Fostering Software Design Evaluation Skills in Students using a Technology-enhanced Learning Environment

Submitted in partial fulfillment of the requirements
of the degree of

Doctor of Philosophy

by

Prajish Prasad
(Roll No. 154380001)

Supervisor:
Prof. Sridhar Iyer

Interdisciplinary Programme in Educational Technology
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
2021

Dedicated to my family

Abstract

Evaluating a software design is an important practice required of software engineering students. When students graduate and enter the software industry, they usually work on large existing systems, and spend their first several months resolving bugs and writing additional features based on new requirements. This requires students to comprehend an already existing design, incorporate the required feature in the design, and evaluate if the design satisfies the intended goals. The design of a software system is often specified as a set of Unified Modelling Language (UML) diagrams, which describe different views of the system, such as the structural view (e.g. class diagrams), and the behavioural view (e.g. sequence diagrams). However, students face difficulties in understanding how the overall system specifications actually work based on these views. Moreover, current software design courses do not place sufficient emphasis on teaching students how to evaluate designs.

The broad research goal of this thesis is to "Design and develop a technology-enhanced learning environment (TELE) which enables students to evaluate a software design against the given requirements". Evaluating a given design can be viewed from different perspectives. These include checking for its syntactic quality (whether the UML diagrams are modelled according to the syntax of the language), semantic quality (how well the design maps to the given requirements), or pragmatic quality (how well a given design can be interpreted by different stakeholders). In this thesis, we have focussed on enabling students to evaluate a design by checking for its semantic quality. This requires students to think deeply about the design, and understand the relationship between different diagrams in the design.

In order to answer this research goal, we began by analysing existing literature to identify student difficulties in the design process, as well as practices and strategies experts use to evaluate a given software design. We then conducted two studies with students to understand how they evaluate a given software design, and the difficulties they face. The key insight which we gained from the literature review and novice studies is that effective evaluation of design diagrams depends on the quality of mental models that students create based on the requirements and the design. Experts create a rich mental model of the system and simulate various scenarios of system behaviour in the design. Experts' mental models contain information regarding the control and data flow on simulation of such scenarios. However, novices were unable to simulate such scenarios, and their models focussed on superficial aspects of the design.

These insights form the basis of the VeriSIM (**Veri**fying designs by **SIM**ulating scenarios) pedagogy. VeriSIM trains students to identify and model scenarios in the design. The VeriSIM pedagogy comprises two strategies - the design tracing and the scenario branching strategy. In the design tracing strategy, students construct a model of the scenario, which is similar to a state diagram. They trace the control and data flow of a scenario while constructing the state diagram. In the scenario branching strategy, students identify different scenarios from the requirements by constructing a scenario tree. Traversing the scenario tree enables them to identify scenarios which do not satisfy the requirements.

The VeriSIM pedagogy has been operationalized into a technology-enhanced learning environment having two modules. In Module 1, students go through design tracing activities in the VeriSIM learning environment. In Module 2, students go through the scenario branching activity using a mapping tool, which is facilitated by an instructor. We believe that both the broad exploration of the design by identifying scenarios, and a deep understanding of each scenario by simulating the data and event flow for that scenario, can lead to effective evaluation of a given software design.

We have used design based research (DBR) as our overarching research framework. DBR is the systematic study of designing, developing and evaluating educational interventions, and involves iterative cycles of problem analysis, solution design and evaluation of these solutions. In this thesis, we conducted two cycles of DBR. In the first DBR cycle, we conducted studies with students to identify difficulties they faced as they evaluated a given software design. Based on these findings, we conceptualised the design tracing strategy and incorporated design tracing activities into the VeriSIM learning environment. We conducted a study to investigate the effects of VeriSIM in students' ability to trace scenarios and evaluate a software design. In DBR cycle 2, reflections from the findings of DBR cycle 1 resulted in refining the pedagogy to include the scenario branching strategy. We then investigated the effectiveness of the refined VeriSIM pedagogy and how features in VeriSIM contributed towards learning.

The final outcome of the DBR cycles is the design and development of the VeriSIM pedagogy and its operationalisation as the VeriSIM learning environment. We evaluated the effectiveness of VeriSIM and its pedagogical features in classroom studies across the two cycles. In these studies, we examined differences in students' pre-test and post-test responses for evaluating a given software design, conducted focus-group interviews and analysed VeriSIM interaction logs. The key findings from these studies are that (1) VeriSIM improved students' ability to trace scenarios and evaluate the design diagrams against the given requirements (2) Students perceived that the design tracing and scenario branching strategies are useful and (3) Pedagogical features in VeriSIM contributed towards student learning.

The major contributions of this thesis is the VeriSIM pedagogy, and features in the VeriSIM learning environment. The thesis also contributes towards computing education research by extending the theory of program comprehension, and characterization of expert and novices processes in software design evaluation.

Contents

List of Tables

List of Figures

List of Abbreviations

DBR	Design Based Research
TELE	Technology-enhanced Learning Environment
RQ	Research Question
DQ	Design Question
UML	Unified Modelling Language

Chapter 1

Introduction

1.1 Background and Motivation

Software solutions are ubiquitous in our lives, ranging from software used in workplaces, to the apps we use on our phone, to software controlling aircrafts and nuclear reactors. An increased reliance on these software solutions calls for an increased reliability of the software as well. In other words, software solutions must behave as they have been programmed to behave. A failure to do so can result in undesired behaviours, ranging from minor errors (a gaming app crash) to life-threatening scenarios (an airplane crash). According to a report by the National Institute of Standards and Technology (NIST) in 2002, software bugs cost the US economy a massive \$59.5 billion (Newman, 2002). In 2016, that number jumped to \$1.1 trillion (Cohane, 2017). The NIST report also states that a third of the cost in losses can be eliminated by earlier and more effective identification and removal of software defects. Another study by NASA showed that the cost to fix bugs escalates exponentially as the project progresses (Haskins et al., 2004). These findings stress the importance of rigorous and effective evaluation of software in earlier phases of the development cycle, especially in the design phase.

In the design phase of the software development cycle, designers come up with the conceptual design of a software system that satisfies the provided requirements. The conceptual design is often modelled using Unified Modelling Language (UML) diagrams (Rumbaugh et al., 2004). UML diagrams specify behaviours and scenarios of the system across various levels of abstraction. For example, the class diagram gives a structural view of the design, by providing the classes and their relationships, along with its data members and functions. The sequence diagram represents the behavioural view of the design, by describing how various objects pass messages with each other for a particular use-case. 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, which can then trickle into the code as well. Hence, it is necessary that students 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 encounter projects which require working on existing large and complex systems (Begel and Simon, 2008a). They usually spend their first several months resolving bugs and writing additional features based on new requirements (Begel and Simon, 2008b; Dagenais et al., 2010). 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 an already existing design, incorporate the required feature in the design, and evaluate if the design satisfies the intended goals.

Evaluating a software design is an important practice of expert software designers as well. They spend significant time evaluating their solution (Mc Neill et al., 1998). Experts create rich mental models of the design, on which they routinely perform mental simulations (Adelson and Soloway, 1986). They also use various reasoning techniques such as simulating scenarios in the given problem (Tang et al., 2010; Guindon, 1990), constraints consideration and trade-off analysis (Guindon, 1990), while reasoning about the design.

Considering the importance given to software design evaluation in the software industry, we can conclude that analysing and evaluating a design is an essential skill which needs to be incorporated in the curriculum. Design evaluation has been mentioned as a curriculum unit in the ACM syllabus for software engineering (ACM, 2014). However, sufficient emphasis has not been given on teaching and learning of evaluation techniques and practices in a software design course, and hence graduating students find it difficult to critically analyse an existing design and improve upon it (Brechner, 2003). Learning to evaluate a software design is also non-trivial. It involves going beyond understanding software design concepts, and requires developing certain cognitive processes in students, such as creating an accurate mental model of the design, simulating various scenarios in the design, and analysing how the design is satisfying/not satisfying the given requirements.

In this thesis, we aim to address this gap in the current teaching and learning of software design, by developing pedagogical strategies that enable students to evaluate a given software design.

1.2 Research Objective

The broad research objective of the thesis is to

"Design and develop a technology-enhanced learning environment (TELE) which enables students to effectively evaluate a software design against the given requirements"

The term *"software design"* has different meanings in different contexts. It can refer to a high-level abstraction of the system, with their components, relations and interactions. Details of various classes, methods or attributes are not mentioned in this context. At the next level, the design can be depicted as a set of Unified Modelling Language (UML) diagrams, which specify different views of the system, such as the structural and behavioural views. There is a close correspondence between the UML diagrams and the implementation in this level. The classes and methods are intended to be mapped to the code in the implementation stage. At the other end of the spectrum, the design can refer to the code-level implementation of the system itself. In this thesis, we refer to software design as the set of UML diagrams which model the requirements of the system.

What does it mean to *"effectively evaluate a software design against the given requirements?"* Evaluating a software design involves assessing the quality of the design from different perspectives. For example, one way to look at evaluation is to check if the syntax of all the design diagrams are correct. Another way is to examine if the non-functional requirements in the design such as modularity, extensibility etc. are met. In this thesis, software design evaluation refers to checking the semantic quality of a given design. Semantic quality refers to how faithfully the modelled system is represented i.e. if there is an accurate mapping between the design and the requirements. Issues in semantic quality occur when certain aspects of the requirements are not conveyed in the design diagrams, leading to semantic defects. For example, consider the case of an ATM system design. A requirement for this design is - *"When the user enters the ATM and inputs the correct PIN, the user can withdraw money from his/her account. If the balance is less than Rs.1000, withdrawal is denied"*. Semantic defects can occur when the ATM design does not correctly model the minimum balance requirement. For example, relevant variables which refer to the balance may be absent in the class diagram, or the sequence diagram does not call member functions from appropriate objects to check the balance before withdrawal. Detecting such semantic deficiencies is non-trivial, as it requires students to critically analyse the design and requirements, understand the relationship between different diagrams and evaluate the design against the given requirements.

In this thesis, we have developed the VeriSIM pedagogy to scaffold students to effectively evaluate a software design against the given requirements. We have instantiated the VeriSIM pedagogy as a technology-enhanced learning environment, which is freely available online, and can be used by instructors and students in their courses.

1.3 Solution Overview

During the software design process, experts create an accurate and rich mental model of the design. What is the nature of these mental models and what knowledge does it contain? Our findings from literature and novice studies show that the mental model contains information about the **diagram surface elements**, **main goals**, and **dynamic behaviours**. Diagram surface elements refer to the basic functions, data members, and messages in the design diagrams. Main goals refer to the purpose of each diagram and its role in the design. Dynamic behaviours refer to the **control flow** (such as what methods are called, which components call which functions, how a function is reached, etc.) and the **data flow** (such as how data flows through a program, change in data variable values on the basis of method calls etc.). During software design evaluation, experts analyse the requirements and map it to the corresponding design. They focus on the diagram surface elements and the main goals to gain a broad understanding of the design. They then might imagine how the system carries out specific scenarios to fulfil the stated requirements. They model each scenario by simulating the control and data flow of the scenario based on the design. In other words, they imagine all changes that occur in different diagrams for a given scenario. By identifying and modelling various scenarios in the design, they examine which scenarios and conditions violate the given requirement. This leads to effective software design evaluation.

For example, consider how experts evaluate the minimum balance requirement stated in Section 1.2. They focus on the relevant design diagrams (e.g. class diagram, sequence diagram for withdrawal), and understand the main goals of these diagrams. They analyse diagram elements such as functions (e.g. *withdraw()*) and data members (e.g. *withdrawal_amount*) relevant to the requirement. They then imagine scenarios such as - *'User entering incorrect PIN', 'User having balance greater than Rs.1000, 'User withdrawing amount greater than balance'*. For each scenario, they then visualize how the system realises these scenarios by simulating the control and data flow for each scenario in the design. For example, for the following scenario - *'User withdrawing amount greater than balance'*, one might visualize the execution of the *withdraw()* method returning 'false', since the value of the withdrawal amount variable is greater than the minimum balance variable.

However, in studies with students, we found that they focus on superficial aspects of the design, and have difficulty in simulating the control flow and data flow of various scenarios in the design. Thus, the key argument which we make in this thesis, which also forms the basis of our solution, is that -

"Scaffolding students to identify and model different scenarios in the design enables them to effectively evaluate the design against the given requirements."

1.3.1 The VeriSIM pedagogy for teaching-learning of evaluating design diagrams

We have conceptualised the VeriSIM (**Veri**fying designs by **SIM**ulating scenarios) pedagogy to scaffold students to identify and model different scenarios in the design (see Figure 1.3.) The VeriSIM pedagogy trains students to model a given scenario in the design using the **the design tracing strategy**, and trains them to identify various scenarios using the **the scenario branching strategy**.

Modelling scenarios using the design tracing strategy

Design tracing is an adaptation of the tracing strategy used in programming. Program tracing is the process of emulating how a computer executes a program (Fitzgerald et al., 2005). While tracing, programmers visualize how control flows and data values change during the execution of a program. In design tracing, students trace the control flow and data flow across different diagrams for a given scenario of system execution. They construct a model of the scenario execution, which is similar to a state diagram. The values in the states correspond to the values of relevant variables, and the transitions correspond to different parts of the scenario.

For example, in the case of the minimum balance requirement mentioned earlier, a probable scenario can be that the *'User withdraws an amount greater than balance'*. Using design tracing, students identify different data attributes from the class diagram, which are added to the states, as shown in Figure 1.1. They identify the main goals from the sequence diagram, and these form the transitions of the state diagram. Based on the transitions, the values in the states keep changing. For example, in Figure 1.1, the final transition in the state diagram results in a change in the value of *withdrawal_amount*. The final state corresponds to when the user enters the withdrawal amount greater than the balance. As students construct the state diagram, they simulate the execution of the given scenario, and also simulate the change in control flow and data flow. Hence, constructing such state diagram models for other scenarios in the design scaffolds learners to simulate dynamic behaviours of these scenarios in the design.

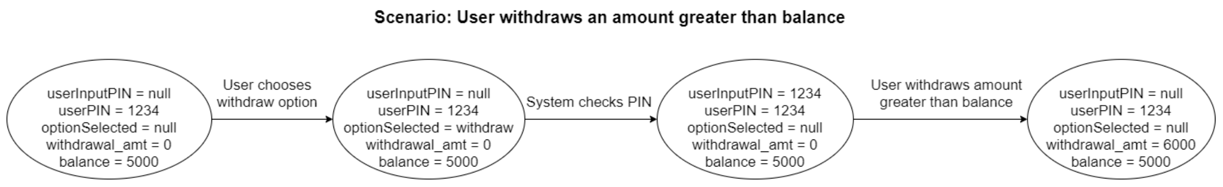


Figure 1.1: The model state diagram for a given scenario generated using the design tracing strategy

Identifying scenarios using the scenario branching strategy

Once students are trained to construct models of an already given scenario, the scenario branching strategy scaffolds them to identify different scenarios for each requirement in the design. Scenario branching is adapted from cognitive mapping strategies which have been widely used in requirement analysis (Montazemi and Conrath, 1986). In scenario branching, students analyse each requirement, and break it down into units called subgoals. For each subgoal, they identify relevant variables and different possibilities for these variables. They represent these different possibilities for each subgoal in a visual tree-like representation, known as a 'scenario branching tree'. The tree captures the identified subgoals and different values for each of these variables. After constructing the tree for a given requirement, students identify scenarios by traversing from the root to a leaf in the tree. An example of a scenario branching tree is shown in Figure 1.2.

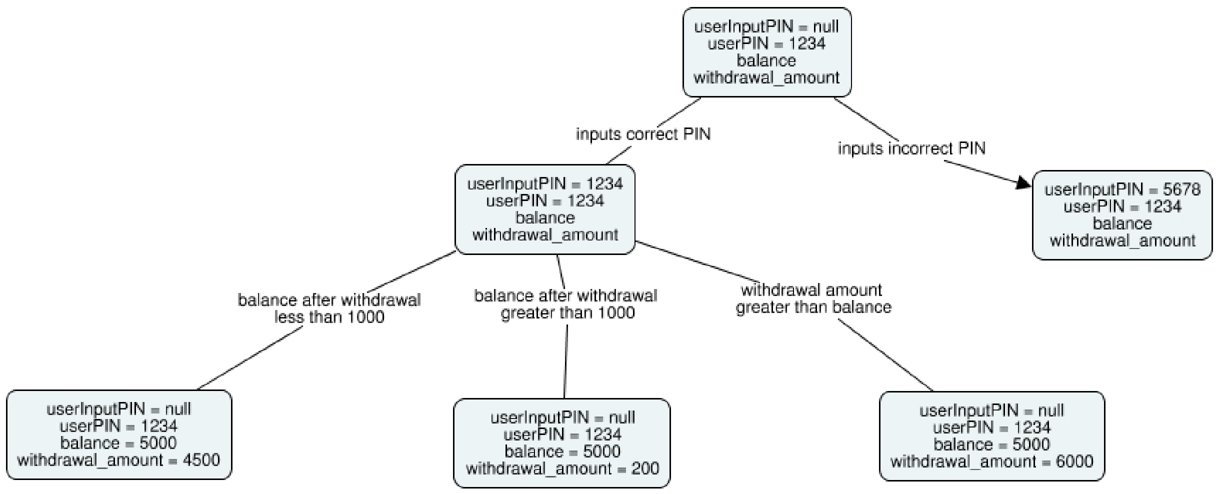


Figure 1.2: The scenario branching tree for a given requirement

Consider the minimum balance requirement - *"When the user enters the ATM and inputs the correct PIN, the user can withdraw money from his/her account. If the balance is less than Rs.1000, withdrawal is denied"*. The subgoals for this requirement are - *'User inputs correct*

PIN', 'Withdrawal of amount from account'. Based on these subgoals, students are scaffolded to construct the scenario branching tree. as shown in Figure 1.2. The scenarios which can be identified by traversing the tree are - '*1. User inputs correct PIN, and balance after withdrawal is less than 1000, 2. User inputs correct PIN, and balance after withdrawal is greater than 1000, 3. User inputs correct PIN, and withdrawal amount is greater than balance, 4. User inputs the incorrect PIN"* . Students can then examine whether these scenarios are present in the design. Hence, the scenario branching strategy enables students to identify scenarios which do not satisfy the given requirements.

The design tracing and scenario branching strategies help students develop a rich mental model of the design by promoting both a broad as well as deep exploration of the design (see Figure 1.3). Scenario branching helps students generate alternate scenarios for the given requirements. This helps them broadly explore the design space. In design tracing, students trace the execution of each scenario and examine the corresponding changes which occur in the design diagrams. This enables students to develop a deep and integrated understanding of each scenario in the design. We argue that both the broad and deep exploration of the design space, using scenario branching and design tracing, helps students develop a rich and accurate model of the design, thereby enabling them to effectively evaluate the design diagrams against the given requirements.

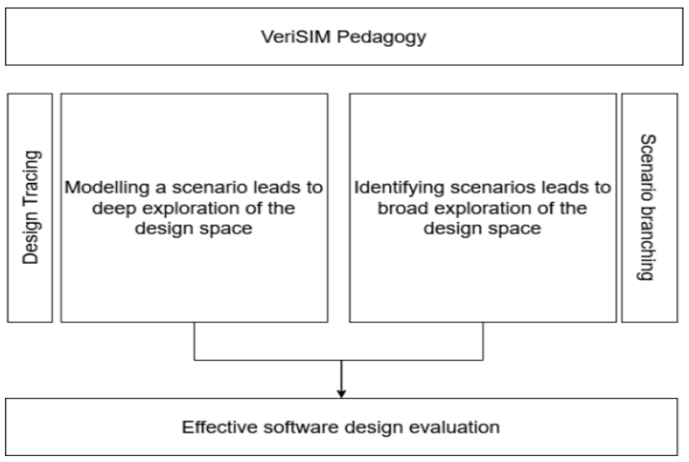
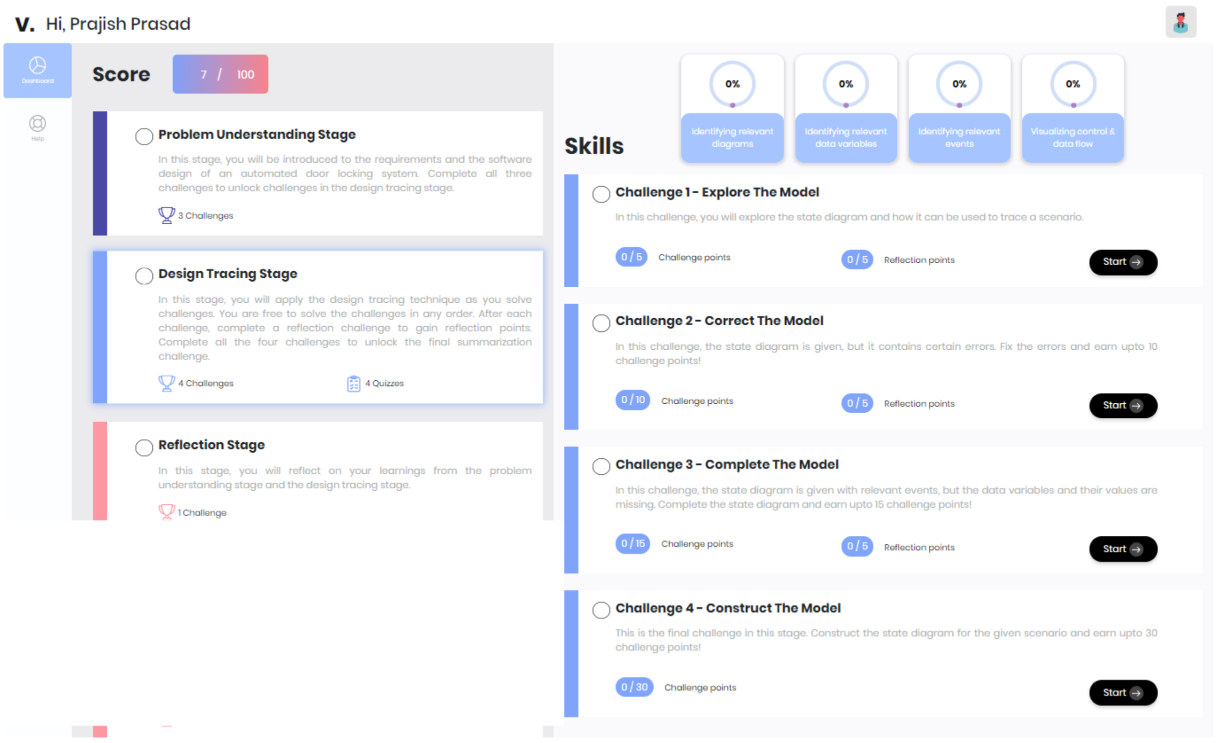

VeriSIM Pedagogy
Design Tracing
Modelling a scenario leads to deep exploration of the design space
Identifying scenarios leads to broad exploration of the design space
Scenario branching
Effective software design evaluation

Figure 1.3: The VeriSIM pedagogy

1.3.2 Operationalising the VeriSIM pedagogy into a learning environment

We have incorporated the design tracing and scenario branching strategies as two modules into a learning environment - VeriSIM. In the first module, students apply the design tracing strategy to trace the given scenarios in the design. They are introduced to the design tracing strategy and go through activities which enable them to trace given scenarios in the design (Figure 1.4 shows a screenshot of the design tracing activities in VeriSIM). The first module of VeriSIM is a self-learning module. All information and scaffolds required for students to perform design tracing are present in the learning environment itself.



Hi, Prajish Prasad

Score 7 / 100

Problem Understanding Stage

In this stage, you will be introduced to the requirements and the software design of an automated door locking system. Complete all three challenges to unlock challenges in the design tracing stage.

3 Challenges

Design Tracing Stage

In this stage, you will apply the design tracing technique as you solve challenges. You are free to solve the challenges in any order. After each challenge, complete a reflection challenge to gain reflection points. Complete all the four challenges to unlock the final summarization challenge.

4 Challenges 4 Quizzes

Reflection Stage

In this stage, you will reflect on your learnings from the problem understanding stage and the design tracing stage.

1 Challenge

Skills

Identifying relevant diagrams	Identifying relevant data variables	Identifying relevant events	Visualizing control & data flow
0%	0%	0%	0%

Challenge 1 - Explore The Model

In this challenge, you will explore the state diagram and how it can be used to trace a scenario.

0 / 5 Challenge points 0 / 5 Reflection points

Challenge 2 - Correct The Model

In this challenge, the state diagram is given, but it contains certain errors. Fix the errors and earn upto 10 challenge points!

0 / 10 Challenge points 0 / 5 Reflection points

Challenge 3 - Complete The Model

In this challenge, the state diagram is given with relevant events, but the data variables and their values are missing. Complete the state diagram and earn upto 15 challenge points!

0 / 15 Challenge points 0 / 5 Reflection points

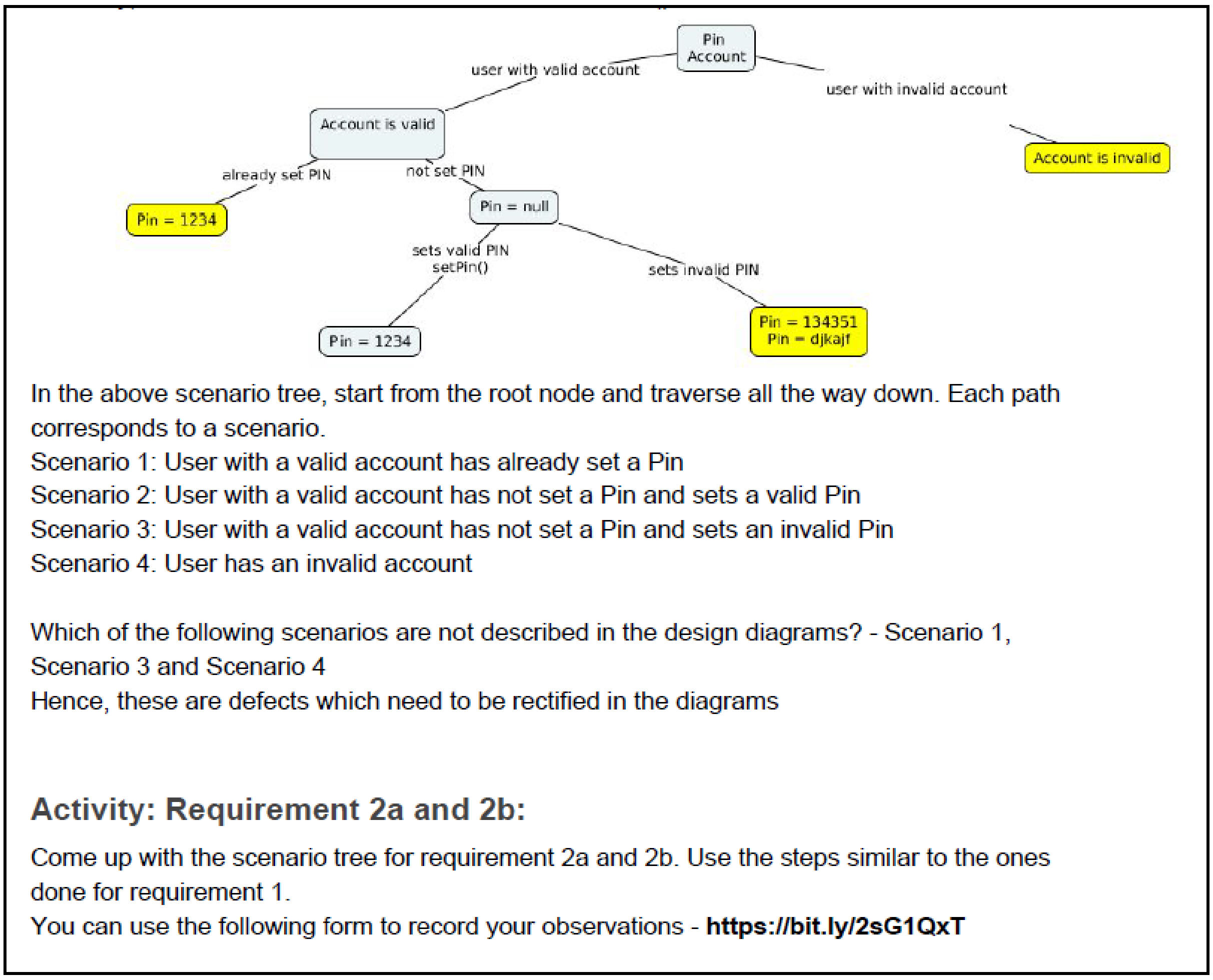
Challenge 4 - Construct The Model

This is the final challenge in this stage. Construct the state diagram for the given scenario and earn upto 30 challenge points!

