projects. The notes we distribute to students [33] suggest a number of design reasoning techniques that may help them reason with their designs. Last year we noticed a substantial improvement in the general quality of project works. However, we are yet to obtain any proofs that the use of design reasoning techniques is a major contributing factor. Moreover, the design reasoning techniques are based on what we think the reasoning issues are, and on this basis we suggest ways to tackle those issues. Our understanding of the biases and their effects in software engineering is still insufficient. Without complete knowledge, we resort to a common software engineering approach – propose a method that appears to make sense and hope that it works. Below is a list of reasoning techniques. Not all of them are invented by us. Some of them have already been used in the software industry:

**1. Reasoning and Inferences.** First and foremost, a designer should be able to articulate and explain the premises, arguments and design conclusions. A designer should ensure that the basic premises used in the design arguments are valid and unbiased. Assumptions must be clearly spelled out. All relevant premises must be considered in the inference process. Tradeoffs, if applicable, must be evaluated in a decision. As design is a creative process, inductive arguments would be used regularly to identify design issues and solution options. Key design issues that arise from inductive arguments should be relevant and cogent, and they should be captured. Reasoning should help designers make sound and cogent decisions. However, to create a good design, one cannot treat it as a recipe and a step-by-step process. It is essential that designers remain creative, open-minded and careful.

Reasoning is a thought process. The results of such thought process can be demonstrated by documenting design rationale. One can imagine that capturing design rationale can be costly, but wrong decisions can be extremely costly also. Perhaps we should first try to make design decisions correctly by applying design reasoning. In the meantime, research and technology development shall continue to find ways to ease capturing design rationale.

**2. Problem Structuring.** We suggest that software designers should approach design in two interacting phases: design planning and problem-solution co-evolution (Figure 3). Firstly, designers should consider a high-level design plan. An overview of the requirements and key design issues should be identified. The software designer should identify major system goals, requirements and design issues. It has been found that early decisions on what design issues to tackle influence the way design activities are carried out [3, 29].



**Figure 3. Design Planning and Problem-Solution Co-evolution**

Secondly, designers should consider the design issues and then their potential solutions. This is an iterative process of exploring

the problem space and the solution space. Designers should also relate design issues to investigate how they influence each other. From a well-considered set of design issues, solution options can be devised. Again, there are no recipes on how this should be done, some designers take a breadth-first approach, some a depth-first approach. It may be that the nature of the design problems and the knowledge of the designers dictate what is more useful and effective in practice [2, 3].

**3. Assumption Analysis.** The validity and accuracy of a premise is based on whether there are assumptions behind the premise and if any hidden assumptions may inadvertently affect the design. It is therefore prudent to carry out an assumption analysis, questioning possible tacit assumptions that may have been made, consciously or unconsciously. A suggestion is for the stakeholders to explicate the assumptions of any key requirements and design. For instance, a modifiability requirement must state exactly in which parts of the software is required to be modified. This process may sometimes require on-going validation as there can be unforeseen parts of a design where assumptions need further elaboration.

**4. Constraint Analysis.** Requirements, system environments, project environments and organizations all exert some constraints on the way a system may be designed and implemented. These constraints are often tacit and not explicitly discussed or documented. As noted earlier, constraints can morph and they may also have a blanketing effect on the entire design [5]. It is therefore prudent of a designer to note the constraints of a requirement and a design. This would serve to detect conflicts in design. Additionally, at a decision point, software designers may assess constraining requirements for tradeoff analysis.

**5. Option Analysis.** In order to minimize the anchoring effect in which the first impression of a solution dictates the thought process, a software designer ought to grow a habit of exploring options in design. Additionally, a designer should also often consider the what-ifs scenarios, even relaxing some given constraints to evaluate if better design options are available. In an empirical student, we have found that designers who are prompted to state solutions options create a better design [34].

