Human Reasoning and Software Design: An Analysis

Antony Tang, Aldeida Aleti
Swinburne University of Technology
{atang,aaleti}@swin.edu.au

Abstract

Software design is a complex cognitive process in which reasoning plays a major role, but we have limited understanding of how human reasoning works in the identification of design problems and the formulation of design solutions. In this research, we have observed software designers at work and analyzed their design reasoning approaches, the effectiveness of design, in part, depends on how design problems are structured and how design reasoning is applied. We report on how reasoning techniques, such as design structuring, contextualization, co-evolution of problem-solution and inductive reasoning influence software design.

1. Introduction

Software design is a highly complex and demanding activity. A software designer often deals with changing requirements and technical environments. One often faces new problem domains where the knowledge about a design cannot be found readily. The characteristics and behaviors of the software and the hardware systems to be considered in the design are often unknown and the complexity in user and quality requirements is high. Under such complex environment, a software designer needs sound reasoning capabilities to make good design decisions and to devise a good design solution.

Software practitioners and software engineering researchers have invented many processes, modeling techniques and first principles to guide software designers to create software products. There is, however, very little study and guidance on systematic software design reasoning and decision making. In reality, most of the software designers approach design based on personal preferences and habits, with various results in productivity and quality of the end-designs.

Researchers in psychology have proposed that there are two distinct cognitive systems underlying reasoning: the heuristic system relies on prior knowledge and beliefs; the analytic system states that reasoning is according to logical standards [1]. 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, causing designer a rational thinking failure [2]. The comprehension of an issue also dictates how people make decisions and rational choices [3]. The comprehension of design issues also depends on how designers frame or structure the design problems. Different ways to frame design problems may result in different design results. In this paper, we investigate how reasoning influences software design.

The University of California, Irvine, prepared an experiment in which three pairs of software designers were given a set of requirements to design a traffic simulator. Their design activities were video recorded and transcript. Using those transcripts, we have analyzed their design reasoning activities in two levels: (a) structuring the design; (b) reasoning in terms of problem identification and solution formulation.

From a design problem structuring perspective, an important question is how software designers organize design activities to achieve user and quality requirements. In this study, we have encoded protocols and created decision maps from the transcripts. From the qualitative analysis of these materials, we have found that proper structuring of design discussions improves design issue identification and reduces context switching. We also analyze reasoning techniques that have been used by the designers. We have found that systematic problem-solution co-evolution, appropriate contextualization of design problems, explicit communication of design reasoning and application of inductive reasoning are important techniques to improve the effectiveness of software design. We have found that the level of inductive reasoning is related to the exploration of design problems, implying that it plays an important role in defining the design problem space.

2. Related work

Software design expertise is in part domain dependent, which is different to some design areas where the context of the domain is relatively constant. For instance, the issues faced by scientific system designers are quite different to that of transactional financial system designers. Therefore, an expert software designer may act differently when faced with unfamiliar domains and technologies. Cross suggests [4] that expert designers appear not to generate a wide range of alternatives. We have found in our earlier study that 'experienced' (depending on domain exposures) software designers intuitively rule out unviable design alternatives whereas 'inexperienced' software designers can benefit from explicitly considering design options [5].

Another characteristic of software design is the complexity of the design space. When software designers are unfamiliar with a domain, the detailed design problems can be difficult to define and the viability of a solution cannot be assessed easily. This is because the behavior of software and systems cannot be predicted easily, and design issues are complex, interrelated and conflicting.

Although there are many studies on software engineering design, few of them study the cognitive aspects of the software designers. Recent studies address the use of design rationale [6-8] in software engineering, mostly from the perspective of documenting design decisions instead of a systematic approach to applying software design reasoning.

Typically, software design methodologies employ a blackbox approach that emphasizes the development process and its resulting artifact with little exploration of the cognitive and psychological aspects of designers who use them. For instance, Hall et al. suggest that problem frames provide a means of analyzing and decomposing problems, enabling the designer to design by iteratively moving between the problem structure and the solution structure [9]. This method has a resounding similarity with the Mahler model of co-evolution between the problem and solution spaces, which is described by Dorst and Cross in [10]. They suggest that creative design is developing and refining together both the formulation of a problem space and ideas in the solution space (see Figure 1). A study by Zannier et al. has found that designers make use of both rational and naturalistic decision making tactics in decision making [11], designers make more use of a rational approach when the design problem is well structured, and inversely designers use a naturalistic approach when the problem is not well structured.

Another similar representation of problem-solution co-evolution model is shown in Figure 2. The software architecture design rationale model of AREL represents the causal relationships between DesignConcerns such as the requirements, DesignDecision node representing the design problems and the decisions, and DesignOutcome node representing the design outcomes. The design outcomes become a design context or concern when a design is decomposed and interrelated design issues are linked to new design problems [12].