0 / 30 Challenge points

Figure 1.4: Screenshot of the activities in the Design Tracing stage of VeriSIM

In the second module, learners are introduced to the scenario branching strategy. Learners generate alternate scenarios in a given design using a mapping (CMAP) tool. This module is partly guided by the instructor. Students are provided with a worksheet which contains the requirements and design diagrams for a given problem (Figure 1.5). The instructor explains the scenario branching pedagogy and facilitates alternate scenarios creation using the CMAP tool.


Pin Account
user with valid account
user with invalid account
Account is valid
Account is invalid
already set PIN
not set PIN
Pin = 1234
Pin = null
sets valid PIN setPin()
sets invalid PIN
Pin = 1234
Pin = 134351 Pin = djkajf

In the above scenario tree, start from the root node and traverse all the way down. Each path corresponds to a scenario.
Scenario 1: User with a valid account has already set a Pin
Scenario 2: User with a valid account has not set a Pin and sets a valid Pin
Scenario 3: User with a valid account has not set a Pin and sets an invalid Pin
Scenario 4: User has an invalid account

Which of the following scenarios are not described in the design diagrams? - Scenario 1, Scenario 3 and Scenario 4
Hence, these are defects which need to be rectified in the diagrams

Activity: Requirement 2a and 2b:

Come up with the scenario tree for requirement 2a and 2b. Use the steps similar to the ones done for requirement 1.
You can use the following form to record your observations - **https://bit.ly/2sG1QxT**

Figure 1.5: Snapshot of the scenario branching worksheet

1.4 Research Methodology

We have used the design-based research (DBR) methodology to answer our broad research objective. According to Barab et al. (Barab and Squire, 2004), DBR is *"a series of approaches, with the intent of producing new theories, artifacts and practices that account for and potentially impact learning and teaching in naturalistic settings."* DBR provides a framework to study learning phenomena in the real world rather than the laboratory and goes beyond narrow measures of learning. An important feature of the DBR process is the emphasis which it places on advancing theories of learning. According to Barab et al. (Barab and Squire, 2004), "DBR is more than just showing that a particular design works but demands that the researcher moves beyond a particular design exemplar to generate evidence-based claims about learning that address contemporary theoretical issues and further the theoretical knowledge of the field". Hence DBR does not just aim at designing, developing and evaluating pedagogical solutions, but it also aims at identifying underlying design principles or local learning theories (Cobb et al., 2003). Another feature of DBR is that it follows an iterative design process, where an intervention is progressively refined and new conjectures are developed and evaluated. This results in an iterative design process featuring cycles of invention and revision (Cobb et al., 2003).

We have chosen DBR primarily for the above mentioned reasons. The focus of the thesis is not only to evaluate the effectiveness of the TELE, but also understand how students evaluate software design diagrams using the TELE. We wish to uncover how pedagogical features in the TELE are contributing to the development of their evaluation skills. These design principles and local learning theories can contribute to existing teaching-learning theories in software design and programming.

1.4.1 DBR Cycles in this Thesis

A cycle of DBR mainly involves 3 phases - Problem Analysis and Exploration, Solution Design and Development, and Evaluation and Reflection. Problems pertaining to a particular context are identified, potential gaps are explored and based on these explorations, a solution is designed. The solution is evaluated and refined using quantitative as well as qualitative methods. Reflections from these evaluations feed into the next cycle of DBR.

In this thesis, we conducted two cycles of DBR. A summary of these cycles is shown in Figure 1.6. In the first DBR Cycle, review of literature and studies with novices resulted in the VeriSIM pedagogy, and its operationalisation into the VeriSIM TELE. The VeriSIM TELE scaffolds students to apply the 'design tracing' strategy to trace scenarios. We conducted a study with students in order to measure the effectiveness of the VeriSIM TELE.

In the second DBR Cycle, we reflected on the study findings from the first cycle. This led to the refinement of the VeriSIM pedagogy. We added a second module to the VeriSIM TELE which included the 'scenario branching' strategy to help students identify scenarios in the design. In this cycle, we also investigated the effects of various pedagogical features on student learning, and derived a local learning theory, showing how VeriSIM enables students to develop an effective mental model of the design.

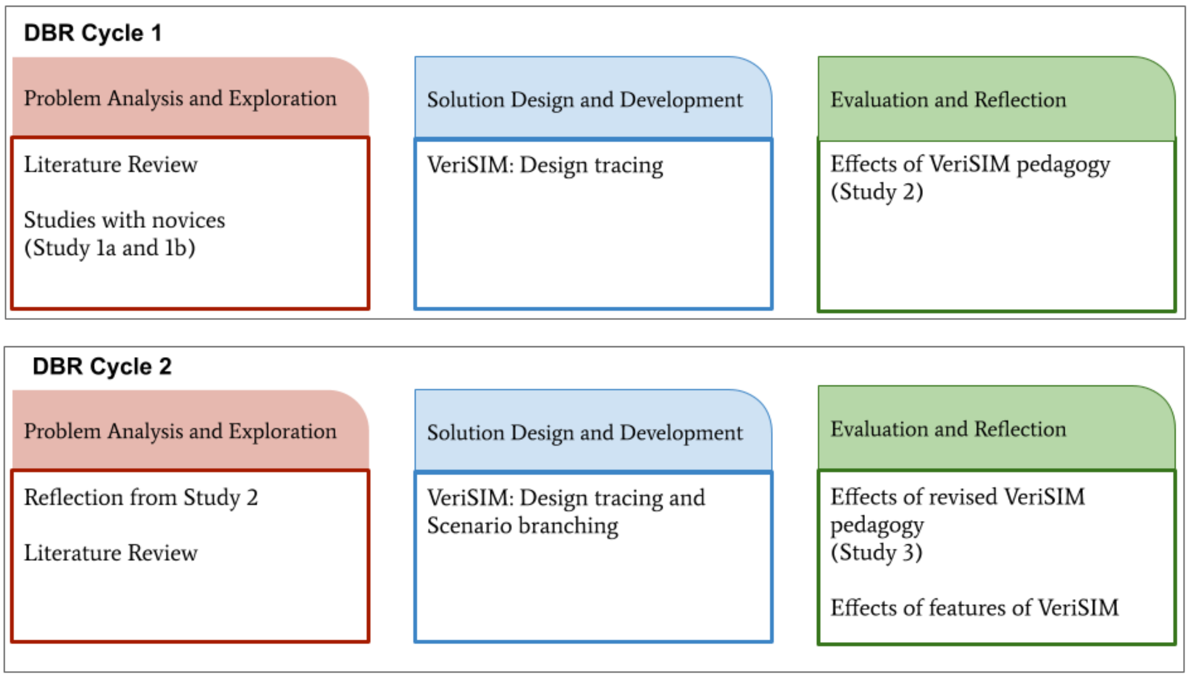

DBR Cycle 1
Problem Analysis and Exploration
Literature Review
Studies with novices (Study 1a and 1b)
Solution Design and Development
VeriSIM: Design tracing
Evaluation and Reflection
Effects of VeriSIM pedagogy (Study 2)
DBR Cycle 2
Problem Analysis and Exploration
Reflection from Study 2
Literature Review
Solution Design and Development
VeriSIM: Design tracing and Scenario branching
Evaluation and Reflection
Effects of revised VeriSIM pedagogy (Study 3)
Effects of features of VeriSIM

Figure 1.6: Overview of the DBR cycles in the thesis

1.4.2 Research Studies in this Thesis

We conducted four research studies in this thesis - three research studies in Cycle 1, and one study in Cycle 2. Table 1.1 summarises the goals of each research study. The first two studies aimed at identifying student processes and their difficulties while evaluating designs. The next two studies involved investigating the effects of the VeriSIM TELE in students' ability to trace scenarios and evaluate the design diagrams against the requirements.

Study 1a: In Study 1a, we investigated the categories of written responses of 100 students when asked to evaluate a given design against the requirements. This gave us indicators about what students were thinking as they evaluated a given design. (More details about Study 1a can be found in Section 4.2)

The research question guiding this study was - RQ 1.1: How do students evaluate a software design against the given requirements?

Study 1b: In order to delve deeper into the findings from Study 1a, we conducted a qualitative study with 6 students. The aim of this study was to analyse the strategies which students use, and investigate aspects of their mental models while they evaluated a given design. (More details about Study 1b can be found in Section 4.3)

The research questions guiding Study 1b were as follows:

- RQ 1.2: What defects are students able to identify in the design evaluation task?
- RQ 1.3: What reading strategies do students use to evaluate software design diagrams against the given requirements?
- RQ 1.4: What are the elements in their mental model?

The key findings from Study 1a and 1b is that students focus only on superficial aspects of a design. Their mental models do not contain information about the control flow and data flow of the design. They also focus on adding new functionalities instead of evaluating the design against the given requirements. These findings informed the design and development of the VeriSIM learning environment.

Study 2 and 3: In Study 2 and 3, we investigated the effects of the VeriSIM TELE in students' ability to trace scenarios and evaluate the design diagrams against the requirements. In Study 2, students interacted with the first module of the VeriSIM TELE, i.e. the design tracing strategy. (More details about Study 2 can be found in Section 6.1)

The research questions guiding Study 2 were as follows:

- RQ 2.1: What are the effects of VeriSIM in students' ability to simulate dynamic behaviours in the design?
- RQ 2.2: What are the effects of VeriSIM in students' ability to identify defects in the design?

In Study 3, students went through both modules of the VeriSIM TELE, i.e. the design tracing and scenario branching strategy (More details about Study 3 can be found in Section 7.6)

The research questions guiding Study 3 were as follows:

- RQ 3.1: What are the effects of VeriSIM in students' ability to identify scenarios in the design?
- RQ 3.2: What are the effects of VeriSIM in students' ability to uncover defects in the design?

In both Study 2 and 3, we also investigated RQ 4: How are pedagogical features in the VeriSIM learning environment contributing towards student learning? (More details about how we answered this RQ can be found in Chapter 8).

Findings from Study 2 and 3 show that VeriSIM enabled students to simulate dynamic behaviours in the design. They also shifted from adding or modifying features in the design, to identifying scenarios which do not satisfy the requirements. These findings also show that the pedagogical features in VeriSIM enabled students to build effective mental models of the design.

Table 1.1: Summary of research studies in the thesis

Research Study	**Goal**
Study 1a	Identifying categories of students responses when asked to evaluate a given design
Study 1b	1. Determine student mental models 2. Identify student difficulties
Study 2	Effects of VeriSIM TELE (design tracing)
Study 3	Effects of VeriSIM TELE (design tracing + scenario branching)

1.5 Scope of the Thesis

As mentioned earlier, the scope of software design evaluation is enabling students to detect semantic deficiencies in the design. The VeriSIM TELE is intended to be used by engineering students who have undergone a basic course in software design. They should be familiar with UML diagrams like class and sequence diagrams, and ideally should have created simple designs using UML diagrams. This familiarity is important, because learners need to first comprehend a given design before they can evaluate it.

In this thesis, the design diagrams which students evaluate are scoped to class and sequence diagrams. The class diagram and sequence diagram correspond to the structural and behavioural categories of design diagrams respectively. These diagrams are the most commonly used design diagrams and the first diagrams which students learn about (Dobing and Parsons, 2006; Lange et al., 2006).

1.6 Contributions of the Thesis

This thesis makes the following contributions:

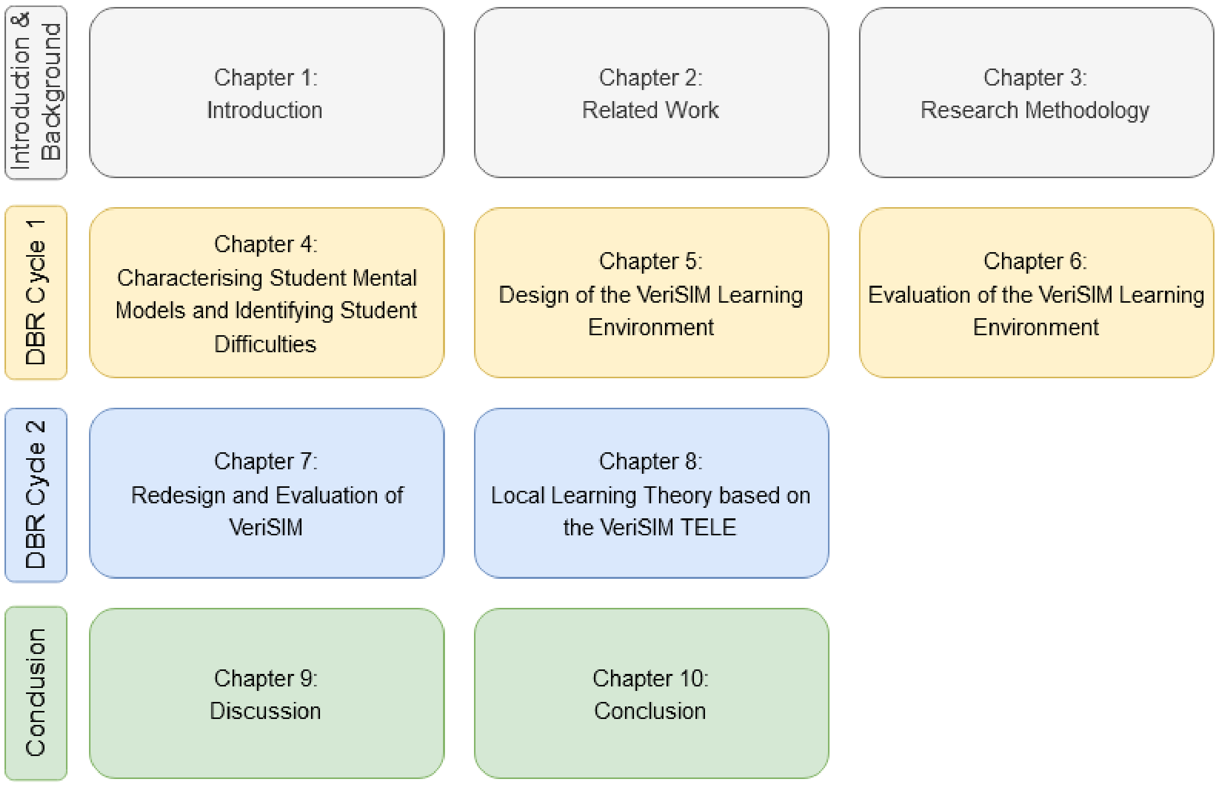
1. **Unpacking novice strategies and mental models while evaluating software designs:** Although there has been sufficient emphasis in understanding student difficulties while creating designs, there has been a lack of research on how students evaluate a given design. In this thesis, we have investigated the strategies and mental models of novices, as they evaluate a given design. Findings from our novice studies contribute towards the theory of novice mental models and strategies used in design evaluation.

2. **Pedagogy for teaching-learning of software design evaluation:** The design tracing and scenario branching strategy can be used by instructors in their software design and engineering courses. We also show in this thesis that the VeriSIM pedagogy is beneficial for students in developing an accurate model of the design, and contributes towards a local learning theory of how students evaluate a software design against the given requirements.

3. **VeriSIM Learning Environment:** The VeriSIM learning environment is freely available to use by instructors as well as students. It can be accessed at **https://verisim.tech**. Various design features in VeriSIM can also be used by learning environment designers in related contexts of software design and programming.

1.7 Structure of the Thesis

The thesis is structured as 10 chapters, as shown in Figure 1.7. In Chapter 2, we review related work in computing education and empirical software engineering in order to investigate existing teaching-learning in software design, and strategies which experts use while evaluating designs. We also review difficulties which students face in the software design process. In Chapter 3, we describe the overarching research methodology: design based research, which provides the framework of our investigations. We provide reasons for why we have chosen this methodology, and describe the research questions and analysis methods used in each DBR cycle.



Introduction & Background
Chapter 1: Introduction
Chapter 2: Related Work
Chapter 3: Research Methodology
DBR Cycle 1
Chapter 4: Characterising Student Mental Models and Identifying Student Difficulties
Chapter 5: Design of the VeriSIM Learning Environment
Chapter 6: Evaluation of the VeriSIM Learning Environment
DBR Cycle 2
Chapter 7: Redesign and Evaluation of VeriSIM
Chapter 8: Local Learning Theory based on the VeriSIM TELE
Conclusion
Chapter 9: Discussion
Chapter 10: Conclusion

Figure 1.7: Organization of the thesis

Based on the insights from our review of literature, in Chapter 4 we investigate how students evaluate a given design and the difficulties they face (Study 1a and 1b). These findings inform the design and development of our pedagogy. In Chapter 5, we describe the details of the VeriSIM pedagogy and learning environment, and the rationale behind various activities and pedagogical features in VeriSIM. In Chapter 6, we describe details of Study 2, which investigated the effects of VeriSIM in students' ability to trace scenarios and evaluate a given design.

Chapter 7 describes the redesign of VeriSIM which is based on the reflections from Study 2, where we describe the scenario branching strategy, and argue for its appropriateness. We then investigate the effects of the revised pedagogy in students' ability to evaluate designs. In Chapter 8, we show how various pedagogical features in VeriSIM are contributing towards the learning of evaluation of design diagrams. In Chapter 9, we discuss the claims, limitations and implications of the thesis. In Chapter 10, we summarize the contributions of the thesis and discuss future directions which can be informed by this thesis.

Chapter 2

Related Work

2.1 Teaching-learning of Software Design

Traditionally, teaching software design often means teaching object-oriented (OO) design, and typically entails courses such as the Unified Modelling Language (UML), design patterns and OO design principles (Topi et al., 2010; Sahami et al., 2013). These courses are expected to enable learners to establish a requirements model of an application (object-oriented analysis, OOA), successively transform those requirements into computer-based models of the application (object-oriented logical design, OOLD), and further translate the models for specific technical platforms (object-oriented physical design, OOPD). This model-centric approach, known as Model-Driven Engineering (MDE) is a prominent area in the software engineering field and has languages, standards, tools, and well-defined practices (Burgueño et al., 2018). The purpose of these models is to enable software designers to capture and precisely state all the requirements and domain knowledge in a manner that is understandable to all stakeholders. These models also enable developers design solutions easily before writing code.

2.1.1 Unified Modelling Languages (UML)

In typical Model-Driven Engineering courses, various UML diagrams are taught, where each diagram describes a particular view of the system. At the top level, views of the system can be divided into these areas - structural classification, dynamic behaviour, physical layout and model management.

- Structural Classification - These views define the structure of the software system. They describe the things in the system and their relationship to other things. Different views are the static view, design view, and use case view. A static view does not describe time-dependant behaviour of the system, but describes the logical parts of the system, including classes and relevant data and functions in these classes. The design view models the design structure of the application, collaborations that provide functionality, and assembly from components with well-defined artifacts. The use-case view models the functionality of a subject, as perceived by outside agents, called actors, that interact with the subject from a particular viewpoint.

- Dynamic behaviour - These views describe the behaviour of the system over time. Various views include the state machine view, activity view and the interaction view. The state machine view models the possible life-histories of an object of a class. The activity diagram shows the flow of control among computational activities involved in performing a calculation or workflow. The interaction view describes sequence of message exchanges among parts of a system. It gives a holistic view of behaviour in a system by showing the flow of control across many objects.

- Physical layout - These views come later in the design phase and is used to mainly model computational resources and deployment of artifacts.

- Model management - Model management describes the organization of the models themselves into hierarchical units.

While designing a software system, the structural as well as dynamic behavioural views are particularly important. In order to model the structure of a system, various diagrams like the class diagram and use case diagram are used. A class diagram describes discrete objects that hold information, such as the data attributes and functions which implement specific behaviour related to the requirements. In order to model the dynamic behaviour, diagrams like state machine, activity diagrams and sequence diagrams are used.

We explain the structural and behavioural UML diagrams with the help of an example. Consider the requirements of an ATM system -

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 structural view of the ATM system design can be modelled using a class diagram, as shown in Figure 2.1. The class diagram describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects. We see that the class diagram models relevant classes like the 'Customer', 'ATM' and 'Account'. Each of these classes contain data members and functions necessary to realise certain behaviours of the ATM system. For example, the 'Account' class has a data attribute 'balance', which records the balance of an account holder. The withdraw requirement is realised by the 'withdraw()' function in the Account class, which accordingly updates the 'balance' variable.

The behaviours corresponding to the requirements can be implemented using sequence diagrams. Figure 2.2 represents the sequence diagram which describes the behaviour of the 'withdraw' requirement. The classes from the class diagram are represented in the sequence diagram at the top, and have parallel vertical lines, called lifelines. The horizontal arrows are messages exchanged between them, in the order that they occur. These messages can be functions from the class diagram, such as 'checkATM()', 'checkPIN()' and 'withdraw()'. This sequence of messages correspond to the implementation of a given requirement or sub-requirement. In a similar fashion, sequence diagrams can be constructed to model other requirements as well.

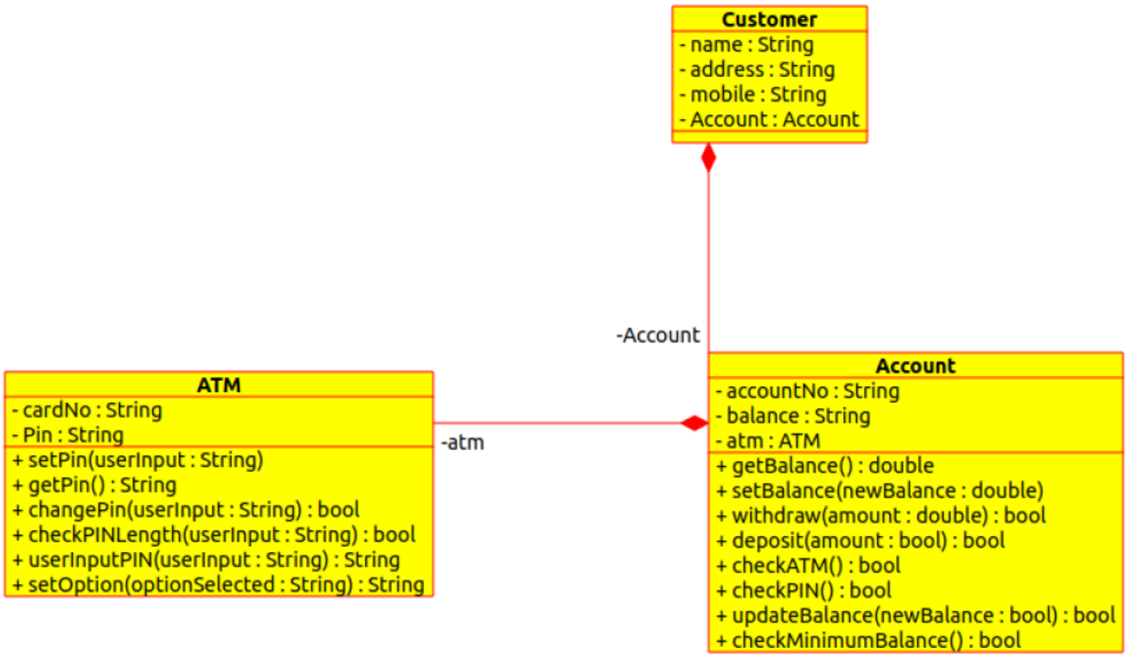

Customer
- name : String
- address : String
- mobile : String
- Account : Account
-Account
Account
- accountNo : String
- balance : String
- atm : ATM
+ getBalance() : double
+ setBalance(newBalance : double)
+ withdraw(amount : double) : bool
+ deposit(amount : bool) : bool
+ checkATM() : bool
+ checkPIN() : bool
+ updateBalance(newBalance : bool) : bool
+ checkMinimumBalance() : bool
ATM
- cardNo : String
- Pin : String
+ setPin(userInput : String)
+ getPin() : String
+ changePin(userInput : String) : bool
+ checkPINLength(userInput : String) : bool
+ userInputPIN(userInput : String) : String
+ setOption(optionSelected : String) : String
-atm

Figure 2.1: Class diagram of the ATM system

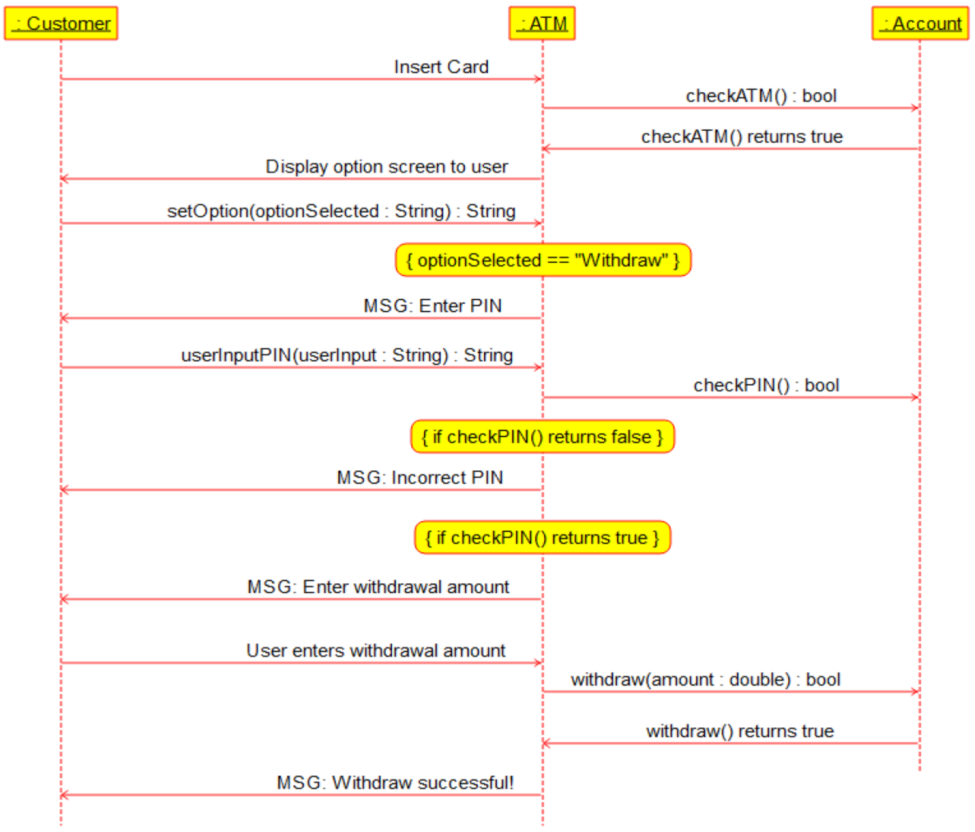

: Customer
: ATM
: Account
Insert Card
checkATM() : bool
checkATM() returns true
Display option screen to user
setOption(optionSelected : String) : String
{ optionSelected == "Withdraw" }
MSG: Enter PIN
userInputPIN(userInput : String) : String
checkPIN() : bool
{ if checkPIN() returns false }
MSG: Incorrect PIN
{ if checkPIN() returns true }
MSG: Enter withdrawal amount
User enters withdrawal amount
withdraw(amount : double) : bool
withdraw() returns true
MSG: Withdraw successful!

