How do Graduating Students Evaluate Software Design Diagrams?

Prajish Prasad
Interdisciplinary Programme in Educational Technology
Indian Institute of Technology Bombay
Mumbai, India
prajish.prasad@iitb.ac.in

Sridhar Iyer
Interdisciplinary Programme in Educational Technology
Indian Institute of Technology Bombay
Mumbai, India
sri@iitb.ac.in

ABSTRACT

An important skill graduating computing students require is to evaluate a given software design and ensure that it satisfies the intended requirements. Prior work has shown that while working with software designs, experts think deeply about the design and simulate scenarios where the design does not satisfy the requirements. In this paper, we examine how students evaluate a given set of software design diagrams (UML class and sequence diagrams) against the given requirements.

We conducted a study with 100 final year computing undergraduate students. Each student was given a sheet which contained the requirements and the design diagrams for a problem. They were asked to identify defects based on the requirements. We used an interpretative content analysis method to come up with categories of student written responses. Our findings show that some students were able to identify scenarios which do not satisfy the requirements (24%) and identify necessary functions which were missing in the design (14%). However, we also found that certain students included new functionalities in the design (10%) and modified existing functionalities (22%) which were unrelated to the given requirements. These findings give insights to what students focus on and the mental models they create while evaluating software design diagrams.

CCS CONCEPTS

• **Social and professional topics → Software engineering education**.

KEYWORDS

evaluating design diagrams, class diagram, sequence diagram, simulating scenarios, mental models

ACM Reference Format:
Prajish Prasad and Sridhar Iyer. 2020. How do Graduating Students Evaluate Software Design Diagrams?. In *Proceedings of the 2020 International Computing Education Research Conference (ICER '20), August 10–12, 2020, Virtual Event, New Zealand*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3372782.3406271

ICER '20, August 10–12, 2020, Virtual Event, New Zealand

ACM ISBN 978-1-4503-7092-9/20/08…$15.00
https://doi.org/10.1145/3372782.3406271

1 INTRODUCTION

The first stage of the software development cycle is gathering of requirements from various stakeholders. Designers then come up with the conceptual design of a software system that satisfies these requirements. This conceptual design is often modeled using Unified Modeling Language (UML) diagrams [34]. These UML diagrams specify behaviors and scenarios of the system across various levels of abstraction. It is essential that these diagrams represent the actual working of the system and satisfy the intended requirements of the system. A failure to do so can lead to inconsistent and incorrect designs. Hence, it is necessary that designers develop the ability to accurately reason about a software system design, and evaluate it against the given requirements.

When students graduate and enter the software industry, they usually start working on existing projects and spend their first several months resolving bugs and writing additional features based on new requirements [6, 12]. When new requirements are provided, they need to have an integrated understanding of the design in order to add features into the design. This requires students to comprehend and reason with an already existing design, incorporate the required feature in the design and evaluate if the design satisfies the intended requirements.

Expert software designers use various reasoning techniques such as simulating scenarios in the given problem [19, 39], constraints consideration and trade-off analysis [19] while creating and evaluating designs. However, sufficient emphasis has not been given on teaching and learning of such reasoning techniques and practices in a software design course, and hence graduating students find it difficult to critically analyze an existing design and improve upon it [7].

In this paper, we investigate how graduating students respond when asked to evaluate a set of design diagrams against the given requirements. We conducted a study with 100 final year undergraduate computing students and asked them to identify defects based on the given requirements and design diagrams. We analyzed the data using an interpretive content analysis method to come up with categories based on student responses. Based on these findings, we provide implications for further research as well as teaching-learning of software design.

2 BACKGROUND

2.1 Student difficulties in design

While prior work has explored difficulties students face while creating designs [10, 15, 28, 36, 37, 40], difficulties students face while

evaluating software designs against the requirements have not been given sufficient emphasis.

Studies have shown that a majority of graduating students are not competent in designing software systems [15, 28]. Students do not take efforts to describe the behavior of the system [28], have difficulties in understanding the purpose and relationships between various diagrams [36, 37], and find it difficult to abstract real world problems [36]. Students are also unable to form an integrated understanding of the design diagrams. They produce incomplete class diagrams, sequence diagrams with missing responsibilities and objects at inconsistent abstraction levels [36].

2.2 Strategies for evaluating designs

Evaluating a software design is an important practice of experts. If a whole design episode is considered, experts spend significant time evaluating their solution [29]. Experts do a sound, preliminary evaluation of their tentative design decisions before implementation, as opposed to novices who use trial-and-error techniques to evaluate their design through several iterations [3]. Experts use various reasoning techniques such as simulating scenarios in the given problem [19, 39], constraints consideration and trade-off analysis [19] while reasoning about the design. Studies have also shown that they create more detailed mental models of the system to be designed [2, 11]. They use mental models right from the conception of the problem, even when the design is still quite abstract. These models are incrementally transformed and made more concrete as the design progresses. They repeatedly conduct mental simulation runs on their partially completed designs [2].