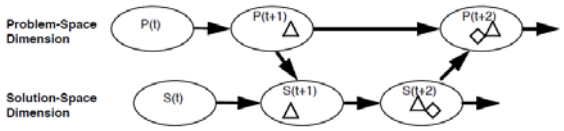

Figure 1. Problem-solution co-evolution

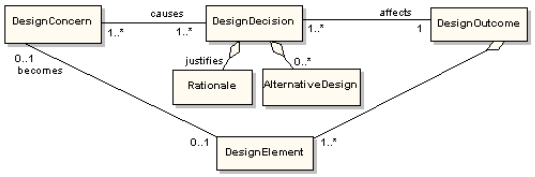

Figure 2. Causal relationships between design concern, decision and outcome

Simon and Newell suggest six sources of information that can be used to construct a problem space [13], providing a context or a task environment for the problem space. So defining the context is an important aspect to formulating and identifying a problem. Similarly, we suggest that decisions are driven by decision inputs, or context to a design problem, and the results of a decision are some decision outcomes. AREL diagrams enable software designers to document and trace design decisions and design rationale from requirements to design artifacts.

Given some software requirements, designers must look at how to structure the design problems and plan an approach to designing. Goel and Pirolli reported that on average 24% of statements made by the architects are devoted to problem structuring and this activity occur mostly at the beginning of the design task [14]. Therefore, design structuring or planning is an important aspect of design.

Rittel and Webber [15] viewed deign as a process of negotiation and deliberation. They suggested that design is a "wicked problem" in which it does not have a well-defined set of potential solutions. Even though the act of design is a logical process, it is subject to how a designer handles this wicked problem. Goldschmidt and Weil describe design reasoning as the relationship between contents and structure [16]. They suggest that the process of reasoning as represented by design moves are double speared where

one direction is to move forward with new design, another direction is to look backwards for consistency.

Inductive reasoning, especially analogical reasoning, is a means of applying knowledge acquired in one context to new situations [17]. It promotes a creative process in which issues from how different pieces of information can be combined to form a new artifact [18]. There are different techniques of inductive reasoning [19] that software designers can use. One of them is the use of scenarios to analyze architecture design [20], and that agrees with the Klauer's model of inductive thinking paradigm [19]. If design is a result of some forms of reasoning performed by a designer, then it is important to study the reasoning techniques and process relating to software design.

3. Design study and analysis methods

3.1. The design assignment

The design task is to build a traffic simulation program that is going to be used by students to understand the relationships between traffic lights, traffic conditions and traffic flow. The topic of the study is common enough so that the designers would have no issues with its context, but specialized enough that general software designers are unlikely to have designed similar systems previously.

Req-ments	U/I	Traffic Light	Sim. Display	Traffic Density	NFR	Design Outcome
Explicit	4	6	4	2	4	7
Derived	2	5	4	4	0	0

Table 1. Number of requirements of the system

The designers were given a Design Prompt or a brief specification on the problem where there are twenty explicit functional and non-functional requirements, and fifteen derived requirements (see Table 1). Derived requirements[1] are requirements that are not specified in the Design Prompt but need to be addressed to complete the design. They are derived from analysis by the three teams of designers including the authors' interpretation of the requirements. The Design Prompt explicitly specifies seven design outcomes.

[1] Requirement coverage can be found in supporting document http://www.ict.swin.edu.au/personal/atang/ProfessionalSoftwareDesignStudy/Requirements and derived requirements.pdf

3.2. Purpose of the study

We approach the analysis from the perspective of design reasoning. Software engineering research has been focusing on processes and methodologies without studying why and how these methods would work. In order to comprehend why a software engineering approach should work, we need to understand human cognition and design reasoning. Recently, software engineering researchers have started to study design rationale and decision making in this regard. This study examines design reasoning at two levels: firstly, structuring of design problems; secondly, reasoning and problem solving of design issues.

3.3. The analysis method

This is a case study of three teams of designers: Adobe (A), Anonymous (N) and Amberpoint (M), we'll call them Team A, Team N and Team M respectively. Team A and M took two hours and Team N took just under one hour to complete the design tasks. Each team conducted the design activities in their own ways. We use a protocol coding scheme to analyze their dialogue with a process similar to [21] and [22]. After protocol encoding, we build a decision map using a graphical tool to relate design reasoning topics.

3.3.1. Protocol analysis and coding schemas. To analyze the design process, we have a coding scheme that identifies reasoning activities. It helps dissect the activities of design framing and the design reasoning techniques used in solving individual design problems.

Time	Dialogue	Space	Reasoning Technique	Ex/Im-plicit Reason	Decision	Time Link
[0:12:01.1]	I don't think we need to make …the no of signals at each ….	SS	S+I	E	C	[0:11:45.3]

Table 2. An example of the coding scheme

Software designers typically start by summarizing the requirements, contextualizing the problems and prioritizing issues. This kind of problem structuring is breadth-first reasoning at a high-level of abstraction [23]. We then identify reasoning techniques for protocol coding. Techniques relating to the problem space are problem identification (G), contextualization (X), prioritization (R) and simplification (S). Reasoning techniques relating to the solution space are scenario construction (E), inductive reasoning (I), deductive reasoning (D), option identification (O), constraint identification (C), and trade-off analysis (T).

