Software Designers, Are You Biased?[1]

Antony Tang[2]

Studies have shown that people make biased decisions, software designers may also be subjected to cognitive biases. In this article, I give an overview of how cognitive biases and reasoning failures may lead to unsound design decisions. I suggest that in order to improve the overall quality of software design, we need to improve our understanding of software design reasoning through application of reasoning techniques.

1. INTRODUCTION

Software design was said to be a wicked problem by Rittel and Webber back in the 70's [1]. 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.

Software engineering processes and techniques can help designers to design, 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 software design reasoning failures:

- Cognitive Bias – a distortion of judgment in particular situations due to psychological effects and insufficient regards of probability [2]. 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 [3]. 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.

2. SOFTWARE DESIGN REASONING FAILURES

Many software development projects make use of software development processes, methodologies and quality reviews. However, not all of these projects are successful. Therefore software engineering processes and methodologies do not guarantee success. Their use is a necessary but insufficient condition to developing quality software.

---

[1] This article is an excerpt from a Keynote speech given in the *Sixth Workshop on SHAring and Reusing architectural Knowledge* (*SHARK 2011*)

[2] Antony Tang is Associate Professor, Swinburne University of Technology, email:atang@swin.edu.au

Besides the appropriate use of processes and methodologies, the right *soft-skills* of a designer is a key success factor. S*oft-skills* are about how a 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 are about design reasoning and avoiding biases and they can help a software designer to succeed.

2.1 Cognitive Biases

Cognitive bias was first introduced by Kahneman and Tversky [4]. The idea is that human judgments can be distorted by biases 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. Human tends to judge without regards to the likelihood of events. Therefore biased thinking generates systematic judgments errors.

This phenomenon also happens 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 accuracy of the estimations of efforts by a programmer would improve. From anecdotal evidence, I find that it generally takes a while before this cognitive bias is adjusted. Some other cognitive biases 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 superficial arguments that can lead to under evaluated decisions. 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 [5].

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 [6, 7] 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 [8]. 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 [9]. 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.

2.3 Low 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. 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 are premises missing in this argument. For instance, how many users would use this service simultaneously to warrant an in-memory design? Without specifying such premises, 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 about the way operators operate to a designer. A designer is aware of the number of transactions and the reason for having efficiency. If the transaction rate is not available, a designer can design a simple data entry interface but this user interface does not have to meet the expected efficiency requirement.
- Fairness – the relative priority of all the premises in an argument is important. An unfair evaluation of their relative priority can distort a 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.

The criteria for evaluating the quality of premises are crucial 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 room for misinterpretation. In a real-life instance, a designer required the software to be modifiable but he 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 meet the expectations. The system failed because of mismatched assumptions.

3. LOGICAL DESIGN ARGUMENT

The aim of logic is to develop a system of methods and principles that we may use as criteria to evaluate arguments [3]. 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. So a designer could use different kinds of implementation such as thick client or server-to-server connections.

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.

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. 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, 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.

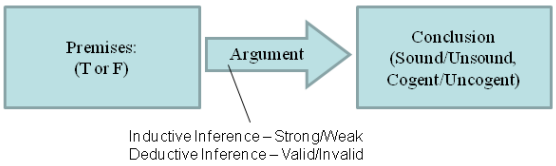

Figure 1. General Form of an Argument

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

The argument that "Oracle runs on UNIX. SAP uses Oracle. Therefore this SAP application should run on UNIX" is weak. Although it is true that 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.

4. HOW TO IMPROVE DESIGN REASONING

Let us for now assume that a designer is conscious of biases and tries to avoid them and this conscious reflection motivates 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 [10] suggest a number of design reasoning techniques that may help them reason with their designs. We noticed a substantial improvement in the general quality of project works. However, we are in the process of obtaining scientific proofs that the use of design reasoning techniques is a major contributing factor.

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 should 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.

2. Problem Structuring. We suggest that software designers should approach design in two interacting phases: design planning and problem-solution co-evolution. 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 [11, 12]. 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.

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 a 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. 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 study, we have found that designers who are prompted to state solutions options create a better design [13].

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 [14] 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. Unfortunately risks cannot be avoided during design time and uncertainties persist. Due to many project circumstances, decisions still have to be made shadowed by some risks. However, any risk, technical and non-technical, that affects a software design should be explicated and analyzed. A checklist of some of the major risks in a software design should be prepared to remind software designers.

8. Learning to Reason. The study of design reasoning is to learn 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. Schön [15] 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 a situation which has led him to adopt a particular course of action, or reflect on the way in which he has framed the problem. Self-reflection works at a meta-level, it concerns with the way we think and the problems that 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. REFERENCES

[1] H. W. J. Rittel and M. M. Webber, "Dilemmas in a general theory of planning," *Policy Sciences,* vol. 4, pp. 155-169, 1973.

[2] D. Kahneman, A. Tversky, and P. Slovic, *Judgment under uncertainty heuristics and biases*. New York: Cambridge University Press, 1982.

[3] P. J. Hurley, *A concise introduction to logic*. Belmont, Calif.: Thomson Wadsworth, 2006.

[4] D. Kahneman and A. Tversky, "Subjective probability: A judgment of representativeness," *Cognitive Psychology,* vol. 3, pp. 430-454, 1972.

[5] N. Epley and T. Gilovich, "The anchoring-and-adjustment heuristic," *Psychological Science,* vol. 17, p. 311, 2006.

[6] J. S. Evans, "In two minds: dual-process accounts of reasoning," *Trends in Cognitive Sciences,* vol. 7, pp. 454-459, 2003.

[7] J. Evans, "Heuristic and analytic processes in reasoning," *British Journal of Psychology,* vol. 75, pp. 451-468, 1984.

[8] H. A. Simon, "Bounded Rationality and Organizational Learning," *Organization Science,* vol. 2, pp. 125-134, 1991.

[9] H. Simon and A. Newell, "Human Problem Solving: The State of The Theory in 1970," Carnegie-Mellon University1972.

[10] A. Tang and P. Lago, "Notes on Design Reasoning Techniques (V1.4)," Swinburne University of Technology 2010.

[11] A. Tang, A. Aleti, J. Burge, and H. van Vliet, "What makes software design effective?," *Design Studies,* vol. 31, pp. 614-640, 2010.

[12] L. J. Ball, B. Onarheim, and B. T. Christensen, "Design requirements, epistemic uncertainty and solution development strategies in software design," *Design Studies,* vol. 31, pp. 567-589, 2010.

[13] 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, pp. 28-42.

[14] 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.

[15] D. A. Schön, "The reflective practitioner : how professionals think in action," ed: Nueva York, EUA : Basic Books, 1983.