**6. Tradeoff Analysis.** A tradeoff exists when a design cannot satisfy all the requirements and constraints at a decision point. For instance, if the high cost of implementation conflicts with a high-performance requirement, then a tradeoff is required. Methodologies such as ATAM [18] suggests to evaluate the priorities and utility of quality requirements in making a tradeoff decision.

**7. Risk Analysis.** Risk can be treated as a kind of unknown with a probability that some adverse conditions can affect a design. Risks can be of technical nature, such as stability of a system platform, or of non-technical nature, such as the ability of a team to successfully adopt and use a new technology in the design [24, 32]. Unfortunately risks cannot be avoided during design time to assure certainty. Due to many project circumstances, decisions still have to be made based on some risk mitigation strategies. However, any risk, technical and non-technical, that affects a software design should be explicated and estimated. A checklist of some of the major risks in a software design can be prepared to remind software designers.

### 4.3 Learning to Reason

The study of design reasoning is about learning how to reason with design, especially under new situations. Generally speaking, the training we provide to computer science and software engineering students do not equip them properly to solve complex software design problem. For example, I need to train new graduates how to think design when they start working in my team. I find that, with training, the quality of design and their communication improves. It would be an idea to look into training students and practitioners on design reasoning as part of the software engineering training.

McCall [23] suggests that a feedback loop from design situation assessment and design discourse are critical to rethink design rationale. The ideation and evaluation in a feedback loop serves to improve creativity and reasoning. Similarly, Dngsøyr et. al [9] suggest a double-loop learning through technical analysis of important rationale to reflect on the experiences learned from software architecture design.

Schön [26] suggests to use reflection to improve practice. A practitioner may reflect on the tacit norms and appreciations that underlies a judgment, or on the strategies and theories implicit in a pattern of behavior. The practitioner may reflect on the feeling for a situation which has led him to adopt a particular course of action, on the way in which he has framed the problem he is trying to solve. Self reflection works at a meta-level, it is about the way we think as well as the problems we deal with. Unlike the other techniques that are directed at design, self reflection works on the mindset of designers. It serves to correct cognitive biases and reasoning failures because it invites designers to self-evaluate. This self-reflection may also help to imprint design reasoning as a habit so that design reasoning can gradually become an autonomous thinking process.

## 5. LOOKING TO THE FUTURE

In view of the existing design rationale methods for supporting software architecture design, the community has generally focused on modeling and representation. This is insufficient as software design problems are complex and the design environments change rapidly. A deeper understanding and learning is required for solving complex problems such as software design [1].

There are three important challenges in the advancement of software design reasoning. Firstly, we need to have a deep understanding of software design reasoning in order to improve our techniques. For examples, we want to understand what designers think and how their design decisions are made; how do they evaluate their premises and projection of the consequences. Secondly, we need to consider what the software engineering community can devise to help improve the design reasoning process. Thirdly, how do we train designers and students to reason with software design? Do we need an apprenticeship system or do we develop textbook techniques and methods? In order to tackle these three challenges, I suggest some studies that can be carried out:

- Study common industry judgment biases. What are the biases and why do they occur? Can they be avoided, and how? Where in the software engineering process are the biases most prominent?

- Study design reasoning techniques and how they may improve design quality. For instances, the precision and completeness of requirements, detecting and explicating hidden assumptions.
- Evaluate design reasoning in terms of software architecture, software components and software programs.

## 6. CONCLUSION

Studies in cognitive psychology tell us that people have a tendency to make biased and illogical decisions. Some contributors to these biases are (a) intuitive reasoning (*System I*) which demands a lower cognitive load; (b) limitation in short-term memory; (c) tendency to ignore probability representativeness. Although there is no hard evidence to demonstrate that systematic biases occur in software design, nor do we have data to show whether this issue is indeed serious in the software industry, there are enough examples to demonstrate that biases do exist and they may be more common than we think.