Table 2 is an example of our protocol coding and the reasoning activities are coded in the 4th column.

We codify the transcripts into problem space (PS) and solution space (SS) (see 3rd column in Table 2). This enables us to study the co-evolution of problem and solution. Reasoning by designers can be made implicitly or explicitly, this is shown in the 5th column. The decision is coded according to status (6th column): (P)roposed, (C)onfirmed, (R)ejected and (I)terated. This coding enables us to analyze when and how designers make their decisions. Column 1 shows the protocol sequence in time, column 7 is a time linking the current dialogue to the related dialogue. It enables us to connect the line of reasoning. Column 2 documents the dialogue.

3.3.2. Decision map. After encoding the protocol, we build a decision map to represent the design activities. The decision map is based on the AREL model where design decision making are three nodes that consist of [*design concern, design decision, design outcome*] and the causal relationships between them. A *Design concern* node represents information that is relevant to the design issue, such as requirements, context or design outcome. Decision node represents the design problem. Design outcome node represents the solution of a decision. A decision map shows the progression of a design by connecting the problem space (decision node in rectangular boxes) to the solution space (design outcome in ovals). The contexts of the design are represented by rectangular boxes with double vertical lines. The nodes in the decision map are linked together by (a) the sequence of discussions, i.e. time; (b) the subject of the discussion. Figure 3 shows Team M's design planning in the first five minutes of their discussions. The decision map depicts the following: (a) structuring of a design approach; (b) sequencing of design reasoning; (c) design problem framing and identification; (d) design justification. [2]

4. Data analysis

Using the coded protocols and the decision maps, we analyze how the three teams reason with their designs. We approach the analysis from two primary perspectives: (a) overall structuring and sequencing of the design problems; (b) application of design reasoning techniques in specific design areas.

[2] The decision maps are constructed with a UML tool Enterprise Architect version 7.00. Decision maps are available at http://www.ict.swin.edu.au/personal/atang/ProfessionalSoftwareDesignStudy/decision-analysis-v1.6.eap.

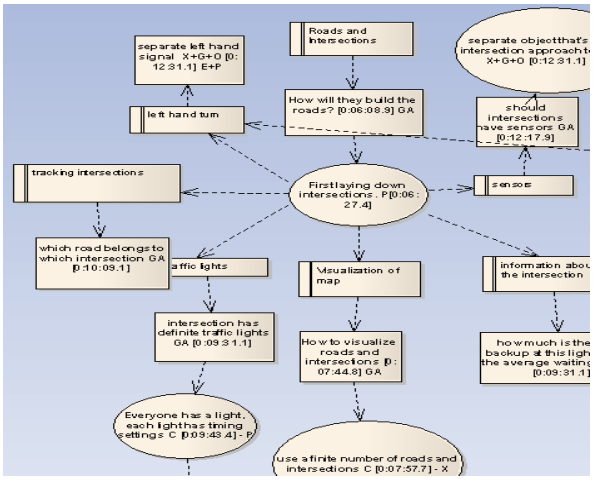

Figure 3. An extract of a decision map

4.1. Design structuring and sequencing

Each team performed some planning of their design activities. We use decision maps to do the analysis. The decision maps depict the design problems and the resulting solutions in a sequence of decisions that are related to the same topic. Although iterations and branching are common, the design problems and solutions are typically depicted in a sequential order. For instance, Team N focused on the topic of *roads* between [0:18:36.2] and [0:19:22.2] and they discussed its properties and design. From the visual maps, the design discussions can be identified easily. Based on the groupings, we analyze how their design activities are structured.

4.1.1. Planning design approach. All three teams started off by planning their design approach, which involves assessing the requirements, identifying and structuring high-level design problems and making some implicit prioritization of problems. Team A spent about 1.5 minute ([0:05:11.0] till [0:06:41.1]) on planning and they identified representation and modeling of traffic flow, signal, intersection as their key design issues. Team N spent about 9 minutes ([0:05:29.7] till [0:14:20.0]) on structuring their problems, i.e. they did an assessment of where the design complexity lie and concluded that the dependency was the signals and turning cars. They also pointed out that the user interface and the data structures of the system needed to be identified. Team M spent about 6.5 minutes ([0:06:08.9] till [[0:12:31.1]) on planning and they identified ten design issues. The three teams spent 1%, 19% and 6% of their total time, respectively, on problem structuring. In comparison, Goel and Poirolli found that on average 25% of statements was devoted to

problem structuring mainly at the beginning of design [14].

Design planning has influenced how each team structures the rest of their design. The initial focus and prioritizations on specific areas dictate the structure of the rest of design discussions. Implicit design planning took place even though none of the teams explicitly mentioned it. For instance, Team A's focal points were data structure, modeling and representation. They spent the next 40 minutes ([0:08:28.4] till [0:48:02.1]) on queue management before moving to the other 11 discussion topics. Team N identified signaling as important and they immediately tackled this design problem after planning [0:15:38.5].