Prior work has explored the evaluation of design diagrams in the context of reading, comprehension, and maintainability of UML diagrams [8, 9, 16, 17, 24, 30]. Studies examining comprehension of UML diagrams have looked at which diagrams (and in what combination) lead to better diagram comprehension [18, 32, 38], and the effectiveness of different forms of notation [23, 24, 31]. Studies have also examined how providing UML diagrams along with source code have led to better comprehension and defect identification in source code [4, 14, 35].

Strategies for evaluating design diagrams have also been explored in the context of inspecting design diagrams in order to identify defects. Inspecting design diagrams involve reviewing different diagrams in order to create an integrated model of the design. Hungerford et al. [21] conducted a study with 12 experienced developers who were asked to perform individual reviews on a software design. The results indicate search patterns that rapidly switch between design diagrams are most effective. Kim et al. [22] proposed a theoretical framework which focuses on perceptual and cognitive processes while reasoning with multiple diagrams. A perceptual process is a bottom-up activity of sensing something and knowing its meaning and value. A conceptual process is a top-down activity which is used to generate, refine and validate hypotheses based on the provided design diagrams. Successful participants' perceptual integration processes involved making several round-trip transitions, enabling them to relate information from different diagrams. Conceptual integration processes involved creating and refining several hypotheses while inspecting diagrams. Haisjackl et al. [20] explored strategies which subjects with varying experience used while inspecting BPMN models. Common strategies which were identified were - (1) getting an overview of the model and then checking the whole model for quality issues (2) directly start with reading the model while checking for quality issues and (3) look for specific types of quality problems across the model.

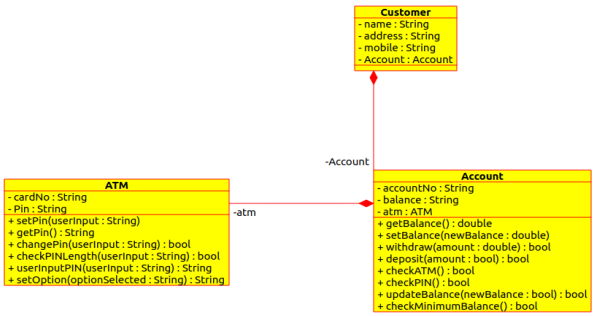

Figure 1: Class diagram of the ATM system

A common characteristic seen in all these effective strategies is that experts create an integrated model of the design diagrams and perform mental simulations on them in order to evaluate their models. Hence, in order to effectively evaluate a design and ensure that it satisfies the requirements, students should be able to analyze and reason about the given requirements, develop an integrated mental model of the given design diagrams, and simulate scenarios in the design based on the given requirements. This background forms the basis of our study, where we examine how students evaluate software design diagrams.

3 RESEARCH DESIGN

3.1 Research Question

The research question guiding this study is - "How do graduating students evaluate software design diagrams against the given requirements?"

3.2 Description of the evaluation task

The requirements and design of an ATM system was given to students. The requirements provided to students are as follows:

(1) A user with a valid account can register his/her ATM card and set a PIN if he/she has not set a PIN yet. The PIN should be of length 4 and should contain only numbers.
(2) When the user enters the ATM card and inputs the correct PIN, the following options are shown.
   - Withdraw - The user can withdraw money from his/her account. If the balance is less than Rs.1000, withdrawal is denied.
   - Change PIN - The user can change his/her PIN by entering the previous PIN correctly.

The provided design consists of a class diagram (Figure 1) and three sequence diagrams - Register (Figure 2), Withdraw (Figure 3) and Change PIN (Figure 4).

The task given to students was - *"Identify defects (if any) in the following design diagrams based on the requirements. For each*