Figure 2.2: Sequence diagram for withdrawing amount

There are several other UML diagrams (such as state charts, communication diagrams, activity diagrams etc.) which can be used to adequately model a given software design. Details of various UML diagrams can be found in Rumbaugh et al. (1999). During the software design process, a subset of these UML diagrams are used to create the design model. Various tools have been used in the teaching and learning of software modelling, which we explain below.

2.1.2 Tools used in Software Modelling

Tools used in modelling courses are varied, which include modelling tools used in industry, specific educational tools, or simply ignoring tools altogether in favour of pencil and paper (Ciccozzi et al., 2018). Ciccozzi et al. conducted a survey with 47 instructors, asking them which tools they used to construct UML models in a software design course. The findings show that instructors use a wide variety of tools in their courses (Ciccozzi et al., 2018). The most commonly used ones were the Eclipse Modeling Framework (EMF) [1], Xtext [2], Papyrus [3] and ATL [4], Modelio [5] and Visual Paradigm [6]. Most of these tools are plugins in EMF, and are open-source. However, there have been concerns that software modelling tools are not well suited to teach modelling, they are not mature enough, or are too complex to use (Ciccozzi et al., 2018). There is also a lack of discussion about modelling tools that support software engineering education (Whittle et al., 2014).

Sufficient emphasis has also not been given to teaching-learning of software design evaluation. Although software engineering courses teach the basic syntax of diagrams in UML, important design evaluation aspects like well-formedness of models, and semantics of designs are not given sufficient importance (Westphal, 2019). For example, while evaluating software designs, apart from checking syntactic issues in the design, students need to focus on developing an integrated understanding of the UML diagrams in order to build designs which are comprehensive and consistent. If we consider the case of the ATM system design, a designer

[1]https://www.eclipse.org/modeling/emf/

[2]https://www.eclipse.org/Xtext/

[3]https://www.eclipse.org/papyrus/

[4]https://www.eclipse.org/atl/

[5]https://www.modelio.org/

[6]https://www.visual-paradigm.com/

should ensure that the class diagram and sequence diagrams adequately model all the given requirements. They should also ensure consistency among elements in the design. For example, functions called in the sequence diagram should match member functions in the class diagram. Such semantic issues can affect the quality of software designs.

Hence, we see that there are different perspectives of quality while evaluating software designs. We explain these software quality issues in detail below.

2.1.3 Software Quality of UML Models

There are three main types used to define the software quality of UML models - syntax, semantics and pragmatics (Lindland et al., 1994; Unhelkar, 2005). These types are summarized in Figure 2.3, and we describe each in detail below.

- **Syntactic quality** refers to how well the models adheres to the rules of the language i.e. the rules and syntax of UML. Hence, issues in the syntactic quality can be due to mistakes corresponding to the syntax and naming conventions in UML diagrams *(e.g.: Are notations in class and sequence diagrams adhering to the standard rules and syntax of UML).*
- **Semantic quality** refers to how faithfully the modelled system is represented i.e. if there is an accurate mapping between the model and the requirements. There are three main quality characteristics with regard to semantic quality - consistency, completeness, and correctness. Consistency ensures that there is coherence between elements in a particular UML diagram, and between different UML diagrams as well. Completeness of a model ensures that all requirements for the system being developed have been represented. Finally, correctness of a model ensures that the UML diagrams represent the system requirements adequately. Issues in semantic quality can occur when the model lacks something that is present in the requirements, or the model includes something that is not present in the requirements. For example, in the ATM system design provided above, the 'withdraw()' function in the withdraw sequence diagram does not check whether the balance is above the withdrawal limit. In the class diagram, the type of 'balance' is incorrectly represented as 'String'.

- **Pragmatic quality** refers to how well a given design can be interpreted by different stakeholders, such as other team members, clients etc. There are several characteristics with regards to pragmatic quality, such as maintainability (how easy is the system to maintain), reusability (how can components of the design be used in building the model of another system), complexity (how complex the system is to understand) etc. (Nelson and Piattini, 2012).

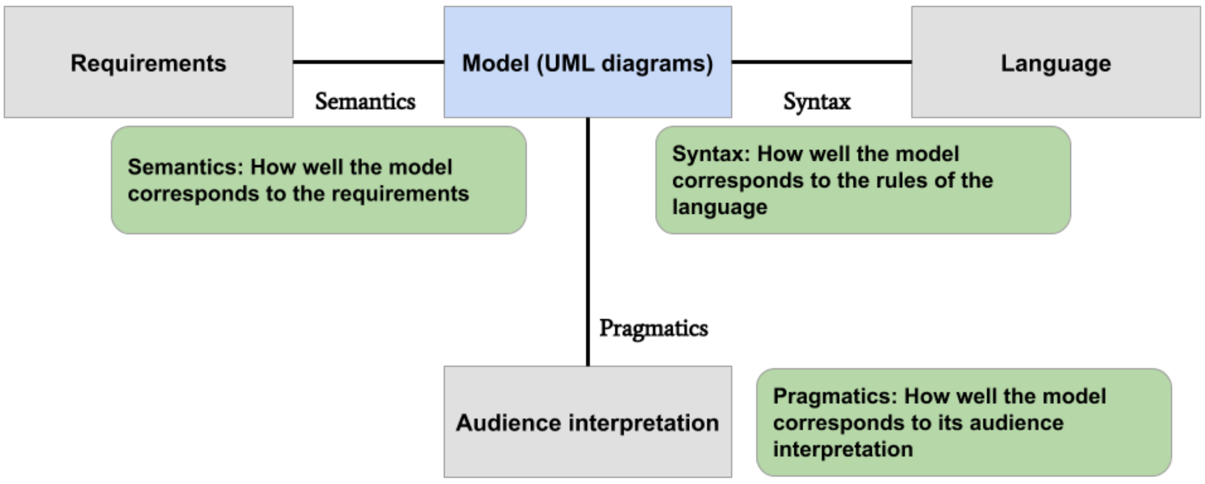

Requirements
Model (UML diagrams)
Language
Semantics
Syntax
Semantics: How well the model corresponds to the requirements
Syntax: How well the model corresponds to the rules of the language
Pragmatics
Audience interpretation
Pragmatics: How well the model corresponds to its audience interpretation

Figure 2.3: Types of software design quality

To summarize this section, we see that UML design diagrams are a prominent paradigm for designing software systems. Although there are several tools that focus on modelling software designs, sufficient emphasis has not been given on effective pedagogies for helping students evaluate a given software design. In the next section, we describe common difficulties which students face in the software design process.

2.2 Student Difficulties in Design

Studies have shown that a majority of graduating students are not competent in designing software systems (Eckerdal et al., 2006; Loftus et al., 2011). In a multi-national, multi-institutional study conducted by Eckerdal et al. (2006), the authors analysed 149 graduating students' software designs. Based on the analysis of the designs, they conclude that "the majority of graduating students cannot design a software system". The analysis shows that around 60% of the designs were either simple restatements of the specification, or just added an insignificant amount beyond the specification. Only 9% produced partial or complete designs. Loftus et al. conducted a similar study, allowing students to design software in groups, yielding similar results (Loftus et al., 2011).

There are several studies conducted in order to understand what type of difficulties students face as they create software designs. Based on the analysis of literature, we have categorized these difficulties as (1) insufficient understanding of domain and specifications (2) inconsistency issues due to lack of an integrated understanding of the design diagrams and (3) missing information between design diagrams.

When students are asked to design a software system, it is essential that they initially develop an understanding of the problem domain and its specifications (Sonnentag, 1998). That is, students are required to understand the concepts and entities in the problem for which they have to create the design, so that they can translate these requirements to a concrete design. However, students find it difficult to abstract real world problems and build models from the problem domain because they do not know "what" to model (Sien, 2011). In a study conducted by Chren et al., the authors analysed over 2,700 diagrams from 123 students to form a catalogue, consisting of 146 types of mistakes in eight types of diagrams (Chren et al., 2019). Over 65% of all mistakes present in the UML models were attributed to the insufficient understanding of the system's domain or specification. Thomasson et al. also came up with similar results, where students know that they need a class to model a concept but could not figure out how to integrate the class into their designs (Thomasson et al., 2006).

Students also face difficulties in developing a consistent understanding among different diagrams (Sien, 2011; Stikkolorum and Chaudron, 2016). While modelling designs, students are required to create models with multiple views, such as the structural view (e.g. class diagrams), and the behavioural view (e.g. sequence diagrams). Students face difficulties in understanding how the overall system specifications actually work based on these views (Burgueño et al., 2018). They struggle to understand the purpose and relationship among these diagrams, and tend to view the diagrams as existing in isolation (Stikkolorum and Chaudron, 2016). This lack of understanding leads to consistency issues as well. For example, in a study conducted by Loftus et al. (2011), students were asked to design a software system based on the given requirements. The authors found that although students constructed use-case diagrams, they did not link these use-case diagrams to other diagrams like the sequence diagram. In another study conducted by Chren et al. (2019), it was found that messages in the sequence diagram were not consistent with the class diagram. That is, students used wrong or made-up methods in the sequence diagram which were not present in the class diagram. In another study by Thomasson et al. (2006), students created classes in isolation without linking them to other classes.

Due to consistency issues, students also failed to add relevant information in design diagrams. For example, class diagrams would have missing class operations or dependencies (Chren et al., 2019). Students would also create sequence diagrams with missing messages, parameters and objects (Bolloju and Leung, 2006).

To summarize, the context of the studies mentioned in this section have been on difficulties students face while creating designs. We saw that students' difficulties are primarily based on an inadequate knowledge of the domain, and inconsistent understanding between various design diagrams. Moreover, studies have not examined how students evaluate an already given software design and the difficulties they face. We believe *that adequate knowledge of the problem domain and developing a consistent understanding of the design is essential in the case of software design evaluation as well.* This is evident from how experts evaluate a design, wherein they build effective mental models of the problem domain and the design. We explain this in detail in the next sections where we describe expert practices in software design and strategies which they use while evaluating designs.

2.3 Expert Practices in Software Design

Expertise in design has been studied in several fields of engineering, like electronic design (Mc Neill et al., 1998) and aerospace design (Ahmed et al., 2003), and it has been shown that evaluation is an important practice of experts. Mc Neill et al. conducted a study with expert electronic designers, and found that throughout the design episode, experts spend time in multiple cycles of analysis, synthesis and evaluation (Mc Neill et al., 1998). In another study, Ahmed et al. conducted a study with expert and novice aerospace designers. The study showed that experts performed a sound, preliminary evaluation of their tentative design decisions before implementation, as opposed to novices who used trial-and-error techniques to evaluate their design through several iterations (Ahmed et al., 2003).

We see a similar emphasis to evaluation in expert software designers as well. When experts are asked to create a design, they start with an initial model of the design, evaluate it against the requirements, and further expand the model until the design is complete (Adelson and Soloway, 1986). For example, Adelson and Soloway (1986) conducted a study with three expert designers who were asked to design an electronic mail system and incorporate features such as 'read', 'reply', 'send', 'delete', 'save', and 'edit'. Findings from this study showed that designers began with a skeletal version of the mail system, known as the 'sketchy model', and gradually made it more concrete as the design progressed. They divided the design into various modules, which were defined at approximately the same level of detail at a particular point in time. During the design process, they worked with the abstract state of the design and used their existing knowledge of an electronic mail system (domain knowledge), to successively transform the abstract state to a more concrete state based on the final goal. After each refinement, they evaluated the difference between the behaviours of the current and the final goal state. These successive refinements of the initial abstract model led them to finally develop a design which satisfied the given requirements.

We can apply this process to the example of the ATM system described in Section 2.1 as well. When experts are asked to create a design for the ATM system, they typically analyse the requirements, apply knowledge of the domain and start with the initial design. They define the classes, and the initial model can contain a preliminary class diagram, without the data and member functions. Based on each requirement, they then generally use appropriate objects and design preliminary sequence diagrams which describe the behaviour based on the given requirements. In subsequent iterations, they can refine the initial model, by adding elements such as data variables, member functions, and relationships between classes, and refine the sequence diagrams to include these elements as well. Based on the refinements, they evaluate the intermediate model with the given requirements, and continue refinements till they build the entire design which satisfies the intended requirements.

How do experts produce these initial and refined models of the design? They devise these models by creating and manipulating abstract structures in their mind. These abstract mental structures are referred to as “mental models”.

Mental models and mental simulation

A mental model is a “mental structure that represents some aspect of one’s environment” (Sorva, 2013). It is an abstract representation of a system that enables us to describe the underlying mechanisms of systems, answer questions about the system, as well as predict future system states (Schumacher and Czerwinski, 1992). As experts create software designs, they create mental models of the initial design. They use these mental models to reason with the system, and incrementally transform these models based on their reasoning to make the design more concrete.

But what do their mental models consist of? Mental model elements have been investigated in the context of how programmers comprehend code, and have resulted in several program comprehension frameworks (Soloway and Ehrlich, 1984; Pennington, 1987; Von Mayrhauser and Vans, 1996; Letovsky, 1987; Wiedenbeck, 1986). According to these frameworks, programmers’ mental models consist of cognitive structures such as **goals** (Pennington, 1987), **plans** (Soloway and Ehrlich, 1984), **control flow and data flow** (Pennington, 1987; Burkhardt et al., 2002), **program models** and **situation models** (Von Mayrhauser and Vans, 1996).

Goals refer to the purpose and role of a particular program, or parts of the program. For example, in the electronic mail system described earlier, goals of the system can refer to specific purposes and functions present in the system, such as a function to ‘read’ and ‘send’ a mail. **Plans** refer to a bag of tricks or solutions to problems that were solved in the past. For example, to sort all mails in the mail system, programmers can implement a sorting algorithm, which is a plan in their mental model. **Control flow and data flow** refers to the sequence of changes that occur during program execution, such as the flow of control in a program, and change in values of variables on account of program execution. Programmers simulate the control flow and data flow to make sense of the program execution (LaToza and Myers, 2010). For example, for the 'send' function in the electronic mail system, programmers simulate the execution of their code and examine if the control flow and data flow is leading to correct behaviours. In addition to these elements, programmers also require knowledge about the programming language (**program model**), as well as knowledge about the problem domain (**situation model**).

In most frameworks which describe the mental model elements, the dominant external representation has been code. However, an adaptation of these frameworks to design diagrams as the external artifact has not been explored. We delve deeper into this in Chapter 4, where we provide an adaptation for the elements of the mental model for design diagrams.

Mental models are capable of supporting mental simulations (Adelson and Soloway, 1985). That is, mental models can be used to reason about systems in certain situations, and imagine with the mind how the system will work based on a given set of conditions (Gentner and Gentner, 1983). When asked to create a design from specifications, experts repeatedly conducted mental simulation runs of their partially complete designs, and at different levels of abstraction (Adelson and Soloway, 1986). Visser et al. suggest that simulations can serve in problem comprehension i.e. when the designer explores and simulates the problem environment in order to understand the problem domain, as well as in evaluation i.e. the designer runs simulations of tentative solutions, and chooses the effective one based on certain criteria (Visser and Hoc, 1990).

Experts also employ several reasoning techniques in the process of designing software. Marian Petre claims that "reasoning is at the heart of expertise in software design" (Petre, 2009). Experts have a repertoire of reasoning strategies and choose an appropriate strategy for the task (Petre, 2009). Tang et al. provide a list of reasoning techniques that experts employ in software design such as contextualising and simplifying the solution, scenario generation, constraints consideration and trade-off analysis (Tang et al., 2010). Experts use such reasoning techniques to make decisions and refine their mental models of the software design.

Expert and novice differences

There are various differences between the mental models of novices and experts. Studies have shown that experts' mental models are more detailed and accurate than that of novices, and are better able to handle mental simulations (Adelson and Soloway, 1985; Curtis et al., 1988; Sonnentag, 1998). Experts are able to switch easily between the domain model and the situation model, as compared to novices while understanding programs (Pennington, 1987). That is, experts are able to understand the program, translate it to domain terms, and then verify the domain terms back in program terms. However novices stick to either the program or domain model, and are unable to build connections between the two models. Novices mental models are likely to be incomplete, deficient, and prone to change at any time (Sorva, 2013). Their models tend to contain surface-level features i.e., knowledge that is readily available by looking at a program, as opposed to experts whose mental models contain information about control, data flow and goals of the program (Pennington, 1987). Mental models of novices also lack firm boundaries, i.e. a novice is unclear about what aspects of a system their model covers (Sorva, 2013). Hence, experts' mental models are more stable and accurate, and draw on general principles rather than superficial characteristics (Sorva, 2013).

To summarise, expert designers develop a rich mental model of the design during the software design process. They incrementally transform their initial mental models to more concrete models based on goals at each stage of the design process. They routinely perform mental simulations on these intermediate models to arrive at the solution. *Since developing effective mental models are necessary in the design process, we hypothesize that the cognitive strategies of mental modelling and simulation are essential even in the case of evaluating software designs*. We build on this in the next section, where we review literature on specific strategies which have been used to evaluate software designs, and also provide anecdotal evidence from experts performing software design evaluation.

2.4 Strategies for Evaluating Software Designs

In this section, we describe strategies which have been used for evaluating software designs, and studies which examine comprehension and maintainability of UML diagrams. We also describe anecdotal evidence from software designers on how they went about evaluating a given software design.

2.4.1 Reading Techniques for Software Design Evaluation

Strategies for evaluating software designs have been explored in the context of inspecting UML design diagrams in order to identify defects. Inspecting design diagrams involve focusing on relevant information in these diagrams in order to create an effective mental model of the design (Winn, 1994). These cognitive skills, which are needed to identify defects are referred to as reading techniques (Hungerford et al., 2004). Experts are able to effectively evaluate a model, primarily because of effective reading techniques and cognitive processes (Wohlin and Aurum, 2004). Findings from our literature review show that experts use reading techniques like fast switching, horizontal and vertical reading, and perceptual and conceptual processes while evaluating design diagrams. We explain these strategies in detail below.

Hungerford et al. conducted a study with 12 experienced developers who were asked to perform individual reviews on a software design (Hungerford et al., 2004). The results indicate that reading techniques that rapidly switch between the two design diagrams are the most effective. The authors claim that these "fast switching" strategies help in building relationships between diagrams and create a better mental model.

Travassos et al. formulated a set of reading techniques, termed as traceability based reading, which help students integrate information across diagrams (horizontal reading) and also between diagrams and textual requirements (vertical reading) (Travassos et al., 1999). A diagrammatic representation of horizontal and vertical reading is shown in Figure 2.4. In horizontal reading, students focus their attention on a design diagram (such as class diagram), and inspect it with respect to another diagram (such as sequence diagram). Steps are provided to students on what exactly has to be done during the reading. For example, in the case of horizontal reading between a class diagram and a sequence diagram, students have to examine the sequence diagram, identify all its functions, and examine if the class diagram has these functions. They are trained to perform horizontal reading across different diagram types. As a result, they are able to read the entire design and ensure that the design is consistent, i.e. all necessary information is correctly and consistently represented in the design. This ensures that they are building an integrated and correct mental model of the design. In vertical reading, the design diagrams are read with respect to the textual requirements and use-cases. For example, a requirement is read and the specific sequence diagram which realises this requirement is analysed. If there are certain entities missing in the sequence diagram based on the given requirement, it can be added. Thus vertical reading ensures correctness and completeness in the design, and ensures that students build a complete mental model of the problem domain and map them to the corresponding design. Using these horizontal and vertical reading techniques, students were asked to report defects in the given design diagrams. Results from their study show that the techniques did lead to defects being detected.

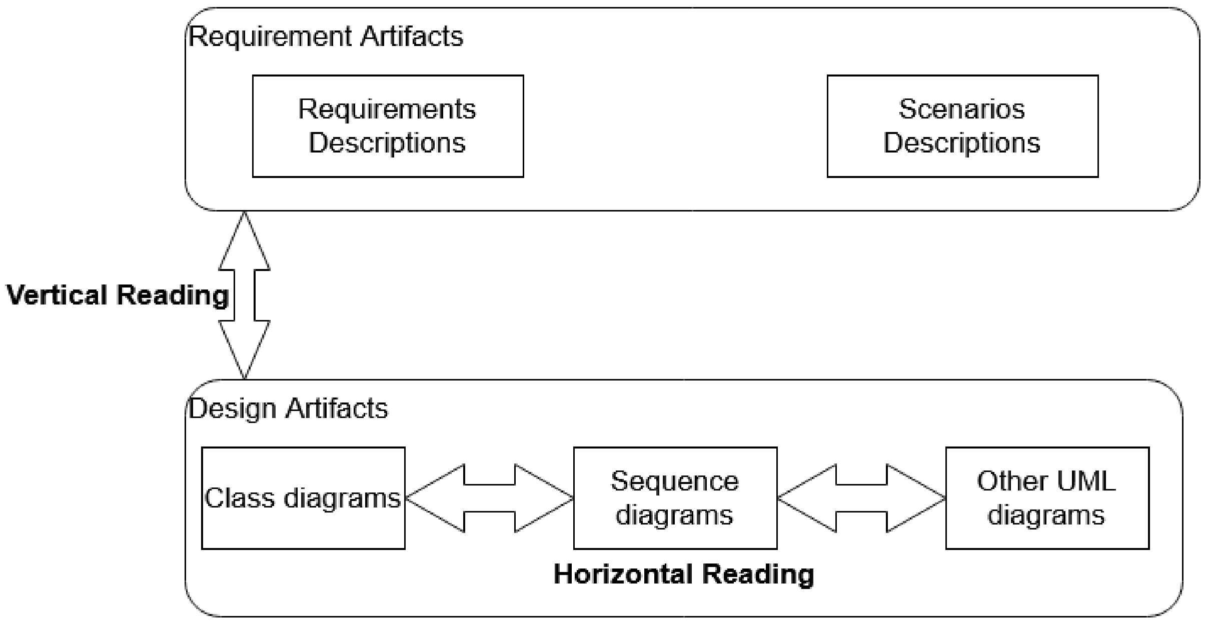


Figure 2.4: Summary of reading techniques based on Hungerford et al. (2004)

Kim et al. conducted a study to explore the cognitive processes which one uses to understand a system represented by multiple diagrams (Kim et al., 2000) . They proposed a theoretical framework which focuses on perceptual and conceptual processes. A perceptual process is a bottom-up activity of sensing something and knowing its meaning and value. Perceptual processes while reasoning with design diagrams involves linking relevant information from different diagrams. A conceptual process is a top-down activity which is used to generate, refine and validate hypotheses based on the provided design diagrams. The authors claim that reasoning with multiple design diagrams involves performing effective perceptual and conceptual processes. Based on the analysis of participants interacting with multiple diagrams, successful participants' perceptual integration processes involved making several round-trip transitions, enabling them to relate information from different diagrams. By doing so, they were able to develop an integrated mental model of the design. Conceptual integration processes involved creating and refining several hypotheses while inspecting several diagrams. Based on these hypotheses, participants refined and adapted their mental model of the design as well.

2.4.2 Anecdotal Evidence from Expert Software Designers

We conducted individual interviews with two software designers, who have extensive experience designing software, and have taught various computer science courses as well. In the interviews, they were provided with the requirements and design (1 class diagram and 3 sequence diagrams) of an automated door locking system (the requirements and design are part of future studies with students. More details about the design can be found in Section 4.3.3). The requirements were provided on paper, and the design diagrams in a modelling tool. They were asked how they would go about evaluating one of the requirements provided. We describe the process they followed below.

Expert E1 read all the requirements, and then broadly analysed the class diagram. He then proceeded to the relevant sequence diagram required to check the requirement. He analysed the messages and function calls in the sequence diagram, and switched to the class diagram to examine these functions. He then went back to the relevant requirement, and checked different parts of the requirement with the design diagrams. While doing this, he realised that a part of the requirement was not being satisfied, and flagged it as a defect (*"there is a discrepancy between the requirement and Message 11 in the sequence diagram"*)

Expert E2 also followed a similar process. He first read the requirements and broadly analysed all the design diagrams. He chose the relevant requirement for which the design was to be evaluated and divided that requirement into parts. He represented the requirement in a tree like form on paper, and drew several branches, indicating possible scenarios for the given requirement. He then proceeded to the relevant sequence diagram, and went through the messages in the sequence diagram. Like E1, E2 also went back to the class diagram to check for the methods mentioned in the sequence diagram. He used the mouse pointer to move across messages, talking out aloud the message names and the sender and receiver of messages.

From these interviews, we see certain common characteristics in the evaluation process followed by experts. They start by analysing the requirements and the design diagrams, and build a rough, initial mental model. E1 said that at the start he was *"building a big picture of what all things are there in the system, I'm not really worried about the details, what exactly are the data members, is it providing the correct return type"*. They also analyse the requirement, and think of different scenarios which realise this requirement. For example E2 said - *"basically you have to look at all the different conditions under which that property becomes relevant, and ensure that in each of these conditions, that property is satisfied. You enumerate the different scenarios and check whether these are the only scenarios, and for each scenario I check whether the property can be satisfied or not."*.

They then focus on the appropriate design diagrams, and switch between these diagrams, in order to develop an integrated understanding of the design. E1 said - *"What I'm creating in my mind is a flow of function calls, based on the class diagram and the sequence diagrams. I'm tracing the sequence diagram along with the corresponding calls that are being made in the class diagram, with the values which the variables are having"*.

From these statements, we see instances of control flow and data flow simulation, as well as instances of horizontal reading when they switch between the class and the sequence diagram. They simulate the control flow and data flow for each part of the requirement. For example, while analysing the first part of the requirement, E1 said - *"The entire sequence of actions through the class diagram and variables I have played out in my mind to say that and that is happening."*. When they encounter a discrepancy between their mental simulation and the provided design, they flag these as defects.

To summarize the previous sub-sections, we see from literature that experts apply reading techniques to build an integrated mental model of the design. Our interviews with experts also confirm this, and show that experts construct and refine their mental models of the design during the evaluation process. They also show instances of control flow and data flow simulation to evaluate their models. Hence, in order to effectively evaluate a design and ensure that it satisfies the given requirements, *students should be able to analyse and reason about the given requirements, develop an integrated mental model of the given design diagrams, visualize different scenarios, and perform control flow and data flow simulations on these models.*

2.4.3 Comprehension and Maintainability of UML Diagrams

Prior work has also explored evaluation of design diagrams in the context of reading, comprehension, and maintainability of UML diagrams (Lange and Chaudron, 2006; Chaudron et al., 2012; Budgen et al., 2011; Fernández-Sáez et al., 2016; Nugroho, 2009; Genero et al., 2011). Studies examining comprehension of UML diagrams have looked at which diagrams (and in what combination) lead to better diagram comprehension (Glezer et al., 2005; Otero and Dolado, 2004; Swan et al., 2005). For example, Swan et al. (2005) conducted a study with students comparing sequence diagrams and collaboration diagrams (collaboration diagrams are a variation of sequence diagrams). Otero and Dolado (2002) conducted a similar study comparing sequence diagrams, collaboration diagrams and state diagrams. Both studies found that sequence diagrams have the highest comprehension performance. Findings from Torchiano (2004)'s study found that students who are provided with class diagrams and object diagrams perform better than those provided with class diagrams alone.