Design re-planning were carried out by Team A and N during the design. At [1:17:59.15], Team A assessed their design status and realigned their focus on the design outcomes, and the rest of the design discussions followed this direction. Team N re-planned at [0:43:52.9] and they identified remaining design gaps that needed to be resolved and proceeded to solve them.

4.1.2. Structuring design activities. Using the discussion grouping by design topics, we observe that the way each team explored the design space followed different reasoning structures. Team A used a *semi-structured approach*. They started with a design discussion that focused on the premise of a queue and the push/pop operations, and they explored this solution concept extensively to design the traffic simulation system. Further, they continued the discussions about roads and cars ([0:42:40.4] till [0:52:15.7]) but without reaching either a solution or a clear identification of the next design problem, they moved to the topic on sensors ([0:54:33.6]). Such inconclusive design discussions continued until [1:17:59.15] when they stopped and planned their design approach. At that point the designers reassessed their design problems that guided them till the end of the design session.

Team N used a *compositional structuring approach*. They firstly spent a short time to investigate each of the major components of the simulation, such as: (a) synchronization of signals ([0:15:38.5] till [0:17:04.7]); (b) traffic; (c) roads; (d) lanes; (e) left-turn lane; (f) model; (g) intersection and map. After gaining a fair understanding of all the key components, they then composed the complex design from the basic components: (a) traffic ([0:34:25.9] till [0:39:48.0]); (b) simulation ([0:40:33.5] till [0:43:52.9]); (c) cars interacting with roads and intersections ([0:43:52.9] till [0:49:39.9]) and so on. This reasoning structure enables the designers to investigate simple objects before moving to design involving multiple design objects.

Team M used a *high-low structuring approach*. Team M tackled the major design problems in-turn and when they concluded the design of the major problems, they addressed associated issues such as user interface. For instance, they started their design discussions by addressing the issue of traffic density [0:13:35.5]. At the end of the high-level design, they investigated the relevant but low-level issue regarding simulation interface [0:26:43.5]. Then they moved to another high-level critical problem on traffic light, and at the end of the discussions they again considered the low-level interface issue [0:41:13.3]. In doing so, the designers started with a high-level design problem, solved it and then tackled other small but related design problems.

4.1.3. Reasoning structure and context switching. Based on the three types of reasoning structure, we have observed that designers' focus shift according to the planning of the design and how the design issues are explored. Once design issues are explored, context switching, i.e. focusing and defocusing of a design issue, takes place between addressing related issues. Context switching is most noticeable in the semi-structured approach. For instance, during the queue management discussion by Team A, the designers focused and defocused on traffic lights several times at [0:20:36.2], [0:39:17.8] and [0:49:41.0]. Then this topic was raised again at [0:55:28.8] and [1:49:44.0] but Team A did not have a clear and complete design for all issues related to traffic light. It appears that the context switching affects the resolution of design issues. The more context switching that takes place, the less the designers follow a single line of thought to explore a design issue thoroughly. Therefore, even when the same issue was discussed many times, a clear solution was not necessarily devised.

In contrast, Team M addressed traffic light [0:28:16.4] in a single discussion. There is no context switching to other design issues apart from brief visits to closely related issues, e.g. sensors [0:28:54.1]. The designers acknowledged the other design issue but quickly returned to the main topic of the design. As such, Team M explored related issues more thoroughly. Based on the observations, we argue that high context switching could lead to defocusing of a single issue, and designers may lose the context of the design, making it harder for designers to formulate the right design problems.

4.2. Design reasoning techniques

Design is a *rational problem solving* activity [10]. It is a process in which designers explore the design issues and search for rational solutions. It is well established by many that design rationale is one of the most important aspects in software design [6, 24]. However, the basic techniques that lead to rational design decisions are not well understood. Therefore, we explore different design reasoning techniques that are used by the three teams.

4.2.1. Problem-solution co-evolution. Dorst and Cross have observed co-evolution of design between problem and solution spaces [10]. Tang et al. have reported on software design decision making in terms of expanding relationships between decision context, design issues and solutions [25]. The way the designers in the three teams conducted their designs was largely aligned with the co-evolution model. For instance, Team M demonstrated a co-evolution process where a problem was identified and solved, leading to the next problem-solution cycle (see Fig. 4). However, this model presumes a logical design reasoning behavior in which a solution is derived from well-defined design problems. However, we have observed design behaviors that differ from this model.