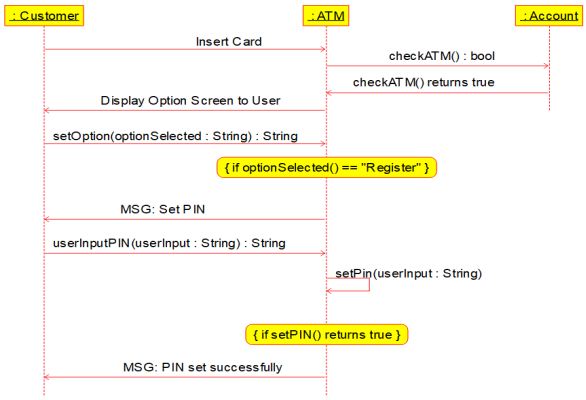

Figure 2: Sequence diagram for registering the ATM card

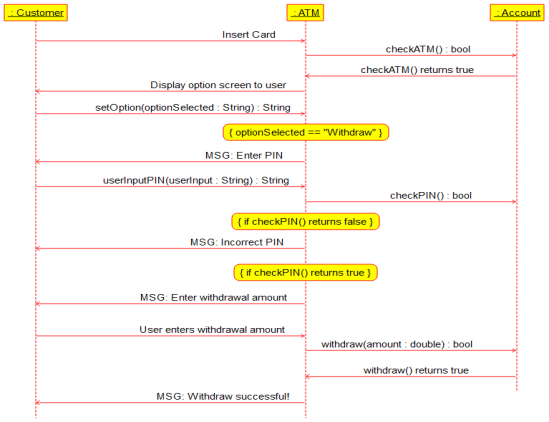

Figure 3: Sequence diagram for withdrawing amount

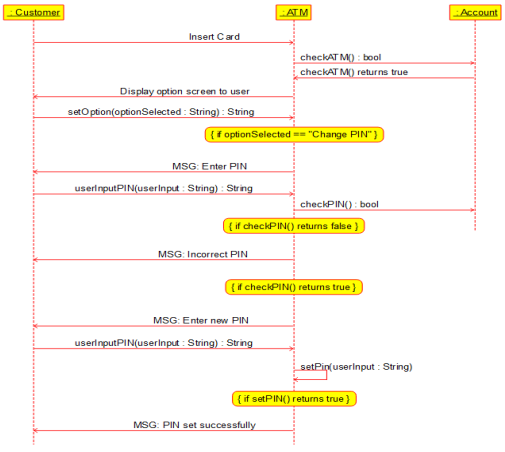

Figure 4: Sequence diagram for changing pin

defect, provide a logical explanation of why you think it is a defect.". The requirements, design diagrams and the task were provided to students on paper.

The ATM system example was chosen, as we wanted students to be familiar with the problem domain and correspond to something they have used in their daily lives. We restricted our focus to only two types of UML diagrams - class and sequence diagrams, primarily for two reasons. Firstly, the class and sequence diagram correspond to the structural and behavioral categories of design diagrams respectively. Hence we believe these diagrams can serve as a good starting point to examine students' responses. Secondly, these diagrams are more commonly used [13, 25] and the first diagrams that students learn about in a software design course. Hence, most graduating students would have had sufficient exposure to these diagrams.

3.3 Deficiencies introduced in the design

The design diagrams were seeded with certain deficiencies. We focused on including semantic deficiencies rather than syntactic deficiencies in the design. The notion of syntactic and semantic deficiencies have been adapted from the work of Lindland et al. [26] which defines a framework for understanding quality in conceptual modeling. Syntactic quality ensures that all statements in the model are according to the syntax of the language. Hence, syntactic deficiencies occur when there are mistakes corresponding to the syntax and naming conventions in UML diagrams *(Eg: Does the same class have the same attributes in different diagrams, is the return type of a class variable correct and consistent across all diagrams etc.).* Syntactic deficiencies can be uncovered by a superficial search on the design diagrams. Semantic quality refers to how faithfully the modeled system is represented. Semantic deficiencies occur when certain aspects of the requirements are not conveyed in the design diagrams. Detecting semantic deficiencies require students to think deeply about the design, understand the relationship between different diagrams and evaluate the diagrams against the given requirements.

In the given ATM system design, certain semantic deficiencies introduced in the design are as follows:

- There is no check if user is already registered.
- The pin requirements are not checked during registration and change pin.
- The minimum balance requirement is not checked during withdrawal.
- There is no check if the withdrawal amount is greater than the balance.
- The balance is not updated after withdrawal.

These semantic deficiencies can be uncovered by examining the requirements and the diagrams and simulating scenarios where the design does not satisfy the requirement. Based on previous pilot studies with students, we found that students have difficulty in simulating such alternate scenarios. Hence, we wanted to confirm if these difficulties hold with a larger group of students as well.

3.4 Study Procedure

The study was conducted with 100 final year (fourth year) computer engineering and information technology engineering students (61

male and 39 female), in their own institution. The medium of instruction in their institution is English, hence all participants were comfortable in reading, writing and speaking in English. Their average age was 21 years. The engineering institution is located in a metropolitan city in our country. Participation in the study was completely voluntary, and participants could withdraw from the study at any point during the study. A few days prior to the study, participants had to fill a registration form with their basic information like name, branch, overall percentage in the last semester, and rate their confidence in their understanding of object-oriented design, class and sequence diagrams. All participants had undergone a Software Engineering course in the previous semester, and hence were familiar with class diagrams and sequence diagrams.

Each student was given the task sheet which contained the requirements, the design diagrams and a question asking them to identify defects. The first author was available to answer any doubts during the session, but did not provide any hints or answers to students. External resources like textbooks, references and use of internet were not provided to students.

3.5 Data Analysis

We analyzed the answers of students using an interpretative content analysis method [5] to come up with categories for the defects identified. The steps which we followed to analyze the data is as follows.