Studies have also examined how providing UML diagrams along with source code have led to better comprehension and defect identification in source code (Dzidek et al., 2008; Scanniello et al., 2012; Arisholm et al., 2006). Arisholm et al. (2006) conducted a controlled experiment with software engineering students to investigate the impact of UML diagrams in a software maintenance task. Participants in both groups were asked to add new functionalities to the existing system and thus had to modify the source. The experimental group was provided the UML diagrams along with the code, whereas the control group was provided the code alone. Findings from this study show that there is no significant difference in time spent in making modifications. However, the authors observed that the quality of modifications were higher for those participants who were provided the UML diagrams. Dzidek et al. (2008) conducted a similar study with 20 professional developers, but the UML diagrams along with the source code had to be modified in this study. The findings are similar to Arisholm et al. (2006), in that although the group provided with UML diagrams took more time to update the source code and UML diagrams, the quality of modifications were higher. In a study conducted by Scanniello et al. (2012), the authors found that comprehension of source code increases when participants are provided with UML class and sequence diagrams.

To summarize, several studies have highlighted the usefulness of UML diagrams in software comprehension and maintenance tasks. These studies have shown that quality of code improves when UML diagrams are provided along with source code. *However, there is a lack of studies which specifically look at strategies which novices use to evaluate design diagrams and the difficulties they face.*

2.5 Summary

In this chapter, we presented an extensive literature review of current teaching in software design, student difficulties in software design, and strategies experts use while evaluating software designs. Our review of teaching-learning in software design reveals that although there are several tools to model designs, these tools are not mature enough and are complex to use. There is also a lack of pedagogical interventions which focus on the teaching-learning of software design diagram evaluation.

Our review of student difficulties in creating software designs show that they have inadequate knowledge of the domain, and inconsistent understanding between design diagrams. However, sufficient emphasis has not been given on difficulties which students face while evaluating an already given design, and the relation between these difficulties and students' mental models. We believe that these difficulties can be attributed to a deficient mental model of the design in students. An insufficient understanding of the domain can be due to an inadequate problem model. Consistency issues can be due to a lack of an elaborate and comprehensive mental model in students. We examine this in detail in Chapter 4, where we examine students' mental models of how they evaluate design diagrams against the given requirements.

Review of expert practices and strategies in software design show that experts create rich and detailed mental models of the design, employ various reasoning techniques, and perform mental simulations on these designs. They closely analyse the design by employing efficient reading techniques like fast switching, horizontal reading and vertical reading. They simulate the control flow and data flow of various scenarios while evaluating the design against the given requirements. Knowledge of expert practices can be used for the design and development of a learning environment which fosters these cognitive processes in learners. Hence, in order to foster software design evaluation skills in students, the teaching-learning environment should provide activities and affordances that enable students to

- develop an adequate mental model of the problem and the design diagrams
- employ effective reading techniques to analyse the design
- simulate various scenarios in the design and
- simulate the control flow and data flow of these scenarios in the design to ensure that it satisfies the given requirements

We go into more details of the activities and affordances in the learning environment in Chapter 5 by also drawing on literature from modelling in science education and mental models in programming education.

In the next chapter, we describe the underlying research methodology which we applied to the investigation of fostering effective software design evaluation skills in students.

Chapter 3

Research Methodology

3.1 Key Questions Guiding Research in this Thesis

As mentioned in Chapter 1, our broad research goal is to "design and develop a technology-enhanced learning environment (TELE) which enables students to effectively evaluate a software design against the given requirements"

We approach addressing this goal by answering the following questions:

1. What is the existing gap in teaching-learning of evaluating software design diagrams?
2. How do students evaluate software design diagrams and what difficulties do they face?
3. What are pedagogical strategies which can enable students to effectively evaluate software design diagrams?

Educational research is conducted based on an underlying research framework, which guides the progress of answering the broad research goal. We have chosen the design based research methodology in answering our broad research goal.

3.2 Design Based Research

Design based research (DBR) is *"the systematic study of designing, developing and evaluating educational interventions (such as programs, teaching-learning strategies and materials, products and systems) as solutions for complex problems in educational practice, which also aims at advancing our knowledge about the characteristics of these interventions and the processes of designing and developing them."* (Plomp, 2013). The design based research paradigm emerged due to certain limitations of 'traditional' research approaches such as experiments and surveys. These approaches focussed on examining learning processes as isolated variables within laboratory settings (Barab and Squire, 2004). Such results often did not translate to the naturalistic settings such as classrooms and laboratories. They also did not provide sufficient contributions towards practice, i.e. how instructors can apply these findings in their classrooms.

Design based research comprises the following phases:

1. **Problem Analysis:** In this phase, the researcher determines the research problem which needs to be addressed. The specific problem to be investigated is often determined by a synthesis of literature and/or empirical studies with students.

2. **Solution Design and Development:** Based on the problems identified in the previous phase, the researcher develops solutions which address this problem. These solutions and design decisions are grounded in existing theory relevant to the context.

3. **Evaluation and Reflection:** Research studies are conducted in order to examine the effectiveness of the designed intervention. The findings and reflections from these studies feed into the next cycle of the design based research process.

As seen in Figure 3.1, each research cycle begins by analysing a specific problem which has to be addressed. These problems stem from either the analysis of literature, or reflections from studies. Based on this analysis, certain design decisions are made towards the development of the pedagogy. The effectiveness of these decisions are measured by conducting studies which focus on certain research questions.

The DBR process is iterative in nature i.e. the cycles of problem analysis, solution design and evaluation continue till the broad research problem is satisfactorily examined. A critical component of design-based research is that findings from research should not only solve problems in the local context, but also help in advancing educational theory (Barab and Squire, 2004). Hence the output of the DBR process should also contain design principles and 'local learning theories' which can be applied to other similar contexts as well.

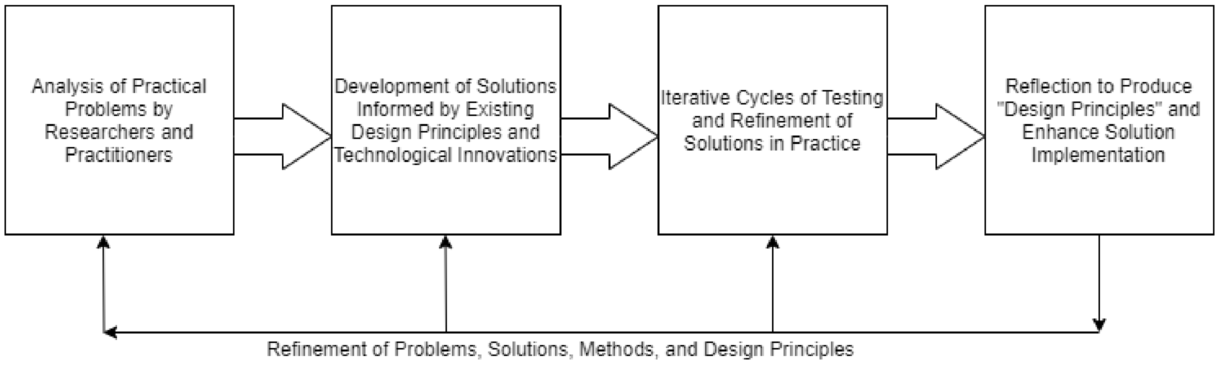

Analysis of Practical Problems by Researchers and Practitioners
Development of Solutions Informed by Existing Design Principles and Technological Innovations
Iterative Cycles of Testing and Refinement of Solutions in Practice
Reflection to Produce "Design Principles" and Enhance Solution Implementation
Refinement of Problems, Solutions, Methods, and Design Principles

Figure 3.1: Phases in a DBR Cycle

3.3 Research Cycles in this Thesis

Research in this thesis has been done over two design-based research cycles. In each cycle, we have looked at uncovering aspects of the three key questions guiding our research - exploring existing gaps in teaching-learning, understanding students' evaluation processes, and pedagogical strategies to help them in their evaluation process. The DBR cycles in this thesis have been summarised in Figure 3.2.

3.3.1 DBR Cycle 1

The primary objectives of DBR Cycle 1 are to (i) Identify existing gaps in teaching-learning of software design evaluation, (ii) Identify student difficulties while evaluating designs, and (iii) Design a pedagogy which addresses these gaps and difficulties. The primary contributions of this cycle are (i) Identifying student difficulties and (ii) the VeriSIM pedagogy which comprises the design tracing strategy.

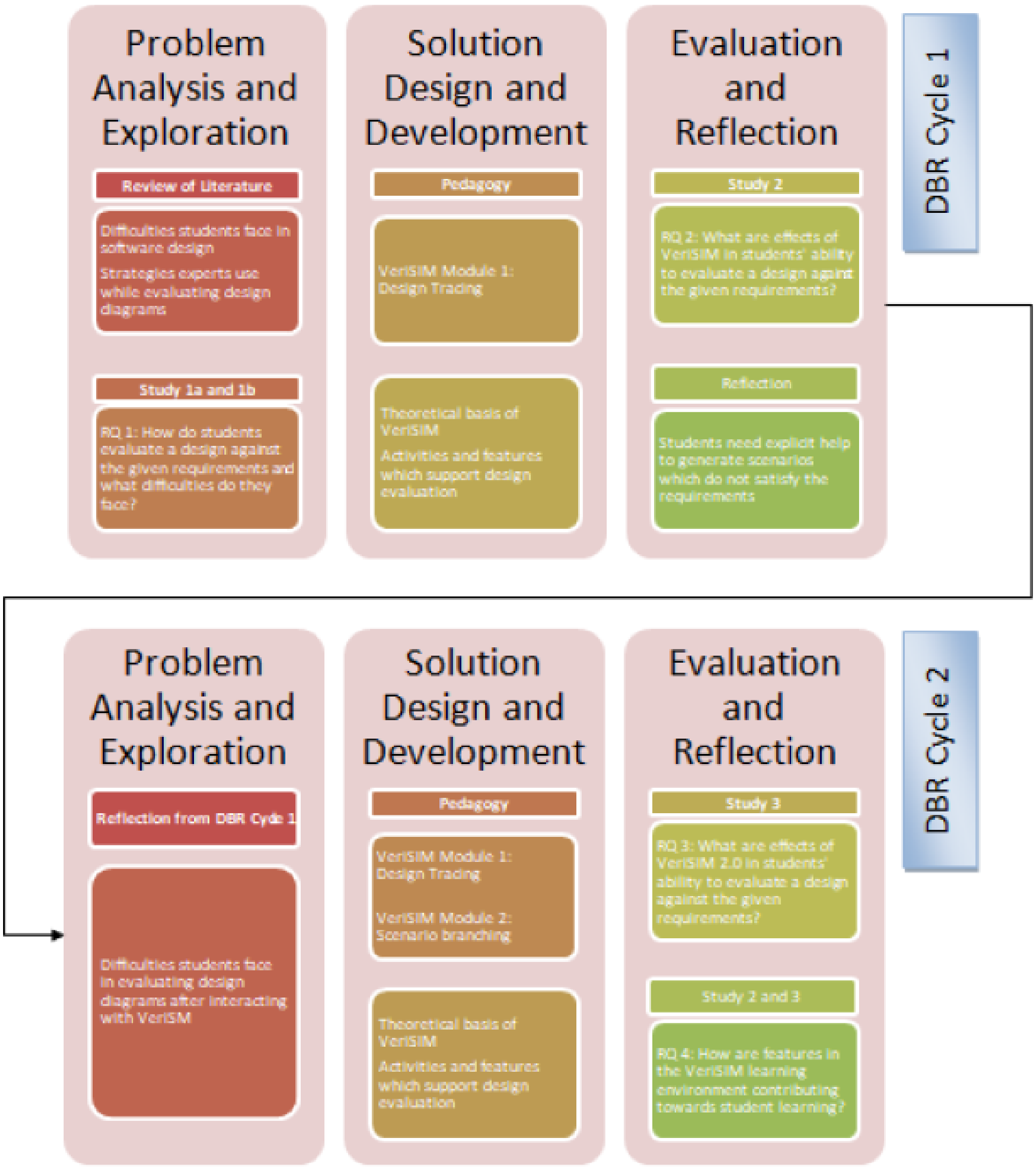

Problem Analysis and Exploration
Review of Literature
Difficulties students face in software design
Strategies experts use while evaluating design diagrams
Study 1a and 1b
RQ 1: How do students evaluate a design against the given requirements and what difficulties do they face?
Solution Design and Development
Pedagogy
VeriSIM Module 1: Design Tracing
Theoretical basis of VeriSIM
Activities and features which support design evaluation
Evaluation and Reflection
Study 2
RQ 2: What are effects of VeriSIM in students' ability to evaluate a design against the given requirements?
Reflection
Students need explicit help to generate scenarios which do not satisfy the requirements
DBR Cycle 1
Problem Analysis and Exploration
Reflection from DBR Cycle 1
Difficulties students face in evaluating design diagrams after interacting with VeriSM
Solution Design and Development
Pedagogy
VeriSIM Module 1: Design Tracing
VeriSIM Module 2: Scenario branching
Theoretical basis of VeriSIM
Activities and features which support design evaluation
Evaluation and Reflection
Study 3
RQ 3: What are effects of VeriSIM 2.0 in students' ability to evaluate a design against the given requirements?
Study 2 and 3
RQ 4: How are features in the VeriSIM learning environment contributing towards student learning?
DBR Cycle 2

Figure 3.2: DBR cycles in the thesis

1. **Problem Analysis and Exploration:** In this phase, we explored literature in order to identify existing gaps in teaching and learning of software design evaluation. We examined strategies and cognitive processes which experts use during the software design process. We also reviewed literature on student difficulties in software design, and realised that sufficient emphasis has not been given on identifying student processes and difficulties in evaluating design diagrams. This led us to conduct two studies (Study 1a and Study 1b) with the focus on understanding how students approach an evaluation task and the difficulties they face.

2. **Solution Design and Development:** Our analysis of literature, and findings from the novice studies informed the design of the VeriSIM pedagogy. This helped us make design decisions regarding the pedagogy and activities in VeriSIM. We operationalised the pedagogy into the VeriSIM learning environment.

3. **Evaluation and Reflection:** In this phase, we investigated the effects of the VeriSIM TELE on students' ability to model scenarios and evaluate the design diagrams against the given requirements (Study 2).

3.3.2 DBR Cycle 2

In DBR Cycle 2, we reflected on certain difficulties students faced in evaluating designs, even after interaction with VeriSIM. The objectives of this cycle are (i) The redesign of VeriSIM, based on reflections from previous study and (ii) Understanding how students learn using various features in the VeriSIM learning environment.

1. **Problem Analysis and Exploration:** In this phase, we reflected on the difficulties faced by students in Study 2. The main reflection was that students need explicit help to generate and identify scenarios which do not satisfy requirements. We analysed literature to address this difficulty and inform the redesign of VeriSIM.

2. **Solution Design and Development:** In this phase, we developed the scenario branching strategy, which helped students explicitly identify scenarios from the requirements. This led to the revised VeriSIM pedagogy, which comprises the design tracing and scenario

branching strategy. We operationalised the revised VeriSIM pedagogy by adding another module in VeriSIM. The modified TELE VeriSIM 2.0 has two modules. Module 1 is identical to the intervention provided in the previous cycle. In Module 2, the scenario branching pedagogy has been operationalised as a worksheet, which is facilitated by an instructor.

3. **Evaluation and Reflection:** We conducted a study (Study 3), which investigated the effects of VeriSIM 2.0. We also looked at how various features in VeriSIM 2.0 helped students evaluate the design diagrams against the given requirements.

3.4 Research and Design Questions answered in this Thesis

There are two categories of questions which we answer in this thesis: Design Questions (DQ) and Research Questions (RQ).

1. Design Questions (DQ) - DQs are answered by referring to how specific education and learning science theories can be operationalised into the pedagogy design which addresses students' difficulties or improves their understanding. DQs are answered in the 'Solution Design and Development Phase' of the DBR cycle.
2. Research Questions (RQ) - RQs are answered by conducting research studies to examine how students' approach a given task, and to understand the effects of the designed pedagogy. RQs are answered in the 'Problem Analysis' and 'Evaluation' phase of the DBR Cycle.

In this thesis, we answer 4 RQs and 2 DQs across the two cycles of DBR.

RQ 1: How do students evaluate a design against the given requirements and what difficulties do they face?

We investigated RQ 1 in the 'Problem Analysis and Exploration' phase of DBR Cycle 1 (see Figure 3.2). To answer RQ 1, we conducted two studies. We conducted the first study (Study

1a) with 100 students. The goal of this study was to identify categories of responses students provided when they were asked to evaluate a design against the given requirements. Details about Study 1a can be found in Section 4.2. The research question guiding this study is -

RQ 1.1: How do students evaluate a software design against the given requirements?

To answer this RQ, we analysed student responses using content analysis (Baxter, 1991). We analysed each response and came up with categories of student responses. Answer to RQ 1.1 was useful in helping us determine what are the broad types of responses, and what students may be thinking when asked to evaluate a given design.

To understand students' cognitive processes and their underlying mental models during the evaluation task, we conducted a qualitative study with 6 students (Study 1b). Details about Study 1b can be found in Section 4.3. The research questions guiding this study are -

RQ 1.2: What defects are students able to identify in the design evaluation task?
RQ 1.3: What reading strategies do students use to evaluate software design diagrams against the given requirements?
RQ 1.4: What are the elements in their mental model?

To answer RQs 1.2-1.4, we used video data of participants solving an evaluation task, their writing, and interviews after the task, as the primary data sources. We used video data analysis (Derry et al., 2010) to infer student reading strategies, and thematic analysis (Braun and Clarke, 2012) to infer elements in their mental model. Study 1b revealed that students' ability to identify defects depends on their mental model of the design, when used appropriately with a strategy they use to read the design diagrams. We found that students have difficulty in simulating dynamic behaviours i.e. the control and data flow of various scenarios in the design.

Hence, the key insight gained from RQ 1 is that scaffolding students to identify and model (simulate the control and data flow of) relevant scenarios in the design can lead to effective evaluation of the design diagrams against the given requirements. These findings and insights informed the design of the VeriSIM pedagogy.

DQ 1: What is an appropriate pedagogy that can scaffold students to identify and model various scenarios in the design?

We investigated DQ 1 in the 'Solution Design and Development' phase of DBR Cycle 1. To answer DQ 1, we draw insights from model based learning in science education, and mental models in introductory programming, to design the VeriSIM pedagogy. In VeriSIM, students are introduced to the design tracing strategy. In design tracing, students simulate the control and data flow of various scenarios in the design, by constructing a model similar to a state diagram.

We have operationalised the design tracing strategy into the VeriSIM learning environment. In VeriSIM, students go through stages and challenges which help them progressively trace various scenarios in the design. Our hypothesis is that as students trace various scenarios in the design, they will be able to effectively evaluate the design against the given requirements.

RQ 2: What are effects of VeriSIM in students' ability to evaluate a design against the given requirements?

We investigated RQ 2 in the 'Evaluation and Reflection' phase of DBR Cycle 1. RQ 2 is part of Study 2, which was conducted with 86 students. Details about Study 2 can be found in Section 6.1. The research questions guiding this study are -
RQ 2.1: What are the effects of VeriSIM in students' ability to simulate dynamic behaviours in the design?
RQ 2.2: What are the effects of VeriSIM in students' ability to identify defects in the design?

Students solved a pre-test, followed by interaction with VeriSIM, and solved a post-test in the end. We analysed the differences in students' pre-post responses to understand the effects of the pedagogy in students' ability to trace scenarios and uncover defects (RQ 2.1 and 2.2). We also conducted focus group interviews, and analysed the transcripts using a thematic analysis approach (Braun and Clarke, 2012) in order to understand students' perceptions of the usefulness of the VeriSIM TELE and its features.

The key reflections from Study 2 was that although students were able to simulate dynamic behaviours of various scenarios in the design, they were not able to identify scenarios in the design which do not satisfy the requirements. Hence, these reflections formed the basis of the next DBR cycle, where we sought to refine the VeriSIM pedagogy to include explicit scaffolds to identify scenarios in the design.

DQ 2: What additional scaffolds are needed to enable students to identify scenarios in the design?

We investigate DQ 2 in the 'Solution Design and Development' phase of the second DBR Cycle. Based on the reflections from the previous cycle, we refined the VeriSIM pedagogy and included the scenario branching strategy. The theoretical basis of scenario branching is taken from cognitive mapping techniques used in requirement analysis of software design (Montazemi and Conrath, 1986). In scenario branching, students are provided with a mapping tool and scaffolds which enable them to construct a scenario tree. As they traverse each path of the tree from the root to a leaf, they identify various scenarios for each requirement. The scenario branching strategy has been incorporated as a worksheet which is facilitated by an instructor. The design tracing strategy and the scenario branching strategy has been incorporated into the VeriSIM 2.0 learning environment.

RQ 3: What are effects of VeriSIM 2.0 in students' ability to evaluate a design against the given requirements?

To investigate the effects of VeriSIM 2.0 (RQ 3), we conducted Study 3 with 22 students. Details about Study 3 can be found in Section 7.6. The research questions guiding this study are -

RQ 3.1 What are the effects of VeriSIM 2.0 in students' ability to identify scenarios in the design?
RQ 3.2 What are the effects of VeriSIM 2.0 in students' ability to uncover defects in the design?

The study procedure and analysis methods were similar to Study 2.

RQ 4: How are features in the VeriSIM learning environment contributing towards student learning?

RQ 2 and RQ 3 helped us understand the effects of the design tracing and the scenario branching strategies in students' ability to identify and model scenarios, and evaluate the design diagrams against the given requirements. RQ 4 investigates how features in VeriSIM contributed towards students' learning. The VeriSIM learning environment captured various student interactions and has stored them as logs. These included actions such as how they constructed the state diagram model during design tracing, their answers to certain reflection and evaluation questions etc. We analysed these logs along with the focus group interviews in order to understand how various feature in VeriSIM contributed towards student learning. More details about how we answered RQ 4 can be found in Chapter 8.

3.5 Analysis Methods used in this Thesis

In this section, we describe the analysis methods we used to analyse the research questions of this thesis. This section primarily serves as a reference to analysis descriptions in subsequent chapters.

3.5.1 Video Data Analysis

To answer RQ 1.3 (What reading strategies do students use to evaluate software design diagrams against the given requirements?), we used the video data analysis method outlined by Derry et al. (Derry et al., 2010). The authors provide guidelines for researchers conducting research examining video data using complex learning environments. Specifically, they address challenges pertaining to Selection i.e., deciding which elements of a video data analysis to select for study, and Analysis i.e., deciding what analytical frameworks are appropriate for the given research problem. The key steps in selecting and analysis of video data are outlined below.

- **Chunking -** In this step, researchers may chunk the video records into segments. The criteria for chunking can be event boundaries i.e. dividing a video based on specific events which occur. It can also be chunked based on time markers, i.e. dividing a video based on specific time intervals (e.g. every 5 seconds etc.) Researchers may choose relevant parts of the video to chunk at first, depending on guiding questions, or to understand unexpected phenomena which occurred.

- **Marking, Transcribing and Categorizing -** After chunking, researchers may mark certain chunks of video which are of interest. They also create intermediate representations which aid them in the analysis process. Some commonly used representations are indexing (time-indexed notes which provide a basic outline of the events), macrolevel coding (mark major events as topics or themes), narrative summaries (lengthy description of events) and transcription (transcripts that represent some portion of the events recorded). These representations are used in the next step of analysis and reporting.

- **Analysis and Reporting** - The analysis and reporting of video data is often interleaved. The result of video data analysis depends on the research questions which is being asked, and can lead to quantitative or qualitative findings. For example, researchers can mark chunks of video, and produce codes or themes for each of these chunks. They can then conduct a quantitative analysis on the corpus of codes to generate findings related to number of codes, or patterns occurring in the codes. In a qualitative research study, researchers can provide a rich and detailed description of the video chunks, and hence do not count types of events.

3.5.2 Content Analysis

We used content analysis to analyse students' written responses which described the defects they identified based on the given design. The purpose of the content analysis method is to take data sources, such as texts and analyses, and transform them into a summary form. This can be done by either categorising these texts into pre-existing categories, or generate categories based on the data. It involves breaking down the text into units of analysis, performing certain statistical analysis of these units, and presenting the analysis in an appropriate form. We draw on the content analysis method described by Cohen (Cohen et al., 2017), and describe the key steps.

- **Defining the unit of analysis** - The text to be analysed must be broken down into appropriate units so that analysis can be performed on these texts. These unit of analysis can be at different levels, such as a word, phrase, sentence, paragraph, or even the whole text. The unit of analysis must be defined before proceeding further.
- **Coding of texts** - Once the unit of analysis is defined, appropriate codes have to be chosen which map to these texts. These codes can be at different levels of specificity and generality. Researchers can start with a prescribed set of codes, but usually, the codes are also derived from the data responsively. As a researcher starts assigning codes, the codes themselves go through modifications as coding progresses. Hence the researcher might have to go through a data set more than once, to ensure consistency, refinement, modification and exhaustiveness of coding.
- **Construct categories based on codes** - After performing coding, the researcher is able to detect patterns among codes. Different codes are then linked and grouped together under broad categories and subcategories. During this process, memos and notes are created which describe the category definitions and criteria for assigning descriptive codes to these categories.
- **Analysing the codes and categories** - Once the texts have been assigned codes and categories, the researcher can count the frequency of each code and/or category. Based on these frequencies, the researcher can summarize the inferences from the text, look for patterns and relationships between the codes and categories. Statistical analysis such as graphical representations, factor analysis, regression etc. can be done to summarize these results.

3.5.3 Thematic Analysis

We used thematic analysis to analyse individual and group interviews with students, as well as student responses to certain open ended questions in the TELE. The purpose of thematic analysis is to identify patterns of meaning across a dataset that provide an answer to the research questions. The process followed in thematic analysis is similar to content analysis. However, the key difference is that while content analysis uses a descriptive approach in its interpretation of quantitative counts of the codes, thematic analysis provides a purely qualitative, detailed and nuanced account of the data. The different themes emerging from the data can help the researcher in richly describing the phenomenon under investigation.

We broadly followed the thematic analysis approach outlined by Braun and Clarke (2012), and describe the steps we used in the analysis.

- **Familiarization with the data** - In this step, researchers go through transcripts once or twice in order to familiarize themselves with the data. They usually make observational notes as they read through these transcripts.

- **Generating descriptive and inferential codes** - Once researchers are familiar with the data, they create codes which describe a particular sentence or groups of sentences. The usually start by creating descriptive codes. Descriptive codes stay close to the data, and researchers make sure that they avoid any prejudices or preconceptions while creating these codes. They then generate interpretative or latent codes which identify meanings underlying the descriptive codes. As coding progresses, researchers will usually modify existing codes to incorporate new data. By the end of this step, researchers will have enough codes to capture the diversity as well as patterns within the data.

- **Searching, reviewing and defining themes** - In this step, researchers review the coded data to identify areas of similarity and overlap between codes, and assign themes to these codes. Themes capture something important about the data in relation to the research question, and represents some level of meaning within the data set. During this step, researchers also have to define the theme, its boundaries, and whether there is enough data to support this theme.

- **Reporting the themes** - Finally, while presenting the findings, the themes should be presented in a systematic manner. Themes should connect logically and meaningfully and, if relevant, should build on previous themes to tell a coherent story about the data.

3.6 Summary

In this chapter, we described the overarching methodology guiding our research. We describe Cycle 1 in Chapters 4, 5 and 6, and Cycle 2 in Chapter 7 and 8. Details of the research questions, analysis methods and findings are described in detail in subsequent chapters.