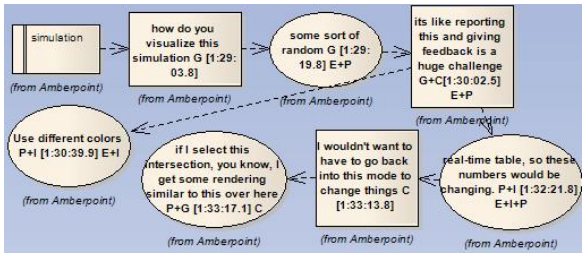

Figure 4. Example of a problem-solution co-evolution in Team M design session

Solution driven design. At a very early stage, Team A identified traffic flow and signal as design issues [0:06:10.6] and they decided that a queue should be used in the solution [0:08:28.4]. From this point onwards, Team A continued to make a series of decisions surrounding the operations of the queue. Some examples are [0:08:51.0] (shown in Fig. 5), [0:38:05.6] and [0:39:17.8]. The idea of the queue drove the discussions from [0:08:28.4] till [0:50:43.1]. The design plan and all subsequent reasoning are predominantly based on this solution concept that was decided up-front, without questioning whether this is the right approach or if there are alternative approaches. It is also evident that there is a lack of exploration or implicit assumption about the problem space. This case demonstrates that when designers take a solution driven approach, the preconceived solution becomes the dominant driver of the design process, and the rest of the solutions evolves from the preconceived solution. Such design behavior has been observed previously. Rowe [26] observed that a dominant influence was exerted by the initial design even when severe problems were encountered. Tang et al. found that for some software designers, the first solution that came to mind dominated their entire design thinking, especially when the designers did not explicitly reason with their design decisions [5].

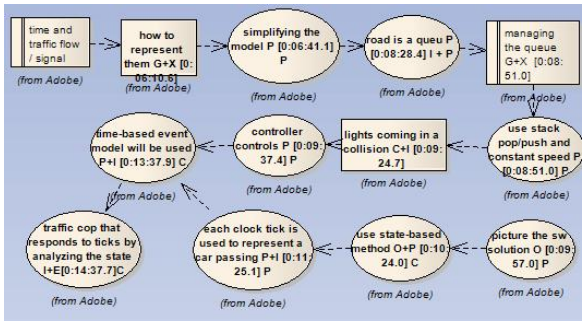

Figure 5. Example of a solution driven design by Team A

Absent or ambiguous design solution. There are cases where designers identified design issues in the problem space without conclusions. When Team M was examining traffic light control for a left-turn signal [0:40:07.0], the discussion led to identification of other issues such as how to deal with the user interface and timing of changing colors of adjacent traffic light. Both unsolved issues were relevant in the simulation but before concluding with a design the discussion drifted to the next topic [0:41:23.8].

From the analysis, there was no definitive evidence to indicate why the discussion drifted before a solution was reached. We assume that the designers might either feel that they had to move on to higher priority design; or the problems are too-hard to contemplate at the time; or simply distraction. Similarly, examples can be found in Team A's design session.

4.2.2. Contextualizing problems. Contextualization is the process of attaching a meaning and interpreting a phenomenon to a design problem. Well reasoned arguments state the context of the problem, i.e. what and how the context influences the design. By identifying the context at the right level of abstraction, it helps to frame the design issues [13], identify the assumptions of the situation and the constraints on the

potential solution. So how designers contextualize a design problem would influence the design outcome.

All teams contextualize their problems in different ways, we counted the number of nodes in the decision map where they discuss the context of the problem and state the details that can influence the potential solution. We identified 26 nodes for Team A and 23 nodes for Team M. Although the number of nodes is similar between the two teams, Team M is more thorough in articulating the context. An example is when Team M examines the traffic lights [0:28:16.4]. The designers spell out 3 relevant contexts of the problem, i.e. the timing of the cycle, the impact of the sensors and the left-turn arrow. Their discussion of the design problem using those contexts happened at the same time frame, as such it provides a focal point for investigating a design problem comprehensively.

On the other hand Team A contextualizes their design problems in a sporadic and uncoordinated way. For example, when Team A discussed traffic signals, they did not consider all relevant contexts at the same time. The designers discussed signal rules at [0:12:31.1], they revisited the subject of traffic signals with respect to lanes at [0:25:37.4]. Later they decided that it is the intersection which is relevant to traffic signals [0:30:41.9]. At [0:56:06.7] they discussed how signals could be specified by the students. Twenty-one minutes later [1:17:58.4] their discussion went to the subject of sensor and left signal. Since contextualization impacts on design problem framing, the difference between how the two teams use contextualization affects the information that is available for the reasoning of a design problem.

4.2.3. Scenario construction. Using scenarios enhances the comprehension of the problem and facilitates communication between designers. Constructing scenario is a way to instantiate an abstract design problem and evaluate if the abstract design is fit for the real-world problem [20]. For example Team A built a scenario of how the simulation could work, talking about the possibility of having a slider to control the timing of lights and car speed [0:48:20.4]. Using this scenario, the designers highlighted a number of related design problems, e.g. the average wait time of a car going through the city streets; car speeds; traffic light control etc.

Scenario construction technique was used in 7 cases by Team M and 22 cases by Team A. This shows that the approaches used by the two teams are very different. Team A considered the design problems by expressing them in real world examples, whereas Team M remains at an abstract design level. Even though Team A used a lot of scenarios in their design reasoning, they often did not arrive at a design conclusion to cater for the scenario, e.g. [1:26:00.0]. Team M, on the other hand, worked at an abstract level of the design, and using scenarios to test the abstract idea, e.g. [0:32:27.5]. Thus it appears that the effectiveness of applying scenarios depends on the construction of an abstract design.