(1) Representation of student answers - The written text which each participant wrote as a response to the question was typed verbatim to a spreadsheet. We considered a written sentence or group of sentences which referred to a particular defect, as the unit of analysis. Each row in the spreadsheet corresponds to a sentence/sentences written by participants. Out of the 100 students, 1 student's response was not clear and hence was excluded from the analysis. Many students listed multiple defects. We identified 168 answers from 99 student response sheets.
(2) Descriptive coding - Each sentence was assigned a descriptive code. The objective of descriptive coding is to avoid any prejudices or preconceptions and stay close to the data. For example, for the following student response - *"In the Register Pin Sequence diagram, we do not use checkPinlength() before setPin(). By doing so, a user may add length(PIN) != 4 breaking the criteria."*, the descriptive code assigned was *"checkPinLength() function is not used before setPin() in register sequence diagram"*
(3) Generate categories and sub-categories - Categories and sub-categories are inferred based on the patterns, explanations and relationships between the descriptive codes. The categories reflect the meaning inferred from the descriptive codes and can explain larger segments of data. During this process, memos and notes are created which describe the category definitions and criteria for assigning descriptive codes to these categories. For example, in the descriptive code provided above, the sub-category inferred is *"checkPinLength() function is not called"*, and the category is *"Identify necessary functions which are not used."*

Table 1: Response categories and number of student responses in each category

Category of student response	Number of student responses
Identify scenarios which do not satisfy requirements	41
Identify necessary functions which are not used	24
Change existing functionalities and requirements	37
Add new functionality	18
Change data types, functions of class diagram	21
Blank/No defects	27
Total	**168**

The first author performed descriptive coding and came up with the initial set of categories and sub-categories. In order to establish reliability of the generated categories, two rounds of coding were done with a rater (non-author). In the first round, the author provided the rater with 20 student responses and the list of categories. After independently assigning categories to the given set of responses, both raters discussed the categories corresponding to each response, refined category names and definitions, and came to an agreement on conflicting entries. In the second round, both raters independently assigned categories to another 10 responses and reached near agreement (90%). Based on the refined categories and definitions, the first author independently assigned categories to the remaining responses.

4 FINDINGS: CATEGORIES OF STUDENT RESPONSES

In this section, we describe the responses which students provided when asked to identify defects in the design diagrams based on the requirements. The categories and number of student responses in each category is shown in Table 1. Definitions and examples of categories and sub-categories are summarized in Table 2. The percentage of responses in each category is represented in Figure 5 and mentioned in brackets next to each category.

Category 1: Identify scenarios which do not satisfy the requirements (24.4%)

All responses in which students explicitly mentioned scenarios where the design is not satisfying the requirements are placed into this category. The scenarios which students were able to identify (which were not satisfying the requirements) were - (1) The balance not being checked during withdrawal, (2) An already registered user registering again (3) When withdrawal is greater than balance and (4) PIN requirements not being checked.

None of the students could identify all the above mentioned scenarios. 23 students identified exactly one scenario. Only 7 students identified exactly two scenarios.

Category 2: Identify necessary functions which are not used (14.3%)

Students explicitly mentioned that essential functions present in the class diagrams are not being used in the sequence diagram, and hence not satisfying a particular requirement. Only responses where students have explicitly mentioned the function name has

Table 2: Categories of defects based on analysis of student responses

Defect Category	Definition	Sub-categories	Example
Identify scenarios which do not satisfy the requirement	Students identified a scenario and it is used to check if requirements are satisfied	1. Identify scenario where balance is not checked	*"In withdraw sequence diagram, after subtracting the withdrawal amount if balance is less than 1000 then withdrawal should be unsuccessful"*
		2. Identify scenario where user is is already registered	*"During PIN registration, there is no option to check if user has already registered. This might lead to double registrations"*
		3. Identify scenario where withdrawal is greater than balance	*"It should also check if withdrawal amount is less than available amount"*
		4.Identify scenario where PIN requirements are not checked	*"For register PIN sequence diagram, length or datatype of PIN not checked"*
Identify necessary functions which are not used	Students explicitly mentioned that a particular function is not used	1. checkMinimumBalance() not used 2. updateBalance() not used 3. checkPinLength() not used	*"Withdraw: - checkMinimumBalance() and updateBalance() are not called"* *"check Pin length() needs to be used both while registering and changing PIN"*
Change existing functionalities and requirements	Students change an already existing functionality by adding a sub-task	1. Change in functionality provided to user	*"In the withdrawal system, PIN should be entered after amount is entered."*
		2. Change in information provided to user	*The minimum amount to be present in the account must be displayed for user's information*
Add new functionalities	Students add a completely new functionality, which is unrelated to any of the requirements provided	-	*"There should also be a way to handle things if a user forgets his/her PIN"* *"First time register PIN doesn't have any 2-step authentication"* *"Defect in provided scenario is that there is no such option of language preferences."*
Changes in data types, functions and structure of the class diagram	Students suggested changes in datatypes, and structure of class diagram	1. Change in data type of variable	*"Datatype of Card no / Account no / PIN should be int and not string"*
		2. Variables missing in class diagram	*"Users, accounts and ATMs are missing an ID variable"*
Blank/No defects	Students left the sheet blank or explicitly stated there were no defects.	-	-