Systematic use of logical argument in design reasoning could help to improve general design quality. There are two basic conditions. Firstly, designers need to know the logical argument process to make sound and cogent design decisions. Secondly, premises such as requirements and design context must fulfill the quality criteria, and the communication of the premises faithfully between designers must be adequate and accurate.

Our understanding of this subject is limited. Doubtless the research community will continue to propose models and techniques to *fix* this problem. More importantly, we need to conduct empirical studies and experiments to understand how to make better software design decisions. This knowledge will allow us to improve design education of students and practitioners.

## 7. ACKNOWLEDGMENTS

This work has been the result of many collaborations and I acknowledge the contributions of my fellow researchers. In particular, I thank Patricia Lago, Hans van Vliet and Philippe Kruchten for their many inputs and comments. My research was in part sponsored by the Dutch "Regeling Kenniswerkers", project KWR09164, Stephenson: Architecture knowledge sharing practices in software product lines for print systems.

## 8. REFERENCES

- [1] J. R. Anderson, *The Architecture of Cognition*: Psychology Press, 1995.
- [2] A. Baker and A. van der Hoek, "Ideas, subjects, and cycles as lenses for understanding the software design process," *Design Studies*, vol. 31(6), pp. 590-613.
- [3] L. J. Ball, B. Onarheim, and B. T. Christensen, "Design requirements, epistemic uncertainty and solution development strategies in software design," *Design Studies*, vol. 31(6), pp. 567-589.
- [4] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Boston: Addison Wesley, 2003.
- [5] M. v. d. Berg, A. Tang, and R. Farenhorst, "A Constraint-Oriented Approach to Software Architecture Design," in *Proceedings of the Quality Software International Conference (QSIC 2009)*, 2009, pp. 396-405.