4.2.4. Implicit or explicit reasoning. Explicit reasons are explicated arguments to support or reject a design problem and a solution choice, but implicit design reasons are decisions made without communicating an explicit reason. When explicit reasons are absent, people other than the designers themselves make assumptions about why the decision takes place. Without explicit reasoning, it is difficult for the designer, the design team or reviewers to identify if there are any omissions. In this study, explicit reasons can be identified with the word "because" in the protocol and the arguments to support the explanations. An example of explicit reasoning is where the designers of Team M clearly state the reason for their solution [0:25:35.1].

However reasoning is often not verbalized for a number of reasons: (a) the designers may have reasons that they do not express explicitly; (b) the reasoning is the assumption made in the discussion of the subject or the context; (c) they have omitted the reasoning. An example of implicit reasoning is when Team A discussed about push/pop [0:41:36.0] putting all traffic complexity in these two actions, without explicitly saying why they chose to do so, and how they will implement it. Moreover, it is implicitly decided that the actual direction of cars while pushed and popped will be handled by a random number generator. When Team A decided that observations should be taken from the simulation [1:31:25.2], they assumed without any reasoning that metrics such as average wait time of cars, average capacity of the roads, etc. are available to support their decisions. When we analyzed the protocols of Team M and Team A, Team M made 33 cases of implicit reasoning and 87 cases of explicit reasoning, Team A made 47 cases of implicit reasoning and 52 cases of explicit reasoning. Team M provided more explicit explanation of how they designed. Team A provided less explanation of their design ideas, the implicit reasons required interpretation to appreciate why they designed in a certain way, therefore there are opportunities for the others to misinterpret the design ideas.

4.2.5. Inductive and Deductive Reasoning. Inductive reasoning is a type of reasoning that generalizes specific facts or observations to create a theory to be applied to another situation whereas deductive

reasoning uses commonly known situations and facts to derive a logical conclusion. While inductive reasoning is exploratory and helps develop new theories, deductive reasoning is used to find a conclusion from the known facts.

Let us consider first how both teams have used inductive reasoning. We have counted 15 induction cases for Team A and 30 induction cases for Team M. Team M has used twice as many inductive reasoning as Team A. It indicates that Team M employs a more investigative strategy, inspecting the relevant facts to explore new design problems and solutions. As an example, Team M discussed the relationships between the settings of traffic lights to prevent collision [0:34:30.6]. From the observations that cars can run a yellow light and to prevent collision, they induce that there should be a slight overlap when all traffic lights facing different directions should be red simultaneously. Then they reason inductively about the manual settings for the timing of the lights and whether the users should be allowed to set all four lights manually. Since the users should not be allowed to set the traffic lights to cause collisions, the designers reason that there should be a rule in which the setting of a set of lights would imply the behavior of the other traffic lights. Team M in this case found new design problems from some initial observations.

On the other hand, deductive reasoning is used more times by Team A than Team M, with 20 and 10 times respectively. For instance, Team A started off by predicating on the basic components used in the solution, i.e. *queue* and *cop* etc., and deduced the solution from that. They used deductive reasoning at [0:30:19.3]. They said that a lane is a queue, so multiple lanes would be represented by multiple queues. Since they also predicated on the use of a *traffic cop*, so they deduced that the *cop* would look after the popping and pushing of multiple queues. Team A was fixed in the basic design of the system from the beginning, so although the logical deduction is correct, the design options that have been explored are limited.

5. Discussion on Findings

Although design experience in software is highly important, it is also domain specific. In this experiment, designers are highly experienced and they possess the general knowledge about the problem domain, but they do not appear to have the detailed design knowledge in this area. So designers have relied on their general design knowledge and reasoning to create their solutions. All designers were given two hours to design, which was insufficient time to exhaustively explore all possible solution branches to provide a complete solution. Therefore, their design is measured by how effective the designers have been analyzing the problems and formulating the solutions.

We cannot evaluate the quality of each team's design because: (a) the criteria we use in design quality reviews can have a bias on the analysis done; (b) the created solutions are incomplete; (c) designers in some cases have not sufficiently explained a design decision. In order to understand how reasoning influences their design outcomes, we investigate the requirement coverage. Design coverage is defined as the number of definitive solutions that are provided to address the requirements, derived requirements and design outcomes. Table 3 indicates the coverage that each team provided for each category of the requirements and derived requirements. The numbers [4,2] in the user interface (U/I) cell, title row, indicates the total number of requirements and derived requirements in the user interface category from Table 1. The number in the cells for each team indicates if a concrete design solution has been provided for the requirement. For instance, Team A covered 3 out of the 4 U/I requirements and 1 out of the 2 derived U/I requirements. In total, Team M has the highest requirement and derive requirement coverage (40 out of 42), followed by Team A (28 out of 42) and then Team N (23 out of 42).