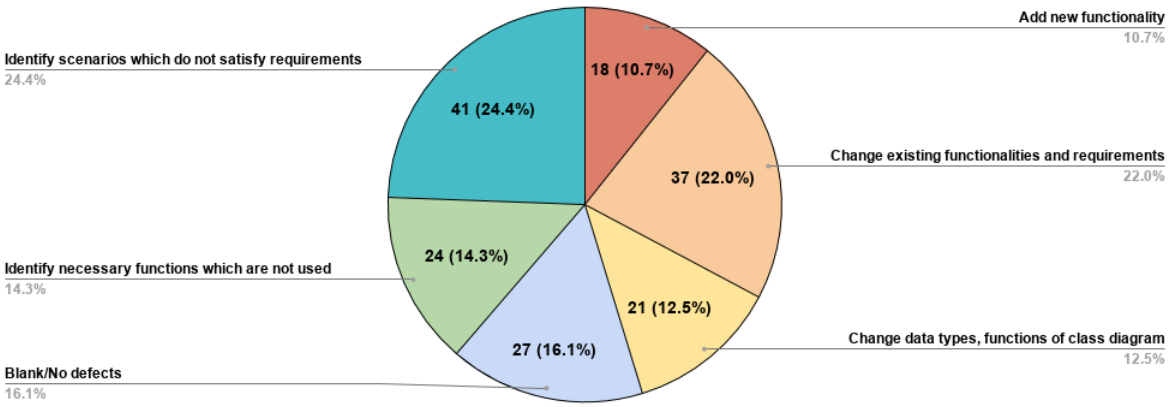

Figure 5: Pie chart of response categories based on content analysis of 168 student responses

been included in this category. Students mentioned three functions which were missing: 1. checkMinimumBalance() 2. checkPinLength() 3. updateBalance(). 12 students identified that at least one function was missing, out of which 5 could identify all three missing functions.

In the above two categories, students are evaluating the design diagrams against the requirements by identifying alternate scenarios and simulating function execution.

Category 3: Change existing functionalities and requirements (22%)

29 students suggested improvements to existing functionalities as well as change of requirements. All responses where students change the requirement, or add a sub-function to an existing functionality have been assigned to this category. For example, students suggested that (1) the PIN should be entered after entering withdrawal amount, rather than before (2) Users should be asked to re-enter PIN in case of invalid PIN (3) Bank should provide PIN along with the ATM card. One student even suggested that the requirement of minimum balance amount of Rs. 1000 should not be present, hence suggesting a change in requirement. Another subcategory of student response is to provide appropriate responses to the user. For example, students suggested that users should be provided with (1) Balance display before/after withdrawal (2) Appropriate error messages for withdrawal rejection (3) Information about the minimum balance requirement.

Category 4: Add new functionalities in the design (10.7%)

Students also introduced new functionalities to the design, rather than identifying defects based on the requirements. We categorize all responses into this category if students are adding a completely new functionality, which is unrelated to any of the requirements provided.

We found that 16 students added new functionalities when asked to verify designs. These included varied functionalities like - One-Time-Password (OTP) authentication, Forgot PIN functionality, using email and mobile number for registration, language preferences, deposit functionality, view current balance functionality, viewing the mini-statement and many others.

Students adding, modifying and changing functionalities and requirements are not wrong in itself; in fact many are essential for an ATM system. However, students added functionalities which were unrelated to the given requirements, as opposed to evaluating the design against the requirements.

Category 5: Change data types, functions and structure of the class diagram (12.5%)

Another category of student responses were based on changes in the class diagram and change in data type of variables. For example, students incorrectly stated that datatype of card number, account number, balance and PIN should be 'int' rather than 'string'. Students stated that some variables are missing in the class diagram as well as suggested structural changes in the class diagram. For example, one student suggested that there should be an aggregation sign instead of a composition symbol in the class diagram. Another commented on the absence of one-to-one, many-to-many relationships between classes.

Category 6: Blank responses and no defects (16.1%)

27 students either did not give any response (23 students) or explicitly mentioned that there were no defects (4 students). None of them mentioned any reasons for why they did not find any defects. We ascertain that they either did not understand the evaluation task, or they examined the diagrams and could not identify defects.