Chapter 4

Characterising Student Mental Models and Identifying Difficulties in Evaluating Design Diagrams

In Chapter 2, we reviewed literature to understand the cognitive processes of expert software designers, and the difficulties students face in the design process. We observed that there hasn't been much research emphasis on how students evaluate a given design, the elements in their mental models while evaluating designs, and the difficulties they face. In this chapter, we specify the mental model elements for design diagrams by adapting the Block Model (Schulte, 2008), a framework which has been used for teaching-learning of program comprehension. This adaptation serves as the basis for identifying the elements in students' mental models, as they perform design evaluation. We describe two studies (Study 1a and Study 1b), which we conducted with students, to understand how they approach a given evaluation task. In Study 1a, we conducted a study with 100 final year undergraduates. We sought to understand what categories of written responses students provide when they are asked to "identify defects in the design based on the requirements." These categories gave us certain indicators of what students were thinking while they were doing the evaluation task. In Study 1b, we conducted a study with 6 computing undergraduates, where we focussed on inferring students' strategies and their mental models of the design during the evaluation task. Findings from both these studies gave us insights to the pedagogical design of VeriSIM.

4.1 Characterizing Mental Models for Software Design Diagrams

In this thesis, we have adapted the "Block Model" proposed by Schulte (Schulte, 2008) to describe the elements of the mental model for design diagrams. The Block Model is an educational model for program understanding. It breaks down the process of comprehending a given program based on understanding blocks of code, present at different levels and dimensions (more details follow in Section 4.1.1). By developing student understanding of each of these blocks, the overall understanding of the program also improves. Thus, situating teaching and learning of programming in terms of the Block Model can help researchers and teachers understand what learning process and competencies are needed by learners. It also helps support theory-driven processes of developing pedagogical interventions, such as learning sequences, and teaching methods (Schulte, 2008)

The Block Model has been designed primarily to help students understand programs, where the external representation is in the form of program code. Schulte et al. have suggested that future research directions for the Block Model can involve modification of the external representation to other artifacts like UML diagrams (Schulte et al., 2010). However, this adaptation has not been explored. Hence, we adapt the Block Model to describe the elements of the mental model for design diagrams.

The adapted model serves two purposes. First, the adapted model can be used as a lens to investigate students' mental models and infer what elements are present and what are absent as they evaluate the design diagrams against the given requirements (Study 1a and 1b). Second, the model can inform the design of a pedagogy which focuses on improving student understanding of specific element(s) of the mental model (Section 5.7).

4.1.1 The Block Model

The Block Model for program comprehension is described in Figure 4.1. The model contains blocks arranged across different levels and dimensions. The model can be considered from two perspectives. The first perspective, which corresponds to the vertical axis in the diagram, refers to four levels based on zooming in and out of a program. The lowest level refers to a single expression or line of code, and moves up to blocks of code, the relation between these blocks, and finally to the entire program. For example, to comprehend a program, students need to understand the syntax and semantics of each line of code. They then integrate their understanding of multiple lines of code to make sense of blocks of code. These blocks can refer to functions or other logical units such as loops and conditionals. They then infer the relation between different blocks, and how each block contributes to the purpose of the overall program. This bottom-up approach is one way of comprehending a given program. A top-down approach of program comprehension is also possible, whereby students first broadly understand the overall structure of a given program, and relations between different parts of the program. They then focus their attention to specific blocks of interest to understand the program.

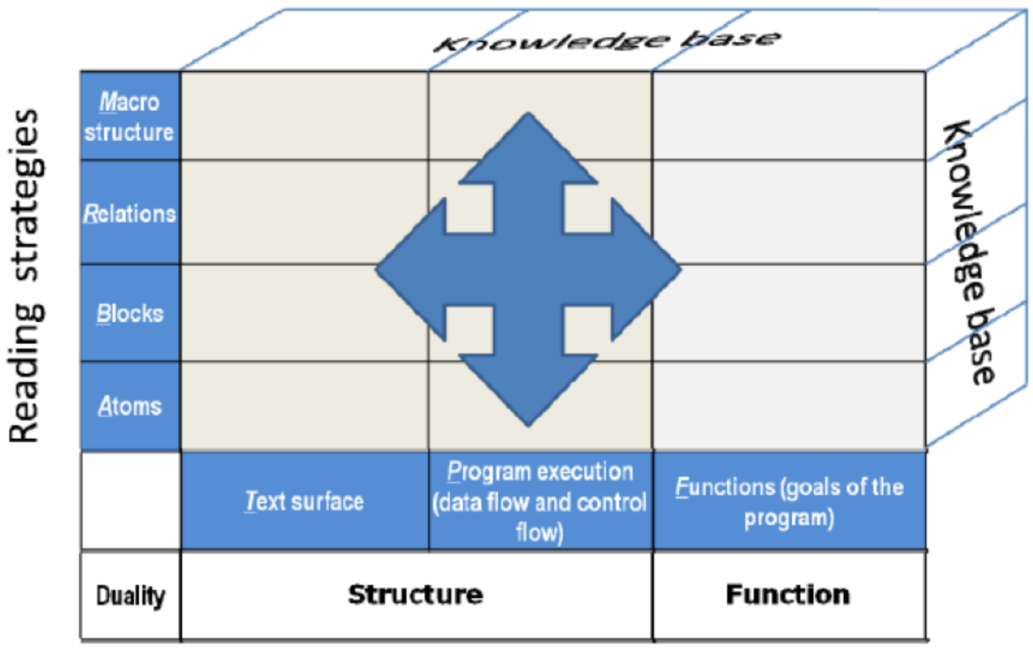


Figure 4.1: The block model for program comprehension, taken from Schulte et al. (2010)

The bottom-up and top-down approaches are known as reading strategies (shown in the vertical axis of Figure 4.1). Reading strategies (or assimilation processes), are used to build and augment the mental model of a given program. In program comprehension, several reading strategies like top-down (Soloway and Ehrlich, 1984), bottom-up (Pennington, 1987), or opportunistic strategies (Letovsky, 1987), are used by programmers to build a mental representation of the given program.

The second perspective, which corresponds to the horizontal axis of Figure 4.1, contains three levels - (1) The text surface, which is a static entity (2) The program execution, referring to the control flow and data flow, which is a dynamic entity and (3) The function and purpose of the program, which also refers to the main goals of the program. While comprehending a given program, students can focus purely on the program text, such as the language elements of atoms and blocks, and the overall structure of the program text. This corresponds to the first level in the horizontal axis. However, understanding how the static text representation is being executed, and what changes occur at each of the program levels is also essential. For example, when a line or block of code is executed, students should understand what changes occur to variables in that state. Students should also be able to understand the flow of control when different methods are called. Thus, the second level of the horizontal axis refers to the flow of control and data, and how it relates to the broad algorithm of the program. The third level corresponds to the purpose or goal corresponding to atoms, blocks and the whole program in its given context. Students should be able to map a line, block or the entire program to its purpose in the given context. For example, in the case of a library management system, students should understand that the purpose of calling a sorting sub-routine is to display all books in alphabetical order.

In addition to these two perspectives, the knowledge base encompasses the understanding about the above mentioned elements, as well as the knowledge about programming language syntax, semantics, and discourse rules (e.g. guidelines for comments, indentation rules etc.).

4.1.2 Adaptation of the Block Model for Design Diagrams

How can this model be adapted when the external representation is design diagrams? Both perspectives mentioned above, hold to a certain degree in the case of design diagrams as well. In the case of design diagrams, the vertical axis corresponds to (1) Atoms (2) Blocks (3) Relations and (4) Macrostructure present in specific design diagrams. Unlike code, where there exists a single external representation, the above elements are present in each type of UML diagram, as we explain below.

1. **Atoms -** In class diagrams, atoms refer to the data members and member functions in each class, and their access specifiers. In sequence diagrams, atoms refer to each message in the sequence diagram.

2. **Blocks -** In class diagrams, each class can be considered as a block which encompasses the data members and functions. In sequence diagrams, a group of messages are aggregated to form blocks which can specify a particular operation or function.

3. **Relations -** In class diagrams, relations refer to associations between classes. In sequence diagrams, relations refer to links between different logical blocks.

4. **Macrostructure -** The macrostructure refers to the design as a whole, and the purpose of each design diagram in the design.

The second perspective (i.e. the vertical axis) also holds for design diagrams. In the case of design diagrams, the three levels are (1) The diagram surface elements, which is the static entity (2) The dynamic behaviour, which is the control flow and the data flow, and (3) The main goals of each diagram. We describe the three levels in detail below:

1. **Diagram surface elements -** These elements are the basic functions and data members in structural diagrams, and messages in behavioural diagrams. For example, the data members and member functions in the class diagram and specific messages in the sequence diagram correspond to the surface elements.

2. **Dynamic behaviours (control flow and data flow) -** Control flow refers to the order in which operations in the elementary blocks are carried out, such as sequence, loop or conditional. For design diagrams, it refers to the control structure present in behavioural diagrams like the sequence, interaction and activity diagrams. Data flow refers to the transformation that data variables undergo in the course of plan execution (Burkhardt et al., 2002). Hence for design diagrams, it involves knowledge about how data members

in the structural diagrams change based on plan execution in the behavioural diagrams. For example, the sequence diagram for an ATM withdrawal feature would involve updating the "balance" variable (present in the class diagram) after the withdraw function call.

3. **Main goals -** Main goals in a program correspond to functions achieved by the program viewed at a higher level of granularity and do not correspond to single program units. In design diagrams, main goals refer to the purpose of each diagram and its role in the design. A sequence diagram for a particular functionality can be composed of a sequence of sub-goals to realise that functionality. For example, a sequence diagram for an ATM authentication feature will include sub-goals such as user input and password validation.

In addition to the knowledge about the elements mentioned above, knowledge about design diagrams and the problem domain are also essential in developing an accurate mental model. **Design diagram knowledge** is based on the syntax and semantics of different design diagrams. For example, students should understand various symbols and representations in class and sequence diagrams, and what they mean in the context of software design. **Problem domain knowledge** refers to the knowledge about the context for which the design has been developed. For example, in the case of an ATM system design, students should be familiar with the working and operations of an ATM system.

In the case of design diagrams, what are effective **reading strategies**? As mentioned in Section 2.4, experts employ effective reading techniques like horizontal and vertical reading to comprehend design diagrams. We consider the horizontal and vertical reading techniques as reading strategies required to build the mental model. These strategies help learners assimilate and make sense of different UML diagrams. In horizontal reading, students integrate information across different diagrams. For example, students read the sequence diagram and check if the various functions and classes mentioned are present in the class diagram. This ensures that the diagrams are consistent. Knowledge of different design diagrams are required for horizontal reading. In vertical reading, students read the requirements and the design diagrams in order to check for completeness and correctness. Both design diagram knowledge and problem domain knowledge are required for vertical reading. As these reading techniques are applied, the student's mental model of the design is updated.

To summarize, our view of the mental model for design diagrams is adapted from the Block model. This adaptation of the mental model elements for design diagrams is summarised in Figure 4.2. It has the following components - knowledge about the diagram surface elements, dynamic behaviours, main goals, design diagram knowledge and problem domain knowledge. For the sake of simplicity, we have combined the horizontal axis elements (i.e. atoms, blocks, relations and macrostructure) for each of the mental model elements (i.e the diagram surface elements, dynamic behaviours and the main goals). Horizontal and vertical reading strategies are used to build and augment the mental model of a given software design.

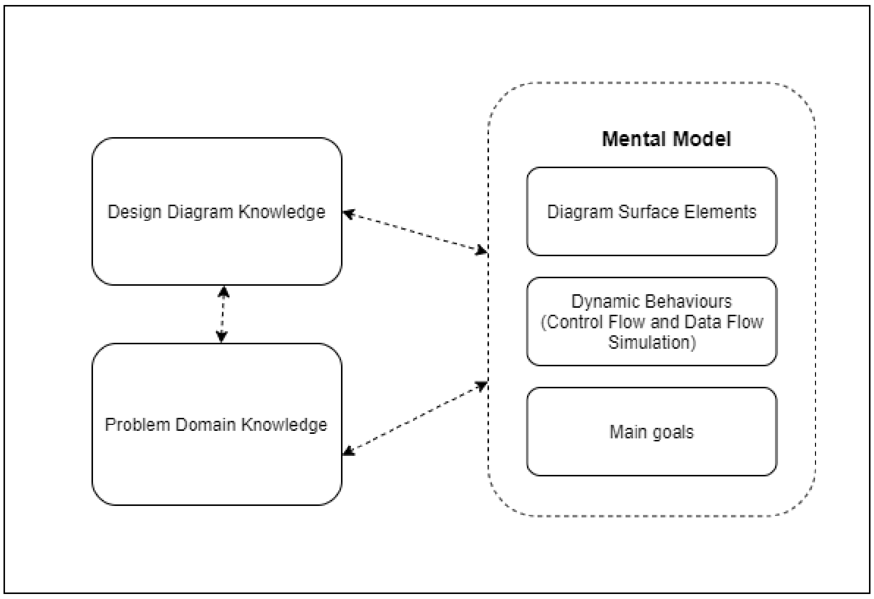


Figure 4.2: Mental model elements for design diagrams (Adapted from the block model)

In Study 1a and 1b, we anchor our analysis of students' evaluation of design diagrams based on this adapted model. We examine what reading strategies they use, and whether students' mental model of the design contains information about the diagram surface elements, the control flow and data flow, and the main goals in design diagrams.

4.2 Study 1a: Categorizing student responses of evaluating design diagrams

In Study 1a, we analysed student written responses when they were asked to identify defects in a given design. The goal of this study was to examine categories of responses students provide, and also how these responses correspond to the mental model elements of design diagrams.

4.2.1 Research Question

The research question guiding this study is -
RQ 1.1: How do students evaluate a software design against the given requirements?

We intend to answer RQ 1.1 by analysing students' written responses when they are asked to evaluate a design against the given requirements. We categorize their written responses to understand the elements of the design they focus on during the evaluation task. Answers to RQ 1.1 provide indicators for how students approach a software design evaluation task. These findings motivate Study 1b, where we delve deeper into their mental model elements and reading strategies while evaluating a given design.

4.2.2 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 4.3) and three sequence diagrams - Register (Figure 4.4), Withdraw (Figure 4.5) and Change PIN (Figure 4.6). 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.

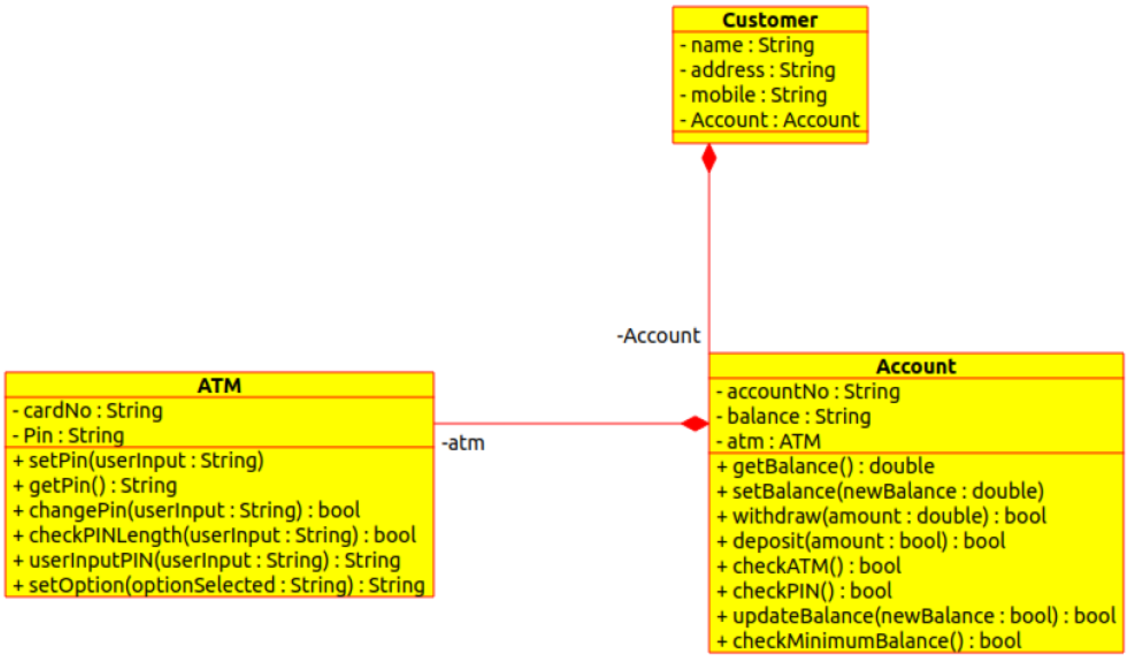
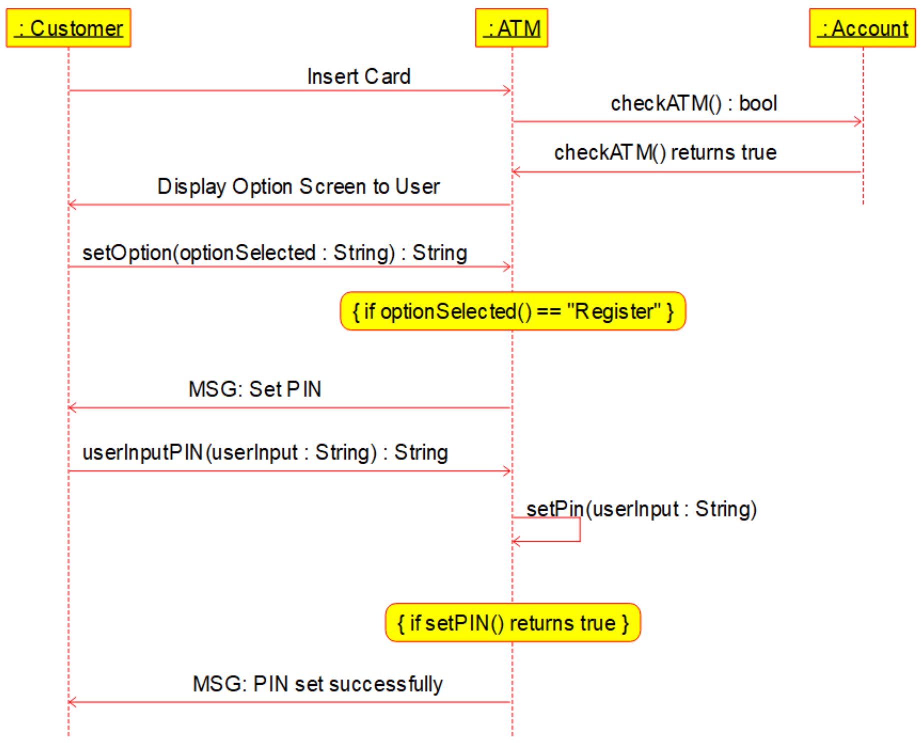

Customer
- name : String
- address : String
- mobile : String
- Account : Account
-Account
ATM
- cardNo : String
- Pin : String
+ setPin(userInput : String)
+ getPin() : String
+ changePin(userInput : String) : bool
+ checkPINLength(userInput : String) : bool
+ userInputPIN(userInput : String) : String
+ setOption(optionSelected : String) : String
-atm
Account
- accountNo : String
- balance : String
- atm : ATM
+ getBalance() : double
+ setBalance(newBalance : double)
+ withdraw(amount : double) : bool
+ deposit(amount : bool) : bool
+ checkATM() : bool
+ checkPIN() : bool
+ updateBalance(newBalance : bool) : bool
+ checkMinimumBalance() : bool

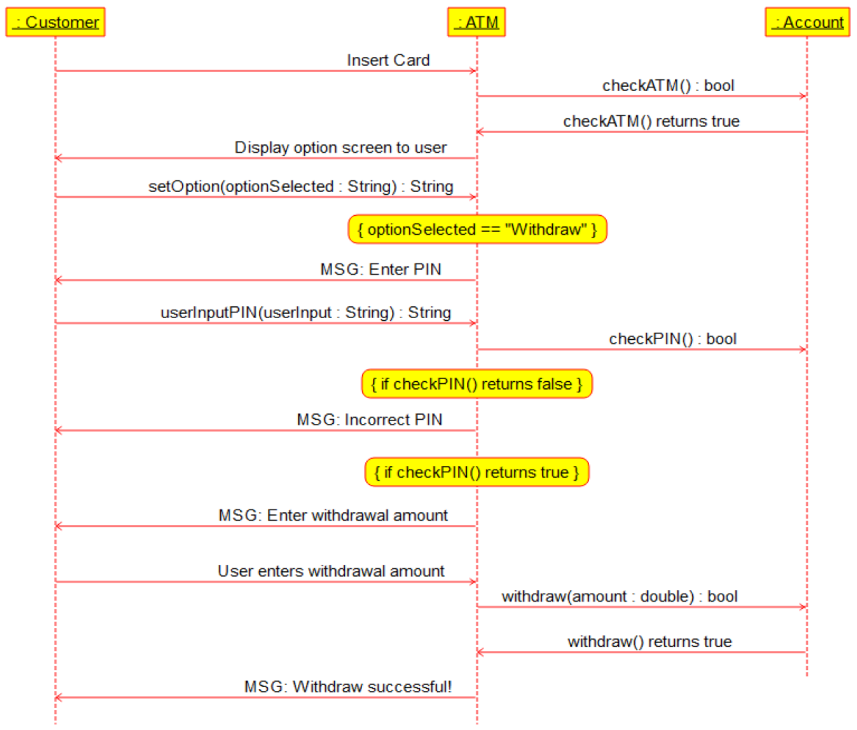
Figure 4.3: Class diagram of the ATM system

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



: Customer
: ATM
: Account
Insert Card
checkATM() : bool
checkATM() returns true
Display Option Screen to User
setOption(optionSelected : String) : String
{ if optionSelected() == "Register" }
MSG: Set PIN
userInputPIN(userInput : String) : String
setPin(userInput : String)
{ if setPIN() returns true }
MSG: PIN set successfully

Figure 4.4: Sequence diagram for registering the ATM card



: Customer
: ATM
: Account
Insert Card
checkATM() : bool
checkATM() returns true
Display option screen to user
setOption(optionSelected : String) : String
{ optionSelected == "Withdraw" }
MSG: Enter PIN
userInputPIN(userInput : String) : String
checkPIN() : bool
{ if checkPIN() returns false }
MSG: Incorrect PIN
{ if checkPIN() returns true }
MSG: Enter withdrawal amount
User enters withdrawal amount
withdraw(amount : double) : bool
withdraw() returns true
MSG: Withdraw successful!

Figure 4.5: Sequence diagram for withdrawing amount

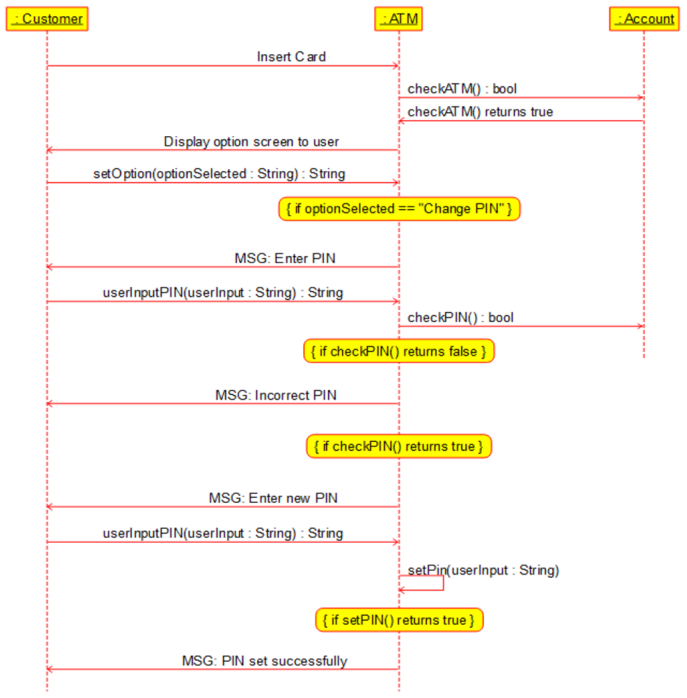

: Customer
: ATM
: Account
Insert Card
checkATM() : bool
checkATM() returns true
Display option screen to user
setOption(optionSelected : String) : String
{ if optionSelected == "Change PIN" }
MSG: Enter PIN
userInputPIN(userInput : String) : String
checkPIN() : bool
{ if checkPIN() returns false }
MSG: Incorrect PIN
{ if checkPIN() returns true }
MSG: Enter new PIN
userInputPIN(userInput : String) : String
setPin(userInput : String)
{ if setPIN() returns true }
MSG: PIN set successfully

Figure 4.6: Sequence diagram for changing pin

4.2.3 Defects introduced in the design

We have focussed on how students evaluate semantic deficiencies in the design, as mentioned in Chapter 1. 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 identifying the main goals in the diagrams, and simulating the dynamic behaviour (control flow and data flow) of scenarios in the design.

Hence, the mental model elements for design diagrams (Figure 4.2) provides a basis for how these semantic deficiencies can be uncovered. To uncover these type of defects, students need knowledge of the syntax and semantics of the design diagrams, the problem domain (how an ATM system works), and knowledge about the main goals and dynamic behaviour in a given design.

For example, consider the defect - *"There is no check if the withdrawal amount is greater than the balance."* To identify this defect, one has to examine the class diagram (Figure 4.3) and identify relevant data members, such as *'balance'* and member functions such as *'checkATM()'*, *'checkPIN()'*, and *'withdraw()'*. The main goals for the withdrawal use case is identified by analysing the withdraw sequence diagram (Figure 4.5). These goals refer to logical parts of the sequence diagram, such as 'Checking ATM and display options', 'Checking PIN' and 'Withdraw'. One then simulates the control flow and data flow for each goal, and realises that there is no check when $amount > balance$ and when the *'withdraw'* function returns false. Hence, by identifying the diagram surface elements, main goals and simulating the control flow and data flow, semantic deficiencies can be identified.

4.2.4 Study Procedure

We conducted the study with 100 final year (fourth year) computer engineering and information technology engineering students (61 male and 39 female), in their own institution. The engineering institution is located in a metropolitan city in our country. Participants filled a consent form prior to the study. Participation in the study was completely voluntary, and participants could withdraw from the study at any point during the study. 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 provided the task sheet which contained the requirements, the design diagrams and a question asking them to identify defects. I 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.

We now describe the data analysis and how it answers RQ 1.1. A summary is provided in

Table 4.1.

Table 4.1: Data source and analysis methods for Study 1a

Research Question	**Data Source**	**Data Analysis Method**
RQ 1.1:How do students evaluate a software design against the given requirements?	Student response sheets	Content analysis of student response to the evaluation question

4.2.5 Data Analysis

I analysed the answers of students using the content analysis method (Baxter, 1991) to come up with categories for the defects identified (Details about the method can be found in Section 3.5.2). The steps which I followed to analyse the data are as follows.