Team	U/I [4,2]	T/L [6,5]	S/D [4,4]	T/D [2,4]	NFR [4]	D/O [7]	Total [42]
A	3,1	4,3	1,4	2,4	1	5	28
N	3,2	4,4	0,3	0,4	0	3	23
M	4,2	6,4	3,4	2,4	4	7	40

Table 3. Solution coverage by Team A, N and M

Team N completed their design within an hour, and although they did a lot of analysis, they did not articulate their design solution in concrete terms, and thus their requirement coverage is the lowest. Since Team N spent only half the designated time on their design, and to avoid this time bias we based our analysis mainly on Team A and M.

Design structuring performed by the three teams resulted in varying level of details. Team A took the least time and provided the briefest plan, Team N and Team M took longer time to structure their design plans, which were more comprehensive than Team A's plan. As a result, Team N and M performed a more structured design analysis with less context switching. In the case of Team M, they have a higher requirement coverage than Team A given that both teams spent two full hours. Comparing the coverage of Team N and A,

although Team N had slightly less requirement coverage, they spent only half the time of Team A.

Kruger and Cross [27] suggest that in a solution-driven design more solutions are generated. However, when the design problem space is not well explored, the generated solution alternatives are also limited. Team A took a solution-driven approach and they were fixated on a given solution from the beginning. The focus of pushing/popping a queue by Team A dictated their design discussions for about an hour. This solution-driven design approach limited the discovery of new design issues and therefore limited the design alternatives that could have been considered.

Contextualization of a problem must be comprehensive in order to be useful. A systematic exploration of the context or design concerns helps software designers to identify design issues. A key challenge in software design is to find all relevant software design problems that arise from a combination of requirements, constraints, and quality attributes. The exploration of the problem space in an unknown domain by designers depends on effectively identifying related design concerns that lead to the identification of relevant design problems, i.e. finding out what you need to know. This process works in a spiral or co-evolution manner because the design solutions themselves become the context to next level of design.

Team M explored the design contexts substantially in relation to a design problem, whereas Team A discussed a certain context and switched to another design area before revisiting it again. The contexts of a design problem represent different aspects or concerns that influence the design problem, and they have to be considered together in order to define a single design problem properly. The sporadic treatment by Team A caused them to context switch before returning their focus to the key problem, making it ineffective in problem solving. This is also a sign of the lack of design structuring by Team A. The different approaches in design structuring and contextualization taken by Team A and M appeared to have influenced the effectiveness of software design. This observation supports in part the design reasoning theory in which Goldschmidt and Weil suggest that the coordination of contents and structure are important in design reasoning [16].

Our study has identified that Team A used more **scenario-driven reasoning** than Team M but Team M had higher requirement coverage than Team A. Studying the protocols, we have found that even though scenarios help design analysis, they do not lead directly to a design solution. A scenario by itself is inherently insufficient to formulate a complete solution, a scenario serves to give an example of how software may behave, and help to evaluate if an abstract design idea would work under different circumstances. Therefore, designers need to work in two levels of abstraction simultaneously, i.e. (a) instantiating scenarios to test an abstract design and (b) creating and refining an abstract design idea based on the scenarios. Having suggested that the two levels of abstraction are both important, we however cannot conclude how often scenario reasoning should be used. We postulate that this type of reasoning depends on the individual and the design problem involved, based on how well a designer can deal with the abstract design problem in a particular context.

Team M used more **explicit reasoning** than Team A. The two designers in Team M communicated their ideas typically through a cycle of problem proposition and solution formulation. In their discussions, they often explained themselves by using “because”. We suggest that such explicit reasoning help them to (a) clearly focus and align the discussion of a problem; (b) reduce any miscommunication of ideas. Examples of Team M’s explicit reasoning are at [0:18:46.1], [0:24:31.2], [0:30:42.8]. We further suggest that this makes the design process more effective and thus, it improves design effectiveness.

The proportion of **inductive reasoning** and deductive reasoning used by Team A and M were quite different. Team M used inductive reasoning to help them explore and find new design problems. The application of inductive reasoning appears to explain how new design problems are identified, which is important in software design since design problem identification is essential in the exploration of the problem space. Such exploration is essential in creating and finding suitable solutions for wicked design problems.

6. Research Validity

Protocol coding was done in three passes by the two authors with one week time between each pass to reduce coding errors. Since this work investigates different reasoning techniques, we need to interpret the reasoning techniques in the protocol coding, which are new and not well-defined. The interpretation process, especially on concepts such as inductive and deductive reasoning, thus required deliberations by the researchers. In each coding pass, we adjusted our interpretations of the reasoning techniques according to our improved understanding. The result was recoding the protocols. The decision maps were created based on the subject and the sequence of the

discussion. They have provided visual support to analyze problem structuring.

The findings from the protocol analysis and the decision maps have highlighted different aspects of design reasoning and their interplays. Our findings in areas such as problem structuring and problem-solution co-evolution are consistent with other design theories, which provide external validation to this work. Although interviews of the designers after the design sessions were made available, we decided not to view them before we had completed our analysis so that we would not be biased by designers' self-reporting.