5 DISCUSSION

The categories of student responses can further be grouped based on what elements of the design diagrams students focus on during the evaluation task. They focus on syntactic elements, on semantic

elements, or on new requirements and functionalities. Figure 6 depicts this grouping in the form of a Venn diagram.

Focus on syntactic elements in the design. 13 students identified defects based on the syntactic elements in the design diagrams (Category 5). These included changes in data type of variables, and structural changes in the class diagram. Out of the 13, 5 students exclusively focused on syntactic defects, and did not identify other types of defects. This can indicate that they restricted their focus to a surface-level comprehension of the design diagrams, without simulating scenarios in the design. This behavior is similar to the concept of "surface knowledge" in program comprehension - *"program knowledge concerning operations and control structures reflect surface knowledge, i.e. knowledge that is readily available by looking at a program. In contrast, knowledge concerning data flow and function of the program reflect deep knowledge which is an indication of a better understanding of the code"* [33]. Students had sufficient surface knowledge about the structure of the class and sequence diagrams, but they did not simulate scenarios based on the design to identify semantic defects.

Focus on semantic elements in the design. Our findings show that 41 students (out of 99) identified scenarios where the design is not adhering to the requirements and identified necessary functions which were not used (Category 1 and 2). Out of the 41, 20 students exclusively focused on these semantic defects. They made interconnections between different diagrams by identifying data variables and functions in the class diagram and checked which functions were missing or not needed in the sequence diagram. This integrated understanding of the design diagrams ensured that they adhered closely to the design, which enabled them to simulate scenarios pertaining to the given requirements and identify semantic defects in the design.

Focus on new elements which are absent in the design. 43 students either mentioned adding new functionalities to the design or changed the existing functionalities and requirements (Category 3 and 4). Out of the 43, 24 students exclusively focused on adding or changing functionalities and did not identify other types of defects. We hypothesize that this behavior of adding functionalities is due to students' prior knowledge of the problem space (i.e., the ATM system). Students encounter several features of the ATM like One-Time-Password (OTP), Two-factor authentication etc. in their daily lives. When students are presented with the current design, the problem domain model is already available in their mind and hence they form hypotheses about the expectations of the design. Students compare the design with their own model, simulate alternate scenarios and generate hypotheses of how they envision the ATM system to function. When there is a mismatch between their hypothesis and the actual design, students flag these as defects. This behavior corresponds to the conceptual processes mentioned by Kim et al. [22] which are used to generate, refine and validate hypotheses based on the provided design diagrams. Adding new functionalities indicate that students are not evaluating the design against the given requirements. They might find it easier to simulate scenarios pertaining to new functionalities rather than deeply examine the design and the requirements.

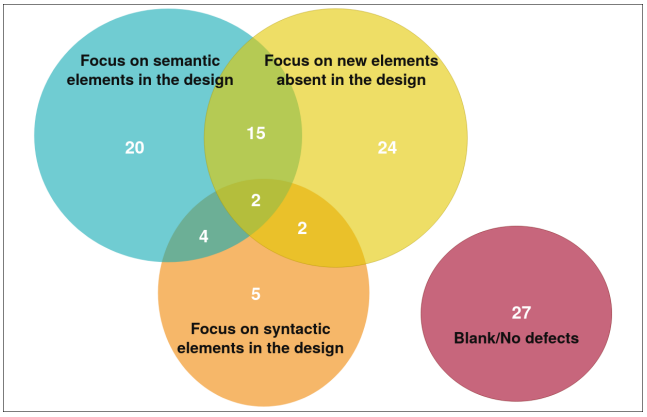

Figure 6: Venn diagram depicting which aspects of the design diagrams students focus on. For example: 5 students focused exclusively on syntactic defects. 2 students focused on all three types. (n=99)

It is to be noted that adding and modifying existing functionalities in itself is not an undesirable behavior. When experts are asked to create designs from requirements, they expand both the problem and the solution space. Experts expand the problem space by simulating alternate scenarios and hence derive new requirements and constraints which are not stated in the problem [1]. The solution space can be expanded by generating alternative solutions using various techniques like brainstorming, analogous thinking, attribute listing, etc. [27]. They then evaluate several alternatives and come up with a suitable design based on identified constraints. However, the task presented to students was different from a design creation task. For this task, the objective was to identify defects based on the requirement, and hence adding or changing functionalities was not the goal.

We hypothesize that these categories of responses also correspond to variation in students' mental model of the design and how they perform simulations on these models. An integrated model enables students to simulate scenarios based on the design diagrams. When students focus on adding new functionalities, they create a model not completely adhering to the design diagrams and hence simulate scenarios not present in the design.

6 LIMITATIONS

There are certain limitations of this study. We have constrained the construct "evaluation of design diagrams" to evaluating semantic deficiencies in the design diagrams. There are several other aspects which can refer to evaluation - evaluating for syntactic deficiencies or evaluating properties of the design such as its modularity and extensibility. These aspects have not been addressed in this study.