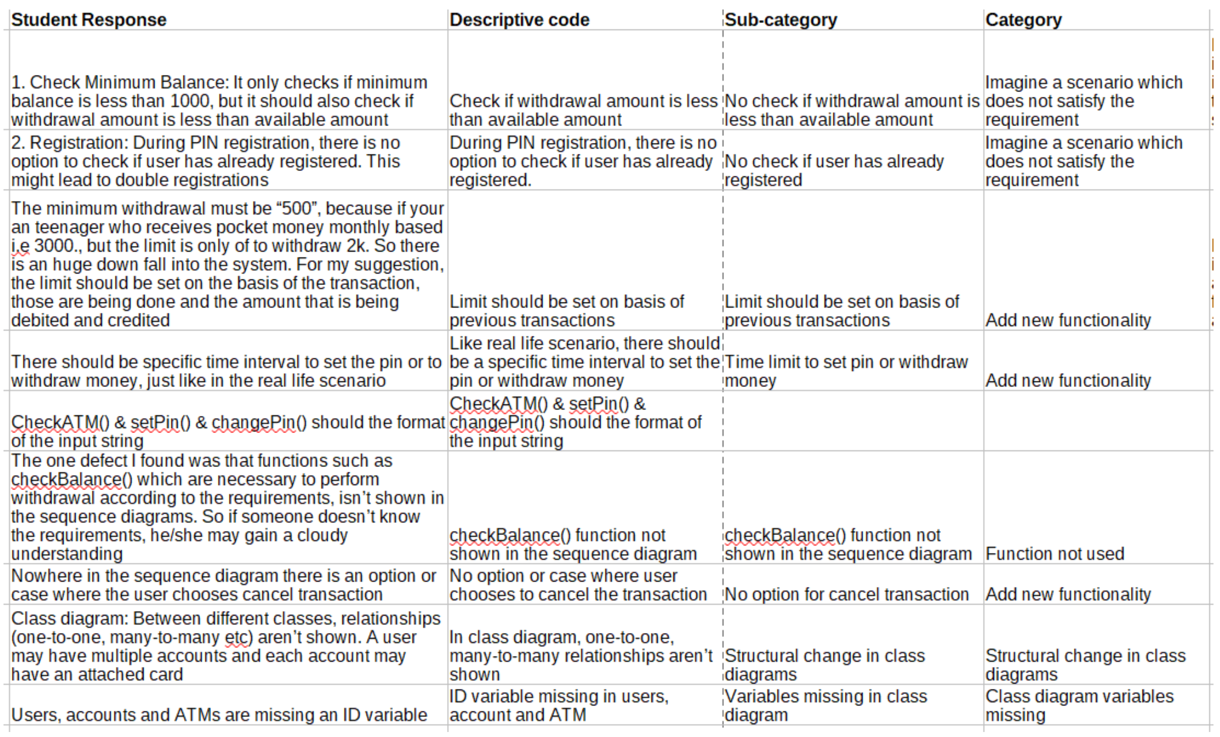
1. **Representation of student answers -** The written text which each participant wrote as a response to the question was typed verbatim to a spreadsheet. I considered a written sentence or group of sentences which referred to a particular defect, as the unit of analysis. Each row in the spreadsheet corresponded to a sentence/sentences written by participants. Out of 100 students, one student's response was not clear and hence was excluded from the analysis. Many students listed multiple defects. I identified 168 answers from 99 student response sheets.

2. **Descriptive coding -** I assigned a descriptive code to each sentence. 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 -** I inferred categories and sub-categories based on the patterns, explanations and relationships between the descriptive codes. The categories reflected the meaning inferred from the descriptive codes and can explain larger

segments of data. I created memos and notes which described 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.”*

A snippet of the spreadsheet used for the analysis is shown in Figure 4.7. I performed descriptive coding and came up with the initial set of categories and sub-categories. To establish reliability of the generated categories, two rounds of coding were done with a rater. In the first round, I provided the rater with 20 student responses and the list of categories. After independently assigning categories to the given set of responses, the rater and I discussed the categories corresponding to each response, refined category names and definitions, and came to an agreement on conflicting entries. In the second round, the rater and I independently assigned categories to another 10 responses and reached near agreement (90%). Based on the refined categories and definitions, I independently assigned categories to the remaining responses.



Student Response	Descriptive code	Sub-category	Category
1. Check Minimum Balance: It only checks if minimum balance is less than 1000, but it should also check if withdrawal amount is less than available amount	Check if withdrawal amount is less than available amount	No check if withdrawal amount is less than available amount	Imagine a scenario which does not satisfy the requirement
2. Registration: During PIN registration, there is no option to check if user has already registered. This might lead to double registrations	During PIN registration, there is no option to check if user has already registered.	No check if user has already registered	Imagine a scenario which does not satisfy the requirement
The minimum withdrawal must be “500”, because if your an teenager who receives pocket money monthly based i,e 3000., but the limit is only of to withdraw 2k. So there is an huge down fall into the system. For my suggestion, the limit should be set on the basis of the transaction, those are being done and the amount that is being debited and credited	Limit should be set on basis of previous transactions	Limit should be set on basis of previous transactions	Add new functionality
There should be specific time interval to set the pin or to withdraw money, just like in the real life scenario	Like real life scenario, there should be a specific time interval to set the pin or withdraw money	Time limit to set pin or withdraw money	Add new functionality
CheckATM() & setPin() & changePin() should the format of the input string	CheckATM() & setPin() & changePin() should the format of the input string		
The one defect I found was that functions such as checkBalance() which are necessary to perform withdrawal according to the requirements, isn’t shown in the sequence diagrams. So if someone doesn’t know the requirements, he/she may gain a cloudy understanding	checkBalance() function not shown in the sequence diagram	checkBalance() function not shown in the sequence diagram	Function not used
Nowhere in the sequence diagram there is an option or case where the user chooses cancel transaction	No option or case where user chooses to cancel the transaction	No option for cancel transaction	Add new functionality
Class diagram: Between different classes, relationships (one-to-one, many-to-many etc) aren’t shown. A user may have multiple accounts and each account may have an attached card	In class diagram, one-to-one, many-to-many relationships aren’t shown	Structural change in class diagrams	Structural change in class diagrams
Users, accounts and ATMs are missing an ID variable	ID variable missing in users, account and ATM	Variables missing in class diagram	Class diagram variables missing

Figure 4.7: Snapshot of the spreadsheet used for analysing student responses

4.2.6 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, number of student responses, and percentage of responses in each category is shown in Table 4.2. Definitions and examples of categories and sub-categories are summarized in Table 4.3.

Table 4.2: Number of student responses in each category for Study 1a

Category of student response	Number of student responses	Percentage
Identify scenarios which do not satisfy requirements	41	24.4%
Identify necessary functions which are not used	24	14.3%
Change existing functionalities and requirements	37	22%
Add new functionality	18	10.7%
Change data types, functions of class diagram	21	12.5%
No defects	4	2.4%
Blank	23	13.7%
Total	**168**	**100%**

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 were 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.

Table 4.3: Categories of defects based on analysis of student responses for Study 1a

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.	-	-

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 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 sub-category of student response was 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 categorized 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, rather than evaluating the design against the requirements.

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

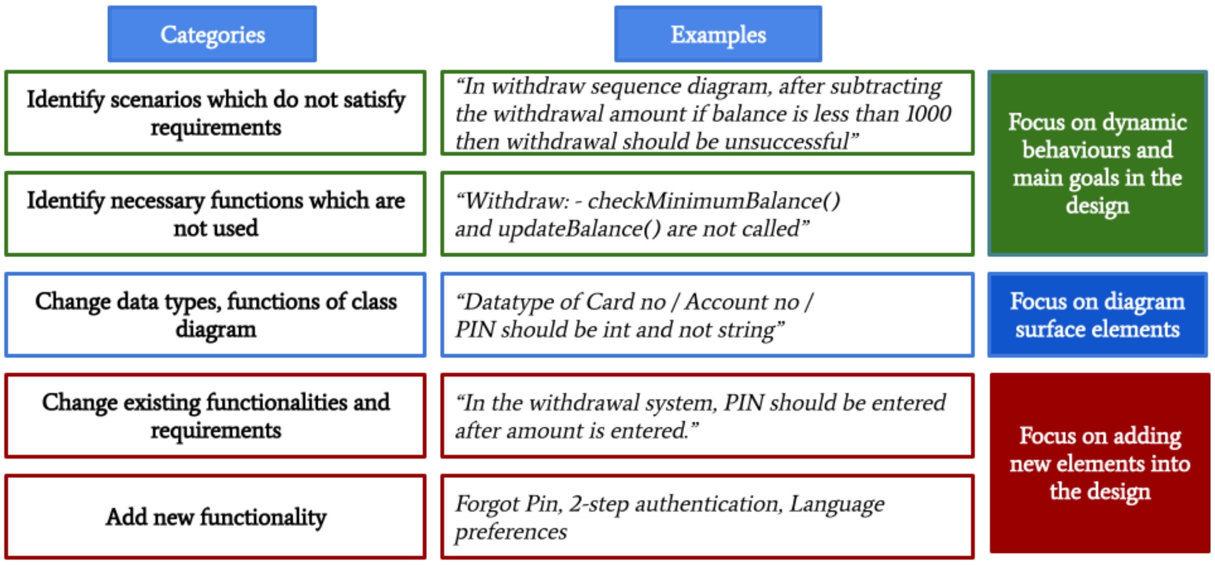
Another category of student responses were based on changes in the class diagram and change in data type of variables. For example, students incorrectly stated that datatype of card number, account number, balance and PIN should be 'int' rather than 'string'. Students stated that some variables are missing in the class diagram and also 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 and 7: No Defects (2.4%) and Blank responses (13.7%)

23 students did not provide any responses. 4 students explicitly mentioned that there were no defects. None of them mentioned any reasons for why they did not find any defects. Students leaving no responses may indicate that they did not understand the evaluation task. Students specifically mentioning that there are no defects can indicate that they examined the diagrams and could not identify any defects.

4.2.7 Inferring students' mental model elements of the design based on identified categories

The categories of student written responses identified can also be grouped based on the mental model elements of design diagrams. Figure 4.8 summarizes the mapping between the response categories and the mental model elements for design diagrams. We see that students' responses map to the diagram surface elements, dynamic behaviours in the design, as well as on new requirements and functionalities. Figure 4.9 depicts this grouping in the form of a Venn diagram. (For example: 5 students focused exclusively on the diagram surface elements. 2 students focused on all three types) We describe the response categories below.



Categories	Examples	
Identify scenarios which do not satisfy requirements	*"In withdraw sequence diagram, after subtracting the withdrawal amount if balance is less than 1000 then withdrawal should be unsuccessful"*	Focus on dynamic behaviours and main goals in the design
Identify necessary functions which are not used	*"Withdraw: - checkMinimumBalance() and updateBalance() are not called"*	
Change data types, functions of class diagram	*"Datatype of Card no / Account no / PIN should be int and not string"*	Focus on diagram surface elements
Change existing functionalities and requirements	*"In the withdrawal system, PIN should be entered after amount is entered."*	Focus on adding new elements into the design
Add new functionality	*Forgot Pin, 2-step authentication, Language preferences*	

Figure 4.8: Mapping categories of student responses to design diagram mental model elements

Focus on diagram surface 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 category of responses give indicators that students' mental models primarily contain knowledge regarding 'diagram surface elements'*. By specifying these type of defects, students restricted their focus to a surface-level comprehension of the design diagrams, without simulating dynamic behaviours in the design. Hence, we can infer that students who focussed only on the diagram surface elements need more scaffolds to map the requirements and the main goals and simulate dynamic behaviours in the design

Focus on dynamic behaviours and main goals 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 type of defects. They mapped the requirements to the main goals of the design diagrams. 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 can indicate that they simulated the control flow and data flow of various scenarios pertaining to the given requirements and identified semantic defects in the design. *These response categories give indicators that students' mental models contain knowledge of the main goals as well as the control and data flow across different design diagrams based on the requirement.*

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 behaviour of adding functionalities is due to students' prior knowledge of the problem domain (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, their problem domain knowledge forms a prominent part of their mental model. 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. *These categories give indicators that students' problem domain knowledge influences their mental model and causes them to add new elements into the design.* Extraneous knowledge about the problem domain interferes with identifying the main goals and simulating scenarios in the design which do not satisfy the requirements. Hence, we can infer that students need scaffolds to help them remain focussed on identifying relevant scenarios which do not satisfy the requirements, as opposed to adding new functionalities and requirements to the design.

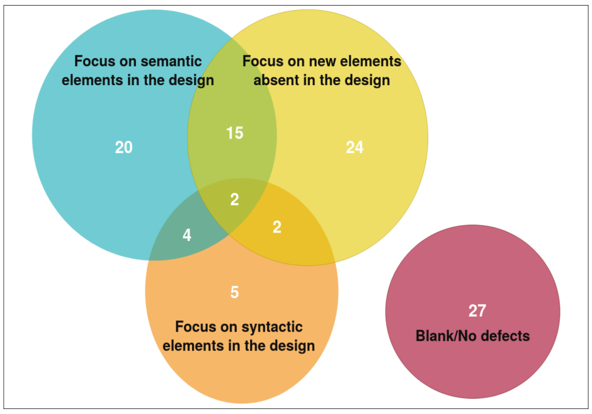


Figure 4.9: Venn diagram depicting which aspects of the design diagrams students focus on

It is to be noted that adding and modifying existing functionalities in itself is not an undesirable behaviour. 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 (Adelson and Soloway, 1985). The solution space can be expanded by generating alternative solutions using various techniques like brainstorming, analogous thinking, attribute listing, etc. (Liu and Schonwetter, 2004). 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.

4.2.8 Reflections

Study 1a helped us identify a range of response categories students provided when asked to evaluate a given design against the requirements. We also showed how each of these categories can be mapped to the mental model elements for design diagrams. Apart from focusing on the diagram structure and the dynamic behaviour (control flow and data flow) of the design, students also brought in extraneous knowledge from the problem domain into their mental model.

These insights provide the basis for Study 1b, where we use qualitative methods to delve deeper into understanding students' strategies and mental models as they evaluate the design against the given requirements.

4.3 Study 1b: Characterizing students' mental models while evaluating design diagrams

In Study 1b, we provided a similar evaluation task as Study 1a to students, whereby they were asked to provide a logical explanation of how the design satisfies/does not satisfy a given requirement. Students' written responses give us answers to what defects they are able to identify in the given evaluation task. We also investigated what reading strategies students' used and what students' mental models contained during the evaluation task. We did this by analysing data from sources such as participant video recordings and interviews.

In Study 1b, we focussed on students' reading strategies and mental models, as we wanted to uncover how students' ability to identify defects depend on their reading strategies (e.g. horizontal and vertical reading) and their mental model elements. Based on the answers to these research questions, we identified difficulties which they faced in the evaluation task which informed the design of our intervention.

4.3.1 Research Questions

The research questions guiding Study 1b are:

- RQ 1.2 - What defects are students able to identify in the design evaluation task?
- RQ 1.3 - What reading strategies do students use to evaluate software design diagrams against the given requirements?
- RQ 1.4 - What are the elements in their mental model?

4.3.2 Participants

We conducted the study with 6 computer science undergraduate students of which 3 were in their third year and 3 were in their fourth year of their bachelors in technology (BTech CS) programme. Participants were recruited from engineering colleges in our city. Among the participants, 3 were male and 3 were female. All the participants had undergone a software design course in the third year of their undergraduate studies and hence all of them were familiar with UML diagrams.

4.3.3 Description of the Task

The participant was provided with a set of requirements and the design of an automated door locking system, specified as a set of UML diagrams. The design was seeded with some defects. The task assigned to the participant was as follows - *"For each requirement, your task is to provide a logical explanation for how the design satisfies/does not satisfy the requirement. You are free to use any notation/diagrams to support your explanation."* (Appendix B contains details of the task, UML diagrams and other information provided to participants)

The requirements provided to the participant were as follows:

- *R1. If the passcode hasn't been set yet, the user can register and enter a required passcode.*
- *R2. When the user chooses the lock option, and enters the correct passcode, the door should lock. If the passcode is incorrect, the door remains unlocked.*
- *R3. When the user chooses the unlock option, and enters the correct passcode, the door should unlock. If the passcode is incorrect, the door remains locked.*
- *R4. The door should lock/unlock only if it is closed.*

The UML diagrams provided to the participant are as follows -

- Use case diagram of the door locking system (Figure B.1)

- Class Diagram of the door locking system (Figure B.2)
- Sequence diagram of the register use case (Figure B.3)
- Sequence diagram of the lock use case (Figure B.4)
- Sequence diagram of the unlock use case (Figure B.5)

Table 4.4 shows the defects corresponding to each requirement and an explanation of how they can be uncovered. 7 defects were introduced in the design diagrams corresponding to the requirements. Uncovering these defects require knowledge of the mental model elements for design diagrams (Figure 4.2). Students need knowledge about the design diagrams and problem domain, should identify the main goals in the design, and be able to simulate the control flow and data flow in the given design. Defect D1 arises due to the absence of a check in the register sequence diagram if a user is already registered. Defects D2 and D4 are due to an incorrect behaviour when the incorrect passcode is entered (the value of door changes state, rather than remaining in the same state). Defects D3 and D5 are due to the absence of a check in the initial state of the door variable. In the design diagrams, there is no condition to check if the door is closed, which results in defects D6 and D7.

4.3.4 Study Procedure

A summary of the study procedure is shown in Figure 4.10. The study was conducted with a single participant at a time. At the start, each participant signed a consent form (The consent form format is shown in Appendix A). The participant was provided with the task sheet containing the requirements and the task as outlined in Section 4.3.3. The solution for the task was to be written on the same sheet. Additional sheets were also provided. The UML diagrams of the automated door locking system was provided in Umbrello [1], a popular UML modelling software (see Figure 4.11 for a screenshot of the Umbrello interface). The participant was also provided with an information sheet which contained an overview of UML diagrams, use-case diagram, class diagram and sequence diagram. They were allowed to refer to this sheet at any point during the task. (The task and information sheet is shown in Appendix B).

[1] https://umbrello.kde.org/

Table 4.4: Description of the defects introduced in the design diagrams

Requirement	**Defect**	**Explanation of how the defect can be uncovered**
R1	D1 - Register Sequence Diagram - There is no check if the passcode has already been set	Read the requirements and the register sequence diagram. Simulate the scenario where the user is already registered
R2	D2 - Lock Sequence Diagram - When incorrect passcode is entered, the door should not unlock but remain in the same state	Read the requirements, class diagram and the lock sequence diagram. Simulate the change of state in the lock variable when unlock() function is called in the else part
	D3 - Lock Sequence Diagram - Initial state of door is not specified (i.e. the door should be in the unlocked state)	Read the requirements and the lock sequence diagram. Simulate the initial state of the door
R3	D4 - Unlock Sequence Diagram - When incorrect passcode is entered, the door should not lock but remain in the same state	Read the requirements, class diagram and the unlock sequence diagram. Simulate the change of state in the lock variable when lock() function is called in the else part
	D5 - Unlock Sequence Diagram - Initial state of door is not specified (i.e. the door should be in the locked state)	Read the requirements and the unlock sequence diagram. Simulate the initial state of the door
R4	D6 - Lock Sequence Diagram - There is no condition to check if the door is closed	Read the requirements, the class diagram and the lock sequence diagram. Variable 'close' or any of its member functions have not been specified in the class diagram or in the lock sequence diagram
	D7 - Unlock Sequence Diagram - There is no condition to check if the door is closed	Read the requirements, the class diagram and the unlock sequence diagram. Variable 'close' or any of its member functions have not been specified in the class diagram or in the unlock sequence diagram

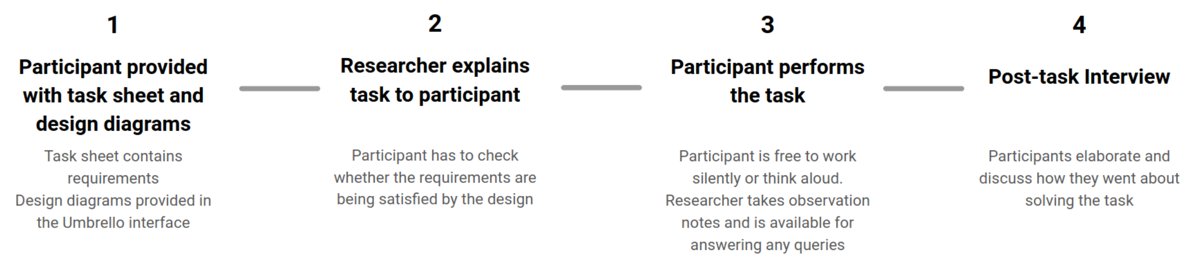

1
Participant provided with task sheet and design diagrams
Task sheet contains requirements
Design diagrams provided in the Umbrello interface
2
Researcher explains task to participant
Participant has to check whether the requirements are being satisfied by the design
3
Participant performs the task
Participant is free to work silently or think aloud.
Researcher takes observation notes and is available for answering any queries
4
Post-task Interview
Participants elaborate and discuss how they went about solving the task

Figure 4.10: Summary of Study1b procedure

At the start of the study, I provided the task sheet to the participant and explained the task objective (i.e. the goal of this task is to evaluate if the design satisfies each intended requirement). I gave a brief description of the Umbrello interface and provided the participant with the UML diagrams of the door locking system. From this point on, the participant was free to explore the UML diagrams and verify the requirements in any order. The participant was free to speak out loud or to work silently. Even though these options were given, each participant worked silently and only spoke when they had to clarify something. I was present in the same room as the participant was solving the task, making periodic observations of what the participant did as he/she performed the task.

After the participant finished the task, I conducted a semi-structured interview asking the participant to describe what they did during the task. I started by asking him/her broad questions like - *" What did you understand about the task?"*, *"How did you go about doing the task?"*. Based on their response, I probed them deeper and asked specific questions about how they went about solving for each requirement. (e.g.: *"Which diagram were you referring to while solving for this requirement?"* and *"Which part of the class diagram did you observe for a particular requirement?"*). The objective of this interview was to encourage the participant to elaborate and discuss their thinking of how they went about solving the task.

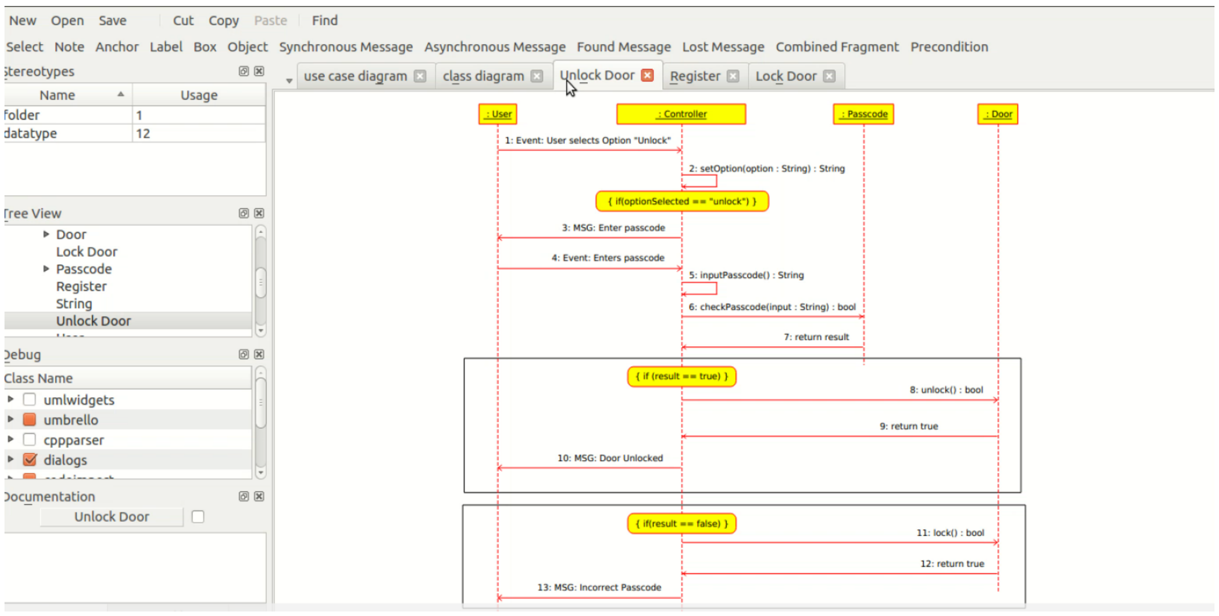


Figure 4.11: Screenshot of the Umbrello interface

4.3.5 Data Sources

To answer the research questions, multiple data sources were used. Table 4.5 provides the mapping of the research questions, and the corresponding data sources and analyses.

Table 4.5: Data source and analysis methods for Study 1b

Research Question	**Data Source**	**Data Analysis Method**
RQ 1.2: Defects students identify	Participant writing	1. Evaluating identified defects
RQ 1.3: Reading strategies	Video of students' performing the task and screen capture	2. Video data analysis
RQ 1.4: Mental model elements	1. Participant writing 2. Post-task interview	1. Evaluating identified defects 2. Thematic analysis of audio transcripts

Participants' writing on the task sheets were used to identify their performance in the evaluation task (RQ 1.2). Analysis of the task sheet answers enabled us to identify whether they were able to uncover the semantic deficiencies in the design based on the requirements. The entire study session and the post-task interview was recorded on a video camera. The video data served as the primary source to infer students' reading strategies (RQ 1.3). The participant writing and post-task interview served as the data sources to identify student's mental models as they were solving the evaluation task (RQ 1.4).

As participants were solving the task, they performed various actions. For example, they looked at the screen, clicked on various elements of the UML diagrams, switched between UML diagrams, looked at the task sheet, wrote on the task sheet etc. In order to capture all these rich interactions, we chose video and screen capture as our data sources. The camera was placed in an angle which captured the participant's screen, the task sheet and the actions performed by the participant. A recording of the screen was also done during the entire session in order to capture important information like mouse clicks, mouse movements, transition from one diagram to the other. Data from the screen capture was primarily used in cases where the video data could not clearly capture certain elements on the screen. Hence screen capture was not directly used in the data analysis, but only to augment the video data information in certain cases.

To identify students' mental models while solving the evaluation task, I used data from participant writing and from the audio transcripts of the post-task interview. Participants' written responses were analysed in a manner similar to Study 1a, to infer categories of mental model elements. In the post-task interview, questions were based on the observations which I made while the participant was solving the task. These observations served as anchor points for the interview later on. For example, actions like multiple switching between different diagrams, and spending a lot of time on a particular diagram were anchor points which I used to elicit what participants were thinking during that time. The questions were also based on answers to each requirement, which the participant wrote on the given task sheet. Observations of these writings on the task sheet and their interactions with the UML diagrams at a particular time, served as mechanisms to uncover what participants were thinking.

4.3.6 Data Analysis

I analysed participant writing to identify what defects they were able to identify (RQ 1.2). I used video data analysis to infer reading strategies (RQ 1.3), and a thematic analysis approach on participants' writing and post-task interview transcripts to infer their mental model elements (RQ 1.4). A summary of the steps in the analysis for RQ 1.3 and 1.4 is shown in Figure 4.12.