7. Conclusion

Human design reasoning plays a major role in software design, but there is a limited understanding on how it works effectively. The use of reasoning techniques is important in software design at least in two ways: (a) design structuring influences the amount of context switching in design, thus affecting design effectiveness; (b) identification of relevant problems through inductive reasoning and other reasoning techniques helps designers create better design.

Common in the software industry, software designers use different reasoning techniques to design, with different design results. We have preliminary evidence to show that the appropriate use of contextualization helps designers to focus and explore key design problems effectively. Creating scenarios helps verify abstract design ideas, and explicit reasoning facilitates communication of design ideas. Inductive reasoning is essential in helping designers identify design problems in the problem space, thus facilitating design problem-solution co-evolution. Further exploration of inductive reasoning and other reasoning techniques will improve the way we design software because of the improved understanding of how we think and solve software design problems. This is fundamental to the software development processes.

8. References

[1] W. De Neys, "Implicit conflict detection during decision making," in *Proceedings of the Annual Conference of the Cognitive Science Society*, 2007, pp. 209-214.

[2] J. S. Evans, "In two minds: dual-process accounts of reasoning," *Trends in Cognitive Sciences*, vol. 7 (10), pp. 454-459, 2003.

[3] A. Tversky and D. Kahneman, "Rational Choice and the Framing of Decisions," *The Journal of Business*, vol. 59 (4 Part 2), pp. S251-S278, 1986.

[4] N. Cross, "Expertise in Design: An Overview," *Design Studies*, vol. 25 (5), pp. 427-441, 2004.

[5] 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.

[6] A. Tang, M.A. Barbar, I. Gorton, and J. Han, "A survey of architecture design rationale," *Journal of Systems and Software*, vol. 79 (12), pp. 1792-1804, 2006.

[7] J. Tyree and A. Akerman, "Architecture Decisions: Demystifying Architecture," *IEEE SOFTWARE*, vol. 22 (2), pp. 19-27, 2005.

[8] A. Dutoit, R. McCall, I. Mistrik, and B. Paech, Eds., *Rationale Management in Software Engineering*. Springer, 2006 pp. 432.

[9] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti, "Relating Software Requirements and Architectures Using Problem Frames," in *IEEE Joint International Conference on Requirements Engineering*, 2002, pp. 137-144.

[10] K. Dorst and N. Cross, "Creativity in the design space: co-evolution of problem-solution," *Design Studies*, vol. 22 (5), pp. 425-437, 2001.

[11] 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.

[12] A. Tang, Y. Jin, and J. Han, "A rationale-based architecture model for design traceability and reasoning," *Journal of Systems and Software*, vol. 80 (6), pp. 918-934, 2007.

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

[14] V. Goel and P. Pirolli, "The Structure of Design Problem Spaces," *Cognitive Science*, vol. 16 (3), pp. 395-429, 1992.

[15] H. W. J. Rittel and M. M. Webber, "Dilemmas in a general theory of planning," *Policy Sciences*, vol. 4 (2), pp. 155-169, 1973.

[16] G. Goldschmidt and M. Weil, "Contents and Structure in Design Reasoning," *Design Issues*, vol. 14 (3), pp. 85-100, 1998.

[17] B. Csapó, "The Development of Inductive Reasoning: Crosssectional Assessments in an Educational Context," *International Journal of Behavioral Development*, vol. 20 (4), pp. 609-626, 1997.

[18] M. Simina and J. Kolodner, "Creative design: Reasoning and understanding," in *Case-Based Reasoning Research and Development*. vol. 1266/1997, ed: Springer Berlin / Heidelberg, 1997, pp. 587-598.

[19] K. J. Klauer, "Teaching Inductive Thinking to Highly Able Children," in *Fostering the growth of high ability: European perspectives*, A. J. Cropley and D. Dehn, Eds., ed: Greenwood Publishing Group, 1996, pp. 175-191.

[20] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-Based Analysis of Software Architecture," *IEEE Software*, vol. 13 (6), pp. 47-55, 1996.

[21] J. Gero and T. McNeill, "An Approach to the Analysis of Design Protocols," *Design Studies*, vol. 19 (1), pp. 21-61, 1998.

[22] R. Yin, *Case Study Research Design and Methods*, 3rd ed.: Sage Publications, 2003.

[23] D. A. Schön, "Designing: rules, types and worlds," *Design Studies*, vol. 9 (3), pp. 181-190, 1988.

[24] B. S. Shum and N. Hammond, "Argumentation-Based Design Rationale: What Use at What Cost?," *International Journal of Human-Computer Studies*, vol. 40 (4), pp. 603-652, 1994.

[25] A. Tang, J. Han, and R. Vasa, "Software Architecture Design Reasoning: A Case for Improved Methodology Support," *IEEE Software*, vol. Mar/Apr 2009 pp. 43-49, 2009.

[26] P. G. Rowe, *Design thinking*: Cambridge, Mass.:MIT Press, 1987.

[27] C. Kruger and N. Cross, "Solution driven versus problem driven design: strategies and outcomes," *Design Studies*, vol. 27 (5), pp. 527-548, 2006.