- [6] J. Bosch, "Software Architecture: The Next Step," in *Software Architecture: First European Workshop, EWSA 2004*, St Andrews, UK., 2004, pp. 194-199.
- [7] J. Burge, "Software Engineering Using design RATIONALE," in *Computer Science: Worcester Polytechnic Institute*, 2005, p. 211.
- [8] R. C. de Boer, R. Farenhorst, P. Lago, H. van Vliet, V. Clerc, and A. Jansen, "Architectural Knowledge: Getting to the Core," in *3rd International Conference on the Quality of Software Architectures (QoSA)*, 2007.
- [9] T. Dingsøyr, P. Lago, and H. V. Vliet, "Rationale promotes Learning about Architectural Knowledge," in *8th International Workshop on Learning Software Organizations (LSO 2006)*, 2006, pp. 59-67.
- [10] N. Epley and T. Gilovich, "The anchoring-and-adjustment heuristic," *Psychological Science*, vol. 17(4), p. 311, 2006.
- [11] J. Evans, "Heuristic and analytic processes in reasoning," *British Journal of Psychology*, vol. 75(pp. 451-468), 1984.
- [12] J. S. Evans, "In two minds: dual-process accounts of reasoning," *Trends in Cognitive Sciences*, vol. 7(10), pp. 454-459, 2003.
- [13] F. Shipman III and R. McCall, "Integrating different perspectives on design rationale: Supporting the emergence of design rationale from design communication," *Artificial Intelligence in Engineering Design, Analysis, and Manufacturing*, vol. 11(2), pp. 141-154, 1997.
- [14] U. v. Heesch and P. Avgeriou, " - A Descriptive Survey Naive Architecting - Understanding the Reasoning Process of Students," in *Software Architecture*. vol. 6285: Springer Berlin / Heidelberg, 2010, pp. 24-37.
- [15] P. J. Hurley, *A concise introduction to logic*. Belmont, Calif.: Thomson Wadsworth, 2006.
- [16] D. Kahneman and A. Tversky, "Subjective probability: A judgment of representativeness," *Cognitive Psychology*, vol. 3(3), pp. 430-454, 1972.
- [17] D. Kahneman, A. Tversky, and P. Slovic, *Judgment under uncertainty heuristics and biases*. New York: Cambridge University Press, 1982.
- [18] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," in *Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '98)*, 1998, pp. 68-78.
- [19] W. Kunz and H. Rittel, *Issues as Elements of Information Systems: Center for Planning and Development Research*, University of California at Berkeley, 1970.
- [20] P. Lago and H. van Vliet, "Explicit assumptions enrich architectural models," in *Proceedings 27th International Conference on Software Engineering (ICSE)*, 2005, pp. 206-214.
- [21] J. Lee and K. Lai, "What is Design Rationale?," in *Design Rationale - Concepts, Techniques, and Use*, T. Moran and J. Carroll, Eds. New Jersey: Lawrence Erlbaum, 1996, pp. 21-51.
- [22] A. Maclean, R. Young, V. Bellotti, and T. Moran, "Questions, Options and Criteria: Elements of Design Space Analysis," in *Design Rationale - Concepts, Techniques, and Use*, T. Moran and J. Carroll, Eds. New Jersey: Lawrence Erlbaum, 1996, pp. 53-105.
- [23] R. McCall, "Critical Conversations: Feedback as a Stimulus to a Creativity in Software Design," *Human Technology*, vol. 6(1), pp. 11-37, 2010.
- [24] E. R. Poort and H. v. Vliet, "Architecting as a Risk- and Cost Management Discipline," in *Proceedings of the Ninth IEEE/IFIP Working Conference on Software Architecture*, 2011.
- [25] H. W. J. Rittel and M. M. Webber, "Dilemmas in a general theory of planning," *Policy Sciences*, vol. 4(2), pp. 155-169, 1973.
- [26] D. A. Schön, "The reflective practitioner : how professionals think in action," Nueva York, EUA : Basic Books, 1983.
- [27] H. Simon and A. Newell, "Human Problem Solving: The State of The Theory in 1970," Carnegie-Mellon University, 1972.
- [28] H. A. Simon, "Bounded Rationality and Organizational Learning," *Organization Science*, vol. 2(1), pp. 125-134, 1991.
- [29] A. Tang, A. Aleti, J. Burge, and H. van Vliet, "What makes software design effective?," *Design Studies*, vol. 31(6), pp. 614-640, 2010.
- [30] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. Ali Babar, "A Comparative Study of Architecture Knowledge Management Tools," *Journal of Systems and Software*, vol. doi:10.1016/j.jss.2009.08.032(2009).
- [31] A. Tang, T. d. Boer, and H. v. Vliet, "Building Roadmaps: A Knowledge Sharing Perspective," in *Sixth Workshop SHARING and Reusing architectural Knowledge*, Hawaii, 2011.
- [32] A. Tang and J. Han, "Architecture Rationalization: a Methodology for Architecture Verifiability, Traceability and Completeness," in *12th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems ECBS 2005*, U.S.A., 2005, pp. 135-144.
- [33] A. Tang and P. Lago, "Notes on Design Reasoning Techniques (V1.4)," Swinburne University of Technology Melbourne, 2010.
- [34] A. Tang, M. H. Tran, J. Han, and H. van Vliet, "Design Reasoning Improves Software Design Quality," in *Proceedings of the Quality of Software-Architectures (QoSA 2008)*, 2008.
- [35] J. Tyree and A. Akerman, "Architecture Decisions: Demystifying Architecture," *IEEE Software*, vol. 22(2), pp. 19-27, 2005.
- [36] C. Zannier, M. Chiasson, and F. Maurer, "A model of design decision making based on empirical results of interviews with software designers," *Information and Software Technology*, vol. 49(6), pp. 637-653, 2007.
- [37] O. Zimmermann, J. Koehler, F. Leymann, R. Polley, and N. Schuster, "Managing architectural decision models with dependency relations, integrity constraints, and production rules," *Journal of Systems and Software*, vol. 82(8), pp. 1249-1267, 2009.