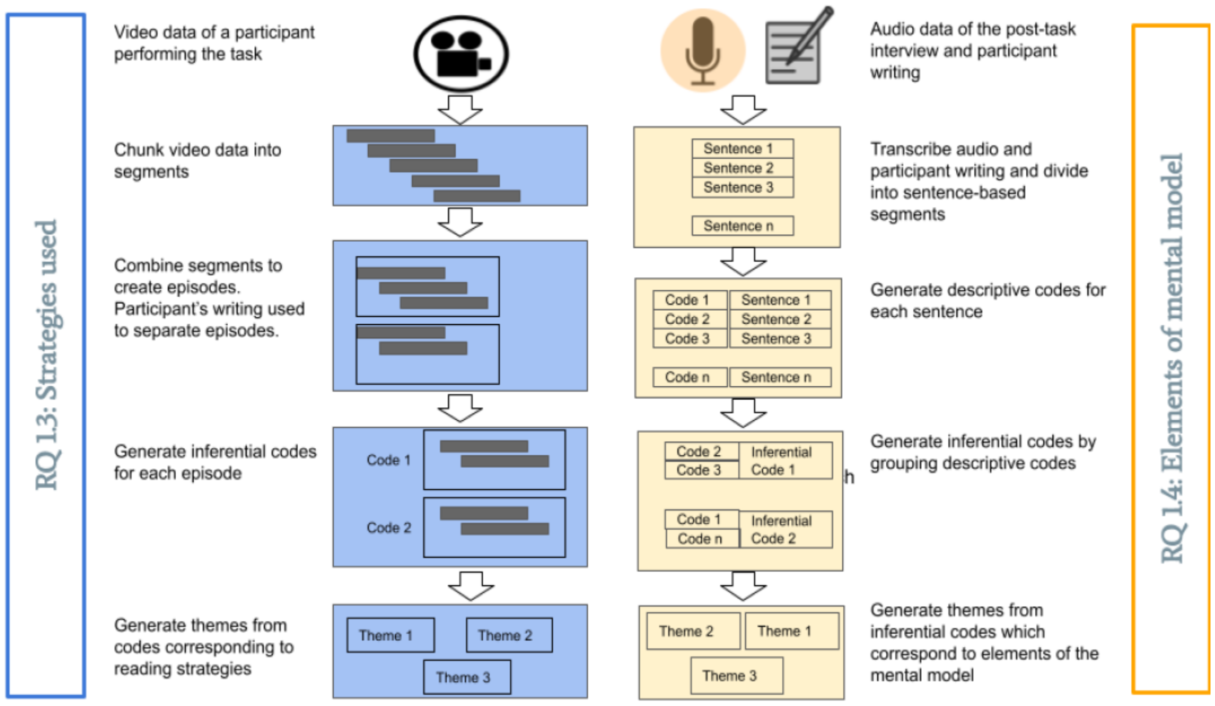

RQ 1.3: Strategies used
Video data of a participant performing the task
Chunk video data into segments
Combine segments to create episodes. Participant's writing used to separate episodes.
Generate inferential codes for each episode
Code 1
Code 2
Generate themes from codes corresponding to reading strategies
Theme 1
Theme 2
Theme 3
Audio data of the post-task interview and participant writing
Transcribe audio and participant writing and divide into sentence-based segments
Sentence 1
Sentence 2
Sentence 3
Sentence n
Generate descriptive codes for each sentence
Code 1
Code 2
Code 3
Code n
Generate inferential codes by grouping descriptive codes
Inferential Code 1
Inferential Code 2
Generate themes from inferential codes which correspond to elements of the mental model
RQ 1.4: Elements of mental model

Figure 4.12: Summary of the analysis steps for RQ 1.3 and 1.4

Analysis method to infer defects identified by students (RQ 1.2)

I analysed participant response sheets in order to understand the defects they were able to identify/not identify. For each participant, I analysed the explanation they wrote for each requirement, and checked if it corresponds to the semantic defects which were seeded in the design diagrams.

Analysis method to infer reading strategies (RQ 1.3)

To answer RQ 1.3, I analysed the video data for identifying reading strategies. I adapted the video analysis data method outlined by Derry et al. (Derry et al., 2010) and used thematic analysis (Braun and Clarke, 2012) with the goal of identifying common themes. These themes correspond to different reading strategies which students used. (These methods are described in Section 3.5).

The steps I followed to analyse the video data were as follows:

1. **Chunking and descriptive coding of video data into segments** - I chunked each participant's video data of the task based on participant's direction of attention and action. I then assigned a code for each segment, which was adapted from Hungerford et al. (Hungerford et al., 2004). The codes are as follows:

   - INF - Information Sheet
   - SP - Statement of Problem
   - UCD - Use Case Diagram
   - CD - Class Diagram
   - RegisterSD - Register Sequence Diagram
   - LockSD - Lock Sequence Diagram
   - UnlockSD - Unlock Sequence Diagram
   - Write - Write on the task sheet
   - Read-Write - Read what the participant wrote
   - NC - not classifiable into any of these categories

   This was the first level of coding done and is directly obtained from observation of the video data. Each segment contained the start time, end time, duration of the action and the corresponding action code as shown in Table 4.6. The Umbrello interface places each design diagram in a separate tab, enabling me to observe from the video and screen recording, which diagram a participant's direction of attention was at a particular time (see Figure 4.11). Inferring when the participant is reading and writing was also possible by observing the video data.

2. **Combining segments to form an episode** - I then combined the identified segments to form episodes. Based on segment patterns, there were several instances of switching between two diagrams. I combined these segment patterns and considered them as a single episode. For example - The pattern $\langle UnlockSD, CD, UnlockSD, CD, UnlockSD \rangle$ was considered as a single episode.

3. **Creating inferential codes for each episode** - I did a second level of inferential coding for each episode based on the pattern of the episode. For example - The pattern $\langle UnlockSD, CD, UnlockSD, CD, UnlockSD \rangle$ was assigned the code *"Vertical reading - between requirements and class diagrams"*

4. **Thematic analysis of inferential codes** - The inferential codes were examined, categorized and combined with other codes in order to generate themes. These themes correspond to the reading strategies which students used to evaluate the given software design diagrams.

Reliability of the video data analysis was done during two stages of the analysis - during descriptive coding into segments (step 1) and inferential coding of episodes (step 3). To establish reliability of the descriptive coding of the video data, I provided three clips (totalling 7 minutes) of a particular participant to another rater. I briefed the rater about the research questions of the study, the analysis framework and the descriptive codes which had to be assigned. After this brief, the rater and I independently coded both the clips. After this, I synchronized the rater's and my time and looked for agreement among the assigned descriptive codes. Cohen's Kappa (Cohen, 1960) was calculated to establish inter-rater reliability. The kappa coefficient came to 0.85, which indicated a substantial agreement.

Secondly, reliability of inferential codes and themes of the video data and participant writing was established. Three rounds of discussion between me and the rater was done to come up with the final set of themes and sub-themes. In each round, I gave a set of episodes to the rater. After coding the given set of episodes, the rater and I discussed the resulting codes and themes emerging and came to an agreement. We followed this process for two more rounds till no more themes or sub-themes could be added.

Table 4.6: A snapshot of the descriptive coding of a participant's video data

Time	**Duration**	**Start Time**	**End Time**	**Action**	**Direction of Attention**
5:35 - 7:37	122	0	122	Reads the task sheet	SP
7:37 - 7:45	8	122	130	Reads info sheet page 1	INF
7:45 - 7:50	5	130	135	Reads info sheet page 2	INF
7:58 - 8:05	7	135	142	Reads the task sheet	SP
10:10 - 10:34	24	142	166	Double clicks on Door class and goes through different properties	CD
10:34 - 10:38	4	166	170	Reads the Door class in the class diagram	CD
10:38 - 10:53	17	170	187	Goes through the Controller class data attributes and functions	CD
10:53 - 10:57	4	187	191	Reads the task sheet	SP
11:00 - 11:13	13	191	204	Reads the Passcode class in the class diagram	CD
11:13 - 11:27	14	204	218	Reads the unlock door sequence diagram - messages 1,2 and if	UnlockSD
..	..	..	..	..	..
..	..	..	..	..	..
..	..	..	..	..	..
17:00 - 17:05	5	517	522	Reads the class diagram	CD
17:05 - 17:33	28	522	550	Reads the class diagram	CD
17:36 - 17:40	4	550	554	Reads the task sheet	SP
17:43 - 18:03	20	554	574	Reads the register sequence diagram	RegisterSD
18:03 - 18:55	52	574	626	Writes - Req 1	Write
18:55 - 19:01	6	626	632	Reads what he wrote	Read-Write
19:01 - 20:00	9	632	641	Writes - Req 1	Write
20:00 - 20:43	43	641	684	Reads what he wrote	Read-Write
20:43 - 20:54	11	684	695	Reads the lock door sequence diagram	LockSD
21:00 - 21:18	18	695	713	Writes - Req 2	Write
21:18 - 21:28	10	713	723	Reads the lock door sequence diagram	LockSD
21:28 - 21:37	9	723	732	Reads requirement 2	SP
21:37 - 22:33	56	732	788	Writes - Req 2	Write

Analysis method to infer elements of the mental model (RQ 1.4)

In order to infer the elements of the mental model (RQ 1.4), I coded the transcripts of participants' writing on the task sheet and post-task interview in order to generate themes. I first explain the steps which I followed to analyse the audio transcripts (see Figure 4.12 for a summary)

1. **Transcribing and segmenting the audio data** - I transcribed the entire post-task interview for each participant. The transcription was divided into segments. I chose the unit of analysis as a sentence, and hence each sentence uttered by the participant corresponded to a segment.

2. **Descriptive coding of each segment** - In this stage, I assigned descriptive codes to each segment. The objective of descriptive coding is to avoid any prejudices or preconceptions and stay close to the data. Each segment was assigned a descriptive code which described the participant's articulation of how they solved the task. For example, the open code for *"Now when it's true, what will happen and when its false what will happen."* is *"What will happen when the condition is true and false"*.

3. **Inferential coding** - Based on the descriptive codes, I inferred codes which could explain larger segments of data. The most significant or most frequent codes were used to go through larger amounts of data. In this stage, I also checked if the inferred codes align with the elements of the mental model for design diagrams. For example, the focused code for *"What will happen when the condition is true and false"*, is *"Simulate control flow when the condition is true and false"*

4. **Generating themes** - In this stage, I looked for patterns, explanations and relationships among the inferential codes and generated themes. For example, several codes such as *"control flow of true and false condition"*, *"control flow of the entire sequence diagram"* were grouped together under the *"Control flow"* theme.

The process I followed to analyse the student writings on the task sheet was similar to the analysis of the audio transcripts. I transcribed the written responses, and divided the transcript into sentence-level segments. I followed a similar process of descriptive and inferential coding, and generated themes. Hence, the themes which emerged based on the analysis of participants' writing and post-task interview was used to infer students' mental model elements.

To establish reliability of the codes and themes, I followed an approach similar to the inferential coding of the video data. I explained the analysis framework to another rater and presented a set of sentences from participant transcripts. The rater then independently coded these sentences and came up with descriptive and inferential codes. After this, both the rater and I discussed, reviewed and refined the themes. After two rounds of discussion, we came to a consensus on the final themes.

4.3.7 Findings

We answer RQ 1.2 by examining students' response sheets in order to examine which defects they were able to identify. We answer RQ 1.3 by describing the strategies students used (inferred from the video data) and RQ 1.4 by describing the characteristics of their mental model (inferred from written responses and post-task interview) for the evaluation task.

RQ 1.2 - Defects identified by students in the evaluation task

We identified the defects which students uncovered by analysing their task sheets. Table 4.7 provides a summary of the defects found by participants. As seen from the table, all participants were able to detect defect D6 and D7 (both involved checking for the 'closed' condition). However, only P4 was able to defect D3 and D5 (checking for the initial state of the 'door' variable). None of the participants were able to uncover defects D2 and D4 (incorrect behaviour when wrong passcode is entered). P4 uncovered 4 defects, P2 and P3 - 3 defects, P1, P5 and P6 - 2 defects.

Based on the analysis, certain patterns emerged. All participants were able to uncover defects D6 and D7. Both defects involved the omission of the 'closed' construct. This required participants to search the different design diagrams, and check if the "closed" construct was available. *Hence, we can infer that all participants were able to uncover defects involving omission of missing constructs.*

Participants P2 and P3 were able to uncover defect D1. Defect D1 can be uncovered by imagining a scenario where the user is already registered. Only P4 was able to uncover defects D3 and D5. In order uncover D3 and D5, participants had to simulate the initial state of the "lock" variable.

We also see that none of the participants were able to uncover defects D2 and D4. In the lock (Figure B.4) and unlock (Figure B.5) sequence diagrams, no function should be called when an incorrect passcode is entered (corresponding to the else part) i.e. the door should remain in the same state. However, calling the unlock function (in the 'else' part of the lock sequence diagram) and the lock function (in the 'else' part of the unlock sequence diagram) causes a change in state and thus does not satisfy the requirements. In both these defects, participants were required to simulate the change of state in the lock variable when the lock() or unlock() function is called. This data flow simulation might have led them to uncover these defects. *Hence, we can infer that students had difficulty in uncovering defects which involve simulation of control and data flow in the design.*

RQ 1.3 - Student Strategies for Evaluating Design Diagrams

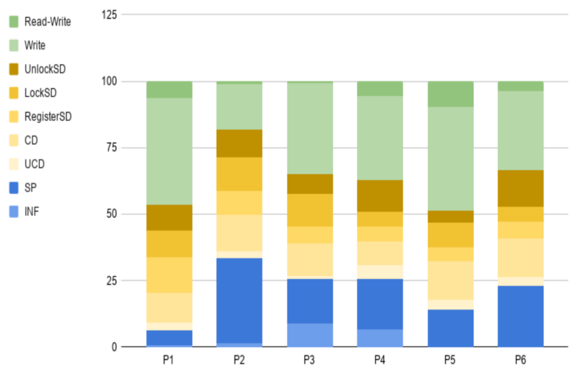
For each participant, we aggregated the time spent for each code corresponding to their direction of attention. Figure 4.13a denotes percentage of time spent for each direction of attention for each participant based on video data. In Figure 4.13b, we group the codes into 3 categories - Problem space (SP and INF), Design diagram space (CD, UCD, RegisterSD, LockSD, UnlockSD) and Writing (Write, Read-Write). From the figures, we see that there were no major differences in how participants spent their time during the evaluation task. P2, P3, P4 and P6 spent considerable time on the problem statement (INF and SP) compared to P1 and P5. P1 and P2 spent the most time on observing the design diagrams (CD, UCD, RegisterSD, LockSD, UnlockSD).

In order to infer high-level strategies used by participants, we examined episodes where they performed round-trips between different 'direction of attention' codes.

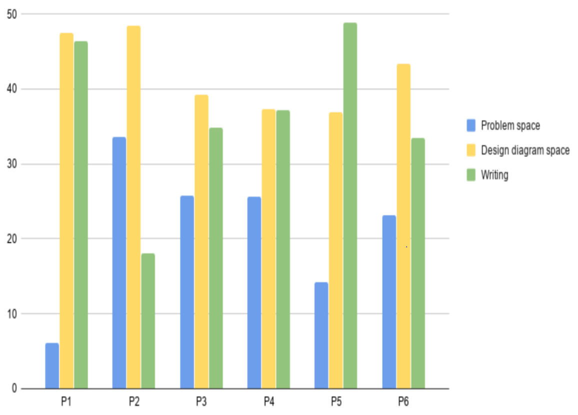
We found that students primarily used three strategies - (1) Horizontal reading (2) Vertical reading and (3) Concurrent use of horizontal and vertical reading.

Table 4.7: Summary of defects detected by participants based on their task sheet writings

Requirement	Defect	P1	P2	P3	P4	P5	P6
R1	D1 - Register Sequence Diagram - There is no check if the passcode has already been set	N	Y	Y	N	N	N
R2	D2 - Lock Sequence Diagram - When incorrect passcode is entered, the door should not unlock but remain in the same state	N	N	N	N	N	N
	D3 - Lock Sequence Diagram - Initial state of door is not specified (i.e. the door should be in the unlocked state)	N	N	N	Y	N	N
R3	D4 - Unlock Sequence Diagram - When incorrect passcode is entered, the door should not lock but remain in the same state	N	N	N	N	N	N
	D5 - Unlock Sequence Diagram - Initial state of door is not specified (i.e. the door should be in the locked state)	N	N	N	Y	N	N
R4	D6 - Lock Sequence Diagram - There is no condition to check if the door is closed	Y	Y	Y	Y	Y	Y
	D7 - Unlock Sequence Diagram - There is no condition to check if the door is closed	Y	Y	Y	Y	Y	Y



(a) Time spent for each ‘direction of attention’



(b) Grouping of ‘direction of attention’ into categories

Figure 4.13: Summary of the video data analysis for each participant

1. **Horizontal reading**- In horizontal reading, participants read different design diagrams in order to integrate information from diagrams. All episodes which had one or several round-trips between the class diagram and a sequence diagram were included in this theme. (Example - $\langle CD, RegisterSD, CD\rangle$, $\langle UnlockSD, CD, UnlockSD, CD\rangle$)

2. **Vertical reading** - In vertical reading, participants read between design diagrams and the system requirements. All sub-episodes which had one or several round-trips between the a specific design diagram and reading the requirements were included in this theme. (Example - $\langle SP, LockSD, SP\rangle$, $\langle UnlockSD, SP, UnlockSD, SP\rangle$)

3. **Horizontal and vertical reading** - In this pattern, students read requirements, class diagram and multiple sequence diagrams in no particular order.

Figure 4.14 shows the strategies which different participants used. Most participants used vertical reading, whereas P1 and P2 used horizontal reading techniques as well. In spite of using these reading strategies, students could not identify defects in the design, especially those that involved data flow simulation. Table 4.8 shows the percentage of the total time each participant spent in such multiple round-trips. We see that there wasn’t any clear relationship between the time spent in multiple round-trips and the defects participants were able to uncover. *Hence, we can infer that students’ ability to identify defects did not depend on the reading strategies they used.*

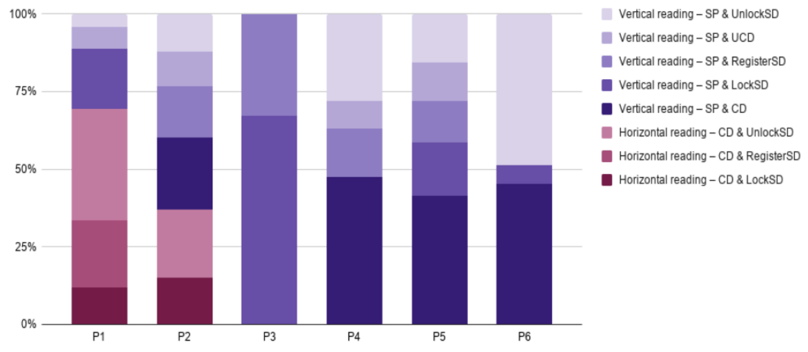


Figure 4.14: Percentage of time spent for each reading strategy based on video data

Table 4.8: Relationship between time spent and defects identified for each participant

Participant	Total Time Spent in Task (Minutes)	Time spent in switching (Minutes)	Percentage Time spent in switching	Defects Identified
P1	31.95	6.35	19.87	2
P2	16.53	4.97	30.04	3
P3	21.73	4.5	20.71	3
P4	14.23	5.05	35.48	4
P5	16.12	5.92	36.71	2
P6	11.02	3.42	31.01	2

Although we did not see differences in reading strategies among participants, we observed certain differences in their explanation of how they went about the evaluation task. We explain these differences in students' mental model below.

RQ 1.4 - Students' Mental Model

We inferred students' mental model elements for design diagrams based on the themes which emerged from the inferential coding of the participants' writing and post-task interview data. A summary of the themes and sub-themes from the post-task interview are described in Figure 4.15, and from the written responses in Figure 4.16. We combined themes from both these analyses and present the mental model elements below.

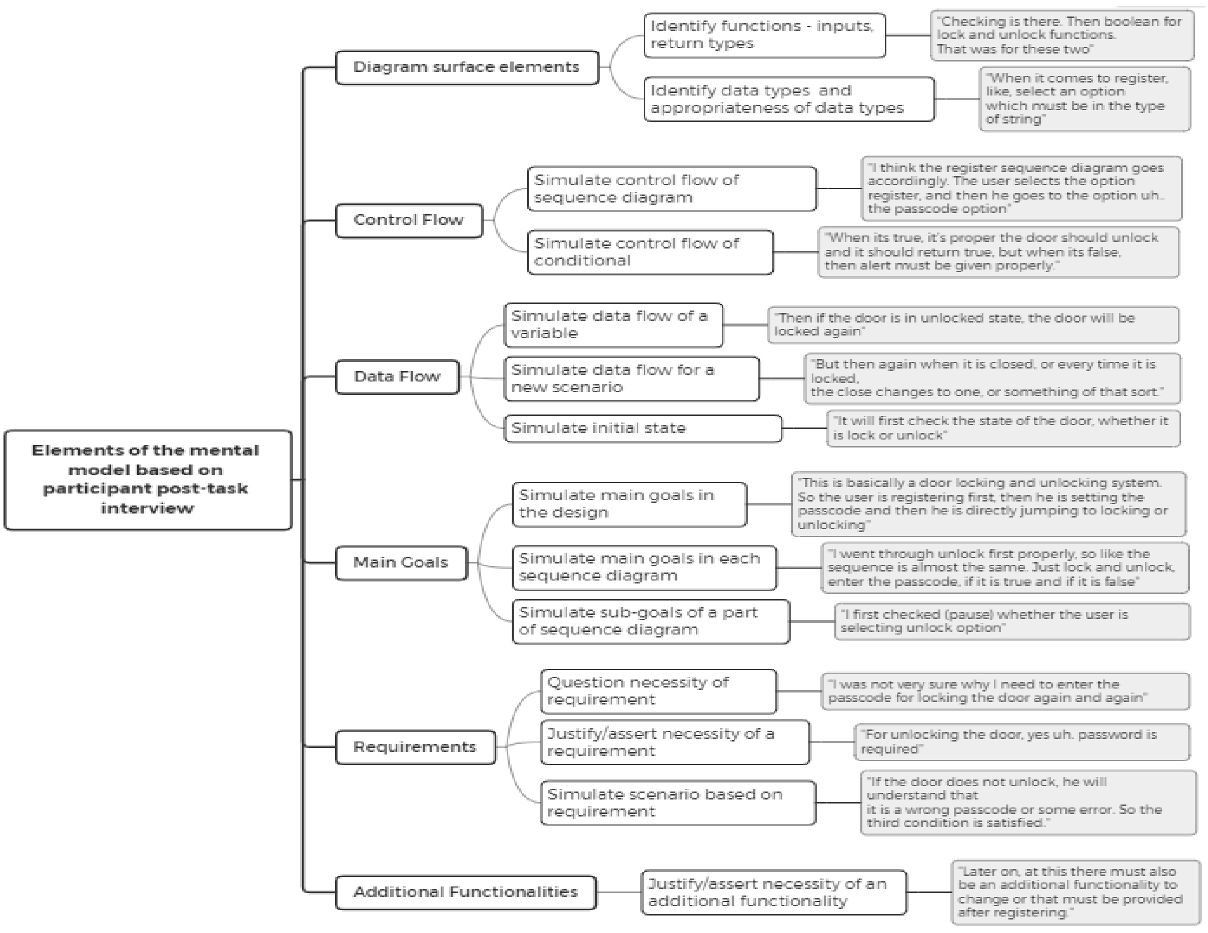

Elements of the mental model based on participant post-task interview
Diagram surface elements
Identify functions - inputs, return types
"Checking is there. Then boolean for lock and unlock functions. That was for these two"
Identify data types and appropriateness of data types
"When it comes to register, like, select an option which must be in the type of string"
Control Flow
Simulate control flow of sequence diagram
"I think the register sequence diagram goes accordingly. The user selects the option register, and then he goes to the option uh.. the passcode option"
Simulate control flow of conditional
"When its true, it's proper the door should unlock and it should return true, but when its false, then alert must be given properly."
Data Flow
Simulate data flow of a variable
"Then if the door is in unlocked state, the door will be locked again"
Simulate data flow for a new scenario
"But then again when it is closed, or every time it is locked, the close changes to one, or something of that sort."
Simulate initial state
"It will first check the state of the door, whether it is lock or unlock"
Main Goals
Simulate main goals in the design
"This is basically a door locking and unlocking system. So the user is registering first, then he is setting the passcode and then he is directly jumping to locking or unlocking"
Simulate main goals in each sequence diagram
"I went through unlock first properly, so like the sequence is almost the same. Just lock and unlock, enter the passcode, if it is true and if it is false"
Simulate sub-goals of a part of sequence diagram
"I first checked (pause) whether the user is selecting unlock option"
Requirements
Question necessity of requirement
"I was not very sure why I need to enter the passcode for locking the door again and again"
Justify/assert necessity of a requirement
"For unlocking the door, yes uh. password is required"
Simulate scenario based on requirement
"If the door does not unlock, he will understand that it is a wrong passcode or some error. So the third condition is satisfied."
Additional Functionalities
Justify/assert necessity of an additional functionality
"Later on, at this there must also be an additional functionality to change or that must be provided after registering."

Figure 4.15: Mental model elements themes emerging from participants' post-task interview

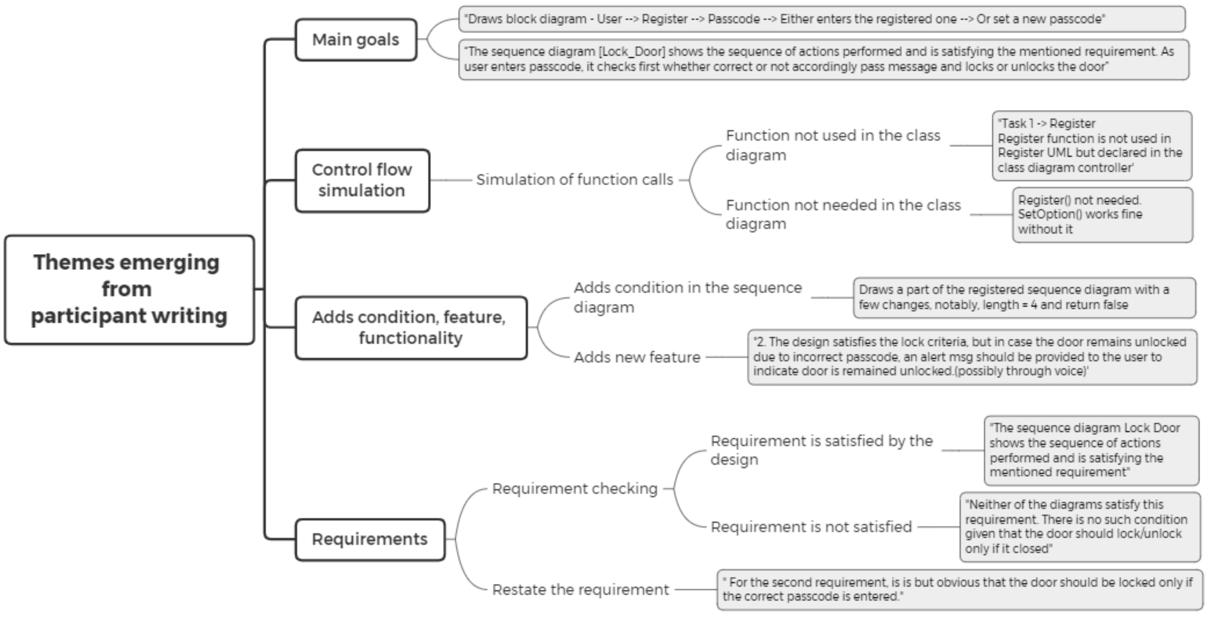

Themes emerging from participant writing
Main goals
"Draws block diagram - User --> Register --> Passcode --> Either enters the registered one --> Or set a new passcode"
"The sequence diagram [Lock_Door] shows the sequence of actions performed and is satisfying the mentioned requirement. As user enters passcode, it checks first whether correct or not accordingly pass message and locks or unlocks the door"
Control flow simulation
Simulation of function calls
Function not used in the class diagram
"Task 1 -> Register Register function is not used in Register UML but declared in the class diagram controller'
Function not needed in the class diagram
Register() not needed. SetOption() works fine without it
Adds condition, feature, functionality
Adds condition in the sequence diagram
Draws a part of the registered sequence diagram with a few changes, notably, length = 4 and return false
Adds new feature
"2. The design satisfies the lock criteria, but in case the door remains unlocked due to incorrect passcode, an alert msg should be provided to the user to indicate door is remained unlocked.(possibly through voice)'
Requirements
Requirement checking
Requirement is satisfied by the design
"The sequence diagram Lock Door shows the sequence of actions performed and is satisfying the mentioned requirement"
Requirement is not satisfied
"Neither of the diagrams satisfy this requirement. There is no such condition given that the door should lock/unlock only if it closed"
Restate the requirement
" For the second requirement, is is but obvious that the door should be locked only if the correct passcode is entered."