We have restricted our focus to two types of design diagrams - class and sequence diagrams. It is unclear if the type of defects identified by students will be similar if other design diagrams are introduced. We have assumed that all students have sufficient prior knowledge about class and sequence diagrams, since they had taken a software design course in the previous semester. The registration form capturing students' self-perception of their UML diagram

knowledge also indicated that all students were familiar with the class and sequence diagrams. However, differences in prior knowledge and experience in working with designs could have affected the results of the study.

The design provided to students was simplistic and does not capture the inherent complexity of an ATM system. However, the finding that many students were still not able to evaluate the design against the requirements provides us with indicators of what difficulties students face even with simplistic designs.

Students creating a mental model and simulating scenarios on this model has been inferred based on students' written responses. Conducting a think-aloud or stimulated recall interviews with a subset of students would have given us additional data for triangulation as well as insights to their mental model and processes. We were unable to do so due to logistical constraints and unavailability of students.

7 FUTURE WORK AND IMPLICATIONS

This study provides insights to how students evaluate a given design. We intend to conduct further qualitative studies with students, in order to understand how they make sense of different design diagrams and what strategies they use to evaluate a design against the given requirements.

We also intend to develop pedagogical strategies to train students in design evaluation. We believe that training students to evaluate design diagrams can help them better understand the interrelationships between these diagrams and develop an integrated understanding of the design.

These findings also provide the basis for computing education researchers to conduct similar studies exploring how students evaluate a design. Researchers can explore other characteristics of evaluation and how students evaluate design diagrams of varying complexity in unfamiliar domains. These studies can also give insights to the characteristics of students' mental models and further contribute to a theory of how students evaluate design diagrams against the given requirements.

REFERENCES

[1] Beth Adelson and Elliot Soloway. 1985. The role of domain expenence in software design. *IEEE Transactions on Software Engineering* 11 (1985), 1351–1360.

[2] Beth Adelson and Elliot Soloway. 1986. A model of software design. *International Journal of Intelligent Systems* 1, 3 (1986), 195–213.

[3] Saeema Ahmed, Ken M Wallace, and Lucienne T Blessing. 2003. Understanding the differences between how novice and experienced designers approach design tasks. *Research in engineering design* 14, 1 (2003), 1–11.

[4] Erik Arisholm, Lionel C Briand, Siw Elisabeth Hove, and Yvan Labiche. 2006. The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering* 32, 6 (2006), 365–381.

[5] Leslie A Baxter. 1991. Content analysis. *Studying interpersonal interaction* 239 (1991), 254.

[6] Andrew Begel and Beth Simon. 2008. Struggles of new college graduates in their first software development job. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 226–230.

[7] Eric Brechner. 2003. Things they would not teach me of in college: what Microsoft developers learn later. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 134–136.

[8] David Budgen, Andy J Burn, O Pearl Brereton, Barbara A Kitchenham, and Rialette Pretorius. 2011. Empirical evidence about the UML: a systematic literature review. *Software: Practice and Experience* 41, 4 (2011), 363–392.

[9] Michel RV Chaudron, Werner Heijstek, and Ariadi Nugroho. 2012. How effective is UML modeling? *Software & Systems Modeling* 11, 4 (2012), 571–580.

[10] Stanislav Chren, Barbora Buhnova, Martin Macak, Lukas Daubner, and Bruno Rossi. 2019. Mistakes in UML diagrams: analysis of student projects in a software engineering course. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training*. IEEE Press, 100–109.

[11] Bill Curtis, Herb Krasner, and Neil Iscoe. 1988. A field study of the software design process for large systems. *Commun. ACM* 31, 11 (1988), 1268–1287.

[12] Barthélémy Dagenais, Harold Ossher, Rachel KE Bellamy, Martin P Robillard, and Jacqueline P De Vries. 2010. Moving into a new software project landscape. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 275–284.

[13] Brian Dobing and Jeffrey Parsons. 2006. How UML is used. *Commun. ACM* 49, 5 (2006), 109–113.

[14] Wojciech J Dzidek, Erik Arisholm, and Lionel C Briand. 2008. A realistic empirical evaluation of the costs and benefits of UML in software maintenance. *IEEE Transactions on software engineering* 34, 3 (2008), 407–432.

[15] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Carol Zander. 2006. Can graduating students design software systems? *ACM SIGCSE Bulletin* 38, 1 (2006), 403–407.

[16] Ana M Fernández-Sáez, Marcela Genero, Danilo Caivano, and Michel RV Chaudron. 2016. Does the level of detail of UML diagrams affect the maintainability of source code?: a family of experiments. *Empirical Software Engineering* 21, 1 (2016), 212–259.

[17] Marcela Genero, Ana M Fernández-Saez, H James Nelson, Geert Poels, and Mario Piattini. 2011. Research review: a systematic literature review on the quality of UML models. *Journal of Database Management (JDM)* 22, 3 (2011), 46–70.

[18] Chanan Glezer, Mark Last, Efrat Nachmany, and Peretz Shoval. 2005. Quality and comprehension of UML interaction diagrams-an experimental comparison. *Information and Software Technology* 47, 10 (2005), 675–692.

[19] Raymonde Guindon. 1990. Knowledge exploited by experts during software system design. *International Journal of Man-Machine Studies* 33, 3 (1990), 279–304.

[20] Cornelia Haisjackl, Pnina Soffer, Shao Yi Lim, and Barbara Weber. 2018. How do humans inspect BPMN models: an exploratory study. *Software & Systems Modeling* 17, 2 (2018), 655–673.

[21] Bruce C Hungerford, Alan R Hevner, and Rosann Webb Collins. 2004. Reviewing software diagrams: A cognitive study. *IEEE Transactions on Software Engineering* 30, 2 (2004), 82–96.

[22] Jinwoo Kim, Jungpil Hahn, and Hyoungmee Hahn. 2000. How do we understand a system with (so) many diagrams? Cognitive integration processes in diagrammatic reasoning. *Information Systems Research* 11, 3 (2000), 284–303.

[23] Christian FJ Lange and Michel RV Chaudron. 2004. An empirical assessment of completeness in UML designs. In *Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering (EASE âĂŸ04)*. IET, 111–121.

[24] Christian FJ Lange and Michel RV Chaudron. 2006. Effects of defects in UML models: an experimental investigation. In *Proceedings of the 28th international conference on Software engineering*. 401–411.

[25] Christian FJ Lange, Michel RV Chaudron, and Johan Muskens. 2006. In practice: UML software architecture and design description. *IEEE software* 23, 2 (2006), 40–46.

[26] Odd Ivar Lindland, Guttorm Sindre, and Arne Solvberg. 1994. Understanding quality in conceptual modeling. *IEEE software* 11, 2 (1994), 42–49.

[27] Zhiqiang Liu and Dieter J Schonwetter. 2004. Teaching creativity in engineering. *International Journal of Engineering Education* 20, 5 (2004), 801–808.

[28] Chris Loftus, Lynda Thomas, and Carol Zander. 2011. Can graduating students design: revisited. In *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 105–110.

[29] Thomas Mc Neill, John S Gero, and James Warren. 1998. Understanding conceptual electronic design using protocol analysis. *Research in Engineering Design* 10, 3 (1998), 129–140.

[30] Ariadi Nugroho. 2009. Level of detail in UML models and its impact on model comprehension: A controlled experiment. *Information and Software Technology* 51, 12 (2009), 1670–1685.

[31] Ariadi Nugroho and Michel RV Chaudron. 2008. A survey into the rigor of UML use and its perceived impact on quality and productivity. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. 90–99.

[32] Mari Carmen Otero and José Javier Dolado. 2004. Evaluation of the comprehension of the dynamic modeling in UML. *Information and Software Technology* 46, 1 (2004), 35–53.

[33] Nancy Pennington. 1987. Comprehension strategies in programming. In *Empirical studies of programmers: second workshop*. Ablex Publishing Corp., 100–113.

[34] James Rumbaugh, Ivar Jacobson, and Grady Booch. 2004. *Unified modeling language reference manual, the*. Pearson Higher Education.

[35] Giuseppe Scanniello, Carmine Gravino, and Genoveffa Tortora. 2012. Does the combined use of class and sequence diagrams improve the source code comprehension? results from a controlled experiment. In *Proceedings of the Second Edition of the International Workshop on Experiences and Empirical Studies in Software Modelling*. 1–6.

[36] Ven Yu Sien. 2011. An investigation of difficulties experienced by students developing unified modelling language (UML) class and sequence diagrams. *Computer Science Education* 21, 4 (2011), 317–342.
[37] Dave R Stikkolorum and Michel RV Chaudron. 2016. A Workshop for Integrating UML Modelling and Agile Development in the Classroom. In *Proceedings of the Computer Science Education Research Conference 2016*. ACM, 4–11.
[38] Jennifer Swan, Trevor Barker, Carol Britton, and Maria Kutar. 2005. An empirical study of factors that affect user performance when using uml interaction diagrams. In *2005 International Symposium on Empirical Software Engineering, 2005*. IEEE, 10–pp.
[39] Antony Tang, Aldeida Aleti, Janet Burge, and Hans van Vliet. 2010. What makes software design effective? *Design Studies* 31, 6 (2010), 614–640.
[40] Benjy Thomasson, Mark Ratcliffe, and Lynda Thomas. 2006. Identifying novice difficulties in object oriented design. *ACM SIGCSE Bulletin* 38, 3 (2006), 28–32.