Figure 4.16: Mental model elements themes emerging from participants' written responses

1. **Diagram surface elements** - These elements form the basic messages/functions in the sequence diagram and class diagram. They are the basic building blocks of the diagrams. Participants said that they observed data types, access specifiers and methods of the class diagram. They also checked the appropriateness of particular data types. They also checked the appropriateness of the return type of functions in the class and sequence diagrams. (e.g.: *"So when this function is passed over here, it will check that passcode and return a boolean value whether the passcode is correct or not. If it is correct or if it is not, it will return it over here. So, it will return true or false."*)

2. **Control flow -** We defined control flow as the control structure linking individual messages in the sequence diagram. We identified control flow codes in their writing and speaking when they explicitly simulated function calls or specified the flow of messages in the sequence diagram. We observed that participants (a) Simulated function calls by articulating about functions not used or not needed (e.g. *"Register() not needed. SetOption() works fine without it"*) (b) articulated the flow of messages in different sequence diagrams. They also (c) specified the control flow of conditionals (e.g.: *"If length is 4, it will call setpasscode upon itself and then, it will return"*). They were also (d) able to simulate the control flow of true and false conditions in the conditional.

3. **Data flow -** We identified codes corresponding to data flow when participants explicitly mentioned simulating the flow of data across the sequence diagram. (e.g.: *"The passcode gets the bool value, and returns the bool. "*). We found no instances of data flow simulation in their written responses, and very few instances of participants (only P2 and P4) describing the data flow when asked how they performed the evaluation task.

4. **Main goals -** Main goals refer to the plans in the design at a higher level of granularity. We identified codes corresponding to main goals when participants abstracted the functionality of the design diagrams and did not write/speak in terms of flow of messages and change in values. We observed at all participants were able to identify the main goals of the design diagrams.

   - Participants considered the main goals of the design when they first encountered the problem (Example: *"This is basically a door locking and unlocking system. So the user is registering first, then he is setting the passcode and then he is directly jumping to locking or unlocking."*).

- Having understood the main goals of the design diagrams, they identified the main goals of each sequence diagram (Example: *"So the first thing that a person does is, he has to select an option."*). Participants identified goals present in the requirement and mapped it to goals in the sequence diagram. (e.g.: *"So this (requirement) has three steps, lock, password and the final function calling. So even this (register sequence diagram) can be divided into that."*).
- They then identify the goal of each part of the sequence diagram (e.g: *"Selecting to lock, so this part does that. Till here it is checking whether I have selected to lock or not."*) and checked with the corresponding goal of the requirement.

5. **Requirements** - Participants were introduced to the requirements as well as the design diagrams. Hence, reading the requirements triggered them to form a mental model of the situation based on the requirements. In participant writings, they specified whether the requirement satisfied the design or not, and even restated the requirement (see Figure 4.16). In interviews, we saw that participants questioned the necessity of a requirement (Example: *" I was not very sure why I need to enter the passcode for locking the door again and again"*) as well as justified the necessity of a requirement. (e.g.: *"For unlocking the door, yes uh. password is required"*). Participants also simulated scenarios in order to check whether a requirement is satisfied or not (e.g.: *"If the door does not unlock, user will understand that it is a wrong passcode. So the third condition is satisfied"*)

6. **Additional Functionalities** - While analysing the requirements and the design diagrams, participants also thought of additional functionalities which were lacking in the design and also justified its necessity in the design (e.g.: *" But in this whole thing, there is no validation involved. For example, a passcode needs to be four digits or four numbers,"*).

We observed that the mental model of the design created by students while evaluating design diagrams, led them to identify defects in the design, especially those that involved data flow simulation. For example, P4 was the only participant who uncovered defects D3 and D5 (checking for the initial state of 'door'). P4 showed instances of describing data flow when asked about how he went about checking the requirements. For requirement 2 and 3, he said that he checked the initial state of the lock variable while solving for the lock and unlock requirement (*"It will first check the state of the door, whether it is locked or unlocked"*). P4 simulated the initial state of the “lock” variable and said that he “imagined” the initial state of the door, which led him to uncover these defects. Other participants did not show instances of data flow simulation and hence were not able to uncover defects D3 and D5. All participants were able to identify the main goals of the design and the sequence diagrams.

Most students mentioned that they visualized scenarios in the design. However, some students visualized completely new scenarios which were absent in the requirements and design diagrams (For example: adding a voice functionality, checking the length of passcode)

4.3.8 Reflections from Study 1b

As seen from the findings, all participants used reading techniques of switching between multiple diagrams and the requirements. However, there was no relationship between these strategies and the defects students identified in the design. *The findings show that the elements present in students’ mental model play a significant role when used appropriately with the reading strategies in their ability to identify defects in the design diagrams*

Our findings show that most students restricted their focus to the diagram surface elements, main goals and superficial control flow of the design diagrams. For example, all students were able to uncover defects D6 and D7, which involved an omission of the ‘closed’ construct. Students were able to do this, as it involved searching the design diagrams to check for the ‘closed’ construct. *Hence, we can infer that students are able to uncover defects, which involve a superficial search over the design diagrams.*

We also see that none of the participants were able to uncover defects D2 and D4, which involved a simulation of the control flow and the data flow for the lock and unlock sequence diagrams. We see a similar pattern for defects D3 and D5 (only P4 could uncover these defects), which involved simulating the initial state of the variable “lock”. *Hence, we can infer that participants have difficulty in uncovering defects involving simulation of data flow.*

These findings are consistent with results from program comprehension literature as well. Pennington makes a distinction between “surface knowledge” and “deep 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" (Pennington, 1987). Students had sufficient surface knowledge about the structure of the class and sequence diagrams to help them identify missing constructs such as "closed", but they could not identify defects which involved simulating the data flow of variables across function calls in the sequence diagram.

Some participants focused on simulating completely new scenarios (voice assistance etc.), and introduced new functionalities in the design, as opposed to verifying the design against the given requirements. This also confirms findings from Study 1a and also gives us insights as to why they would have exhibited such a behaviour. Based on their existing problem domain knowledge (of an automated door locking system), students internalised these requirements and tried to reason and justify the need of these requirements. They also mapped these requirements to the design diagrams using their design diagram knowledge. If something is missing or incorrect in the design diagrams, student flagged these as defects and explicitly mentioned that these additional functionalities are required.

Students adding new functionalities also confirms certain characteristics of mental models, that models of novices lack firm boundaries, and they may be unclear about what aspects or parts of the system their model is supposed to cover (Norman, 2014). In this study, students visualised scenarios outside the model boundary of the design diagram and requirements, which led them to include additional functionalities into their mental model of the design.

To summarize, the above findings indicate what elements are present or inadequate in students' mental models as they evaluate design diagrams. This is summarized in Figure 4.17. In addition to the diagram surface elements, dynamic behaviours, and main goals which we adapted from the Block model (Figure 4.2), we see that students also develop a mental model of the problem domain, by thinking about the requirements, and their problem domain knowledge leads them to incorrectly add new functionalities as well.

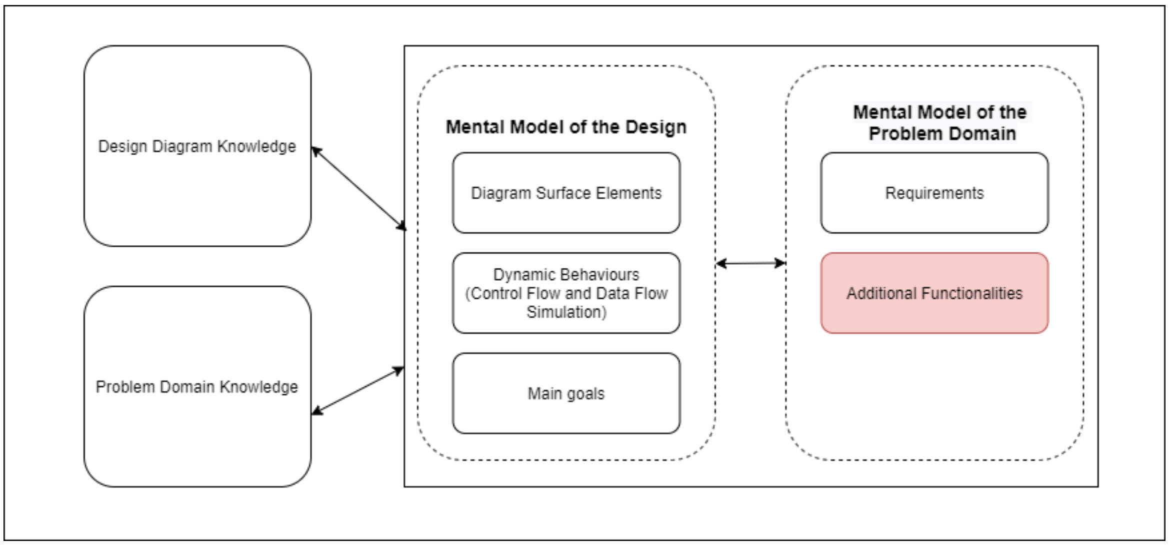

Design Diagram Knowledge
Problem Domain Knowledge
Mental Model of the Design
Diagram Surface Elements
Dynamic Behaviours (Control Flow and Data Flow Simulation)
Main goals
Mental Model of the Problem Domain
Requirements
Additional Functionalities

Figure 4.17: Mental model elements for design diagrams based on findings from Study 1b

4.4 Summary of Insights from Novice Studies

Study 1a and 1b provides insights on how students approach a given evaluation task. These studies also show that the adapted mental model for design diagrams is appropriate for specifying the elements of a mental model for doing design evaluation.

These studies revealed categories of student responses for an evaluation task, and how these categories relate to their mental models. We found that students focus their attention on *superficial aspects of the design* and have *difficulty in simulating the control and data flow in designs*. Some students *add new functionalities and requirements into the design, rather than evaluating the design against the given requirements.*

These insights inform the design of our pedagogy for enabling students to effectively perform software design evaluation. The pedagogy needs to scaffold students to

- Build appropriate mental models of the problem domain and the design.
- Simulate the control flow and data flow in order to verify if the design satisfies the requirements.
- Understand the purpose of design evaluation i.e., they are required to evaluate the design against the given requirements rather than add new functionalities into the design.

In the next chapter, we describe how we have incorporated these insights into the design of the pedagogy for software design evaluation.

Chapter 5

Design of the VeriSIM Learning Environment

5.1 Summary of findings from literature and novice studies

Our findings based on the analysis of literature and results from novice studies show that:

- Experts spend a significant time evaluating their designs. They create a rich mental model of the design and perform mental simulations on these models.
- Experts simulate the control flow and data flow of various scenarios in the design in order to evaluate the design against the given requirements.
- Novices are able to do a superficial search on the design diagrams, but have difficulty in simulating the control flow and data flow within design diagrams.
- Novices have difficulty in identifying and simulating scenarios where the design does not satisfy the requirement. They identify scenarios outside the model boundary of the design

diagram and requirements, which lead them to include additional functionalities into the design.

Thus, we can infer that effective evaluation of the design diagrams depends on (1) the mental model which students create based on the requirements and the design and (2) students' ability to simulate relevant scenarios in their own mental model of the design. Students develop a better understanding of the design when their mental models contain information about the control flow and data flow of various scenarios in the design.

These inferences form the basis of the VeriSIM (**Veri**fying designs by **SIM**ulating scenarios) pedagogy, which we propose for the teaching-learning of software design evaluation. The key idea of the VeriSIM pedagogy is that - *scaffolding students to identify and construct models of relevant scenarios in the design can lead to effective evaluation of the design diagrams against the given requirements.*

How can we support students to identify and construct models of relevant scenarios? To answer this, we draw the theoretical basis of VeriSIM from science education and introductory programming literature, which have emphasised teaching and learning of modelling. We adapt effective affordances and pedagogical features from these sources to inform the design of the VeriSIM pedagogy.

5.2 Theoretical Basis of the VeriSIM Pedagogy

Software design involves modelling, where the design represents an abstract model of the software system, on which mental simulations are performed to evaluate the design. However, there is a lack of pedagogical techniques which emphasise on modelling software designs (Burgueño et al., 2018). On the other hand, science education has placed a lot of importance to modelling, and extensive research has been done on incorporating modelling resources and teaching-learning strategies in the classroom. Learning of introductory programming has also stressed on enabling students to build accurate mental models of the program as the primary way to improve their programming performance (Sorva, 2013).

In this section, we draw insights from science education, and use of mental models in introductory programming to inform the basis of our pedagogy. We adapt pedagogical features and affordances which have been shown to be beneficial, to the context of software design.

5.2.1 Model-based Learning in Science Education

Modelling is a central practice in the discipline of science (Papaevripidou and Zacharia, 2015). When scientists observe a phenomena, they construct models which try to explain the process at a sufficient level of abstraction. Schwarz et al. defines models "as a representation that abstracts and simplifies a system by focusing on key features to explain and predict scientific phenomena" (Schwarz et al., 2009). The model serves as a portrayal of a scientist's current understanding of the phenomena. The model is then tested by observations in the real world and refined based on these observations (Hestenes, 2010). Examples of models such as Bohr's model of the atom, the double helix model of DNA, atmospheric models etc. have helped scientists reason about various microscopic as well as macroscopic phenomena.

Since the modelling process is an important practice of scientists, it is essential that students are explicitly taught this process. Scientists have emphasised the importance of integrating the modelling process in the teaching-learning of science (Hestenes, 1987, 1992). An important goal of science education is to enable students to develop powerful models for making sense of their daily experiences of biological, physical and chemical phenomena (Clement, 2000). A prominent paradigm towards this goal is model-based learning. Simply put, model-based learning involves using modelling for learning purposes. It is defined as "*a dynamic, recursive process of learning by constructing mental models of the phenomenon under study. It involves the formation, testing, and subsequent reinforcement, revision, or rejection of those mental models*" (Buckley et al., 2010). Model-based learning provides students opportunities to interact with physical, representational or computer models which describes the working of a scientific phenomenon. As students engage in constructing scientific models, they mimic modelling practices of scientists and also develop conceptual knowledge of the phenomena.

Knowledge and Practices in Model-based Learning

Modelling a phenomenon requires knowledge of specific modelling practices (Papaevripidou and Zacharia, 2015). According to Louca et al., constructing a model of a phenomenon/system, requires students to identify components that comprise that phenomenon or system, namely its *objects* (e.g. a ball, a molecule), *processes* (e.g., movement of objects), *entities* (e.g. velocity) and *interactions* (e.g. how entities interact with objects or processes) (Louca and Zacharia, 2015).

In addition to construction of models, model-based learning also entails refining, revising, evaluating and validating scientific models. Learning to model a phenomenon involves investigating it, collecting evidence, and bringing one's observations and experiences from the physical world. These conceptions and domain-specific knowledge result in the construction of an initial working model *(model formulation)*. The initial model is modified by restructuring and tuning various elements in the model. During model revisions, several intermediate models are generated. Students go back and forth between the intermediate models, and compare it with the phenomenon or system *(model comparison)*. These comparisons are done by doing mental simulation of these models and validating whether it predicts the corresponding phenomenon accurately *(model evaluation)*. This step-wise enrichment of mental models is referred to as the "fleshing out" of mental models (Johnson-Laird, 1983). *Hence, model formulation, model comparison, and model evaluation are essential practices required during model-based learning* (Papaevripidou and Zacharia, 2015). In addition to knowledge about modelling practices, metacognitive knowledge and epistemic knowledge about modelling is also essential. Metacognitive knowledge refers to knowledge about the steps of the scientific modelling process (Papaevripidou and Zacharia, 2015). Epistemic knowledge corresponds to understanding the nature, purpose and utility of scientific models (Schwarz and White, 2005).

To summarize, model-based learning entails knowledge of the components that comprise a system, essential modelling practices which involve constructing, revising, and evaluating intermediate models, and metacognitive and epistemic knowledge. These elements have been summarised in Figure 5.1.

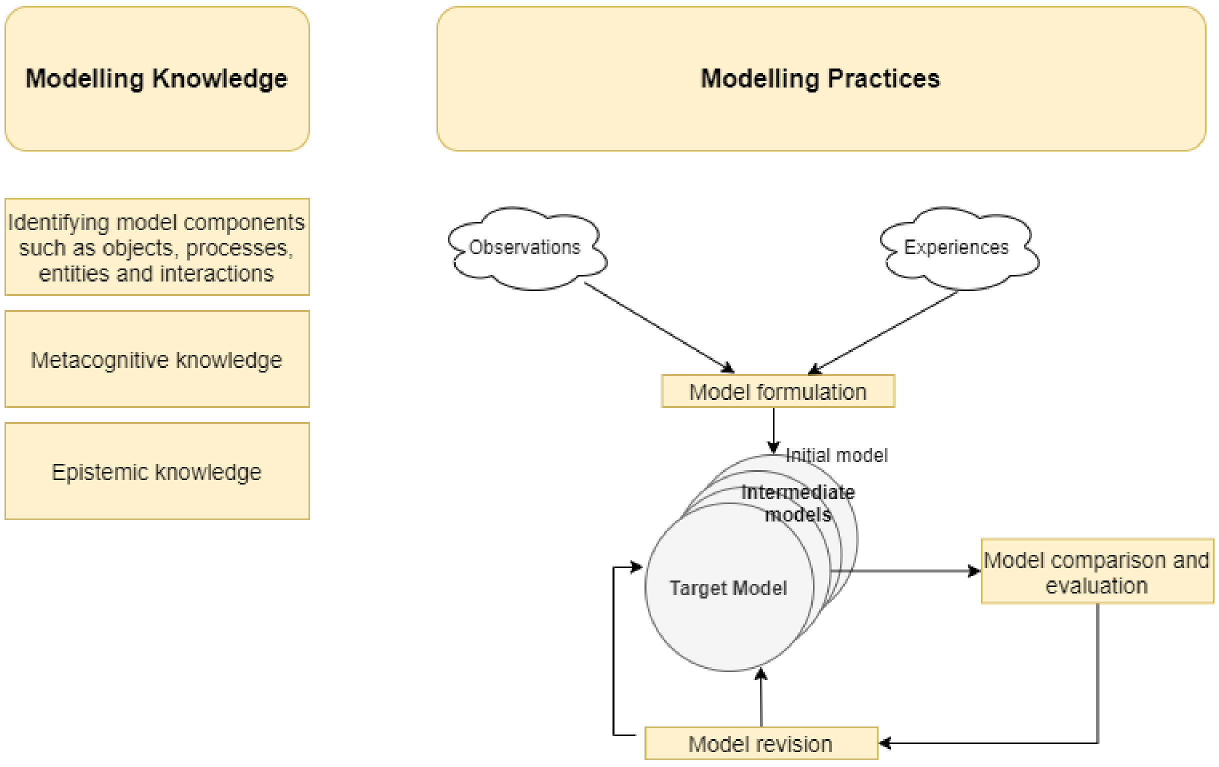


Figure 5.1: Knowledge and practices in model-based learning (adapted from Buckley et al. (2010))

Scaffolds and Technology Support for Model-based Learning

Research has shown the benefits of scaffolding in supporting students' modelling processes (Fretz et al., 2002). Students provided with scaffolds as they interact with models have shown to perform better than those who were not provided any scaffolds (Seel and Dinter, 1995; Sandoval and Reiser, 2004; Azevedo et al., 2011; Buckley et al., 2010). Scaffolds have been implemented as prompts, questions, hints, conceptual models and visualizations, in order to assist students as they progress through modelling tasks (Seel, 2017). Among these scaffolds, there has been a significant emphasis on the advantages of presenting students with visualizations (diagrams, animations, simulations) during modelling tasks. It has been shown that by using visualizations, students are more likely to construct mental models with which they can reason about various scientific issues (Rapp, 2005). There are also empirical studies which demonstrate the effectiveness of visualizations on model-based learning (e.g.: (Davies et al., 2005; Sharp et al., 1995)).

Computer simulations have also been widely used to help students develop mental models of a given phenomena. Simulations provide students with affordances to imitate the processes of complex systems, and provide opportunities to visualize these processes. Simulations facilitate model-based learning by allowing students to manipulate models by altering values of variables and observing their effects (Sokolowski, 2009; Milrad et al., 2003). However, an analysis of literature shows that there needs to be a proper balance between providing necessary guidance and scaffolds, and allowing students to use computer simulations or create simulation programs (Seel, 2017). Students can get overwhelmed by the complexities of the modelling task and fail to develop adequate mental models of the phenomena (Adams et al., 2008).

To scaffold students in modelling, *model progression* has shown to be beneficial (Mulder et al., 2011). In model progression, students are provided models which vary in dimensions such as their perspective, degree of elaboration and their order (White and Frederiksen, 1990). When students are provided opportunities for successive refinements of their model, it helps them progressively understand and develop models of the given phenomenon. Various strategies support this idea of model progression. These include strategies such as *prior exploration* (Kopainsky et al., 2015; Kopainsky and Alessi, 2015), *partially worked-out models* (Mulder et al., 2016) and *learning from erroneous models* (Wijnen et al., 2015). In the *prior exploration strategy*, Kopainsky and Alessi (2015) allowed learners to interact with a simulation model (by changing values of relevant variables) and see the results in practice mode. Students observed the structure and behaviour of the simulation as a result of these interactions. Students modelled the phenomenon in different phases. They started with manipulating a single variable, followed by manipulating an additional variable, and continued till they modelled the entire phenomenon. Findings from this study showed that the prior exploration strategy led to an improved understanding of the underlying model. In *'partially worked-out models'*, learners received support in the form of a partial model which outlined the basic structure of a system (Mulder et al., 2016). Findings from this study showed that the quality of these learners' models were better than those who did not receive support at all. In the *'learning from erroneous models'* strategy, learners worked with a model which contained errors (Wijnen et al., 2015). Learners had to correct the model for the simulation to generate correct outcomes. The findings show that the erroneous models enhanced the acquisition of domain-specific knowledge.

To summarize, model-based learning has been widely adopted as a paradigm for teaching-learning of modelling. It entails learning about the knowledge of the components that comprise a system, and essential modelling practices such as constructing, revising, and evaluating intermediate models. Learners also need metacognitive and epistemic knowledge to abstract what they learnt and apply it to new contexts. We also described various types of scaffolds and strategies which have been used to build modelling competence in students. In the next sub-section, we argue for the analogous nature between the modelling activities in science and software design, and show that the model-based learning paradigm can be adapted for teaching-learning of software design as well.

5.2.2 Adaptation of the model based learning paradigm to software design

The term ‘model’ has been used in science education to refer to physical or biological systems. We have seen that learning of scientific modelling requires students to have knowledge of components (objects, processes and entities) of the system and interactions between these components. Analogously, software design models also comprises components, such as objects, variables and methods. These components interact with each other to describe the system behaviour. For example, an object’s member function can call a function of another object, resulting in changes in variable values in the system. Students require knowledge of these components and how they interact with each other to understand the functioning of a software system. As students develop their understanding of the components and their interactions, they are able to effectively model a software system.

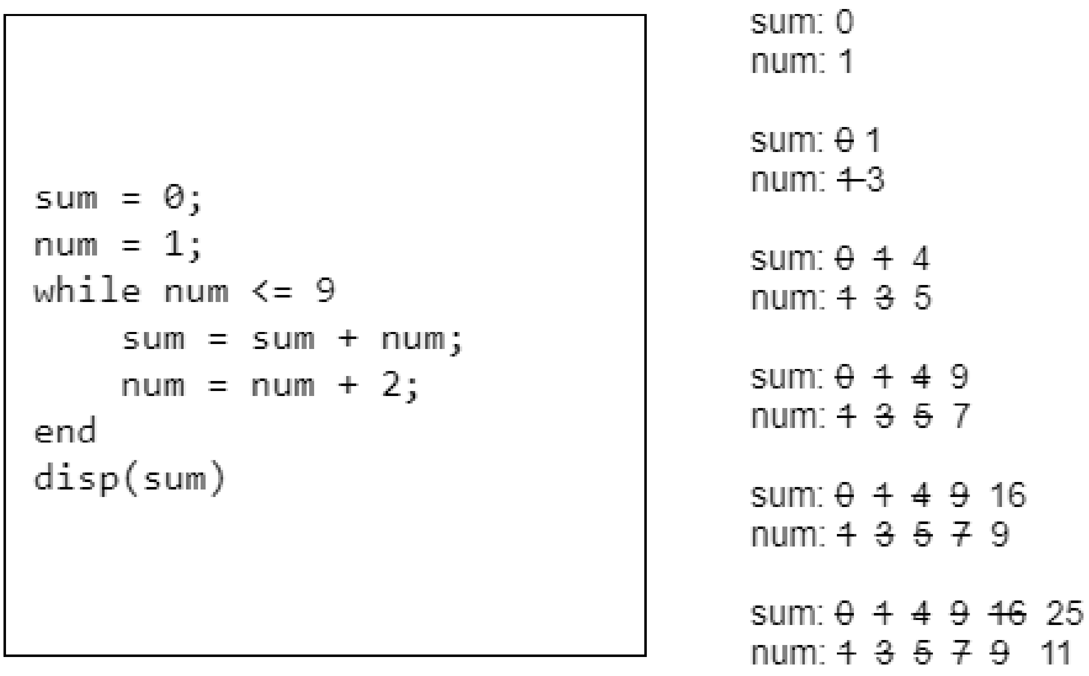
The practices involved in modelling scientific phenomenon can be adapted to software design as well. A common characteristic of modelling activities is that these models are constructed, refined and evaluated to describe the phenomenon under observation. The central tenet of model-based learning is that the process of constructing and manipulating these mental models causes a deeper and integrated understanding of scientific concepts required to understand the phenomenon. Our analysis of literature regarding expert strategies and cognitive processes in software design (Section 2.3), has shown that similar mental modelling processes occur while modelling software designs as well. Experts create a mental model of the system to be designed *(model formulation)*, refine the model by comparing it with their previous models or other information *(model comparison and evaluation)*, and validate their models either by performing mental simulations on them or by prototyping a solution *(model validation)*.

These reasons give us indicators that the model-based learning paradigm can serve as the basis for teaching-learning of evaluating a given software design. Thus, the scaffolds and strategies used in model-based learning, such as model progression can be adapted to the software design context as well. We explain these adaptations in detail in Section 5.3. Having shown the applicability of the model-based learning paradigm as the broad theoretical basis of the pedagogy, we describe the actual strategy which can facilitate students to construct mental models of the design. To address this, we adapt the program tracing strategy from introductory programming literature, and show its applicability in the context of software design evaluation.

5.2.3 Mental models and use of tracing in introductory programming

While programming, programmers simulate the execution of the mental model they have created. These simulations of program execution are often referred to as tracing. Program tracing is the process of emulating how a computer executes a program (Fitzgerald et al., 2005). It involves writing the state change of variables after each line of code. For example, consider the following program shown in Figure 5.2. To trace the following program, one starts from the first lines of the program and identifies the variables in the program and their initial values($sum = 0$ and $num = 1$). One then mentally simulates the execution of the while loop, and the change in the values of variables corresponding to its execution. For example, in the first iteration, the while condition is satisfied and control moves inside the while loop. The values of *sum* and *num* are updated based on a mental simulation of the lines inside the while loop. One then continues these updates to *num* and *sum*, till the condition of the while loop is not satisfied, resulting in the end of program execution.

Tracing a given program enables us to simulate the behaviour of the program, and also understand its purpose. For example, from Figure 5.2, we see from the trace that the *num* variable takes odd number values, and the *sum* variable calculates the sum of these odd numbers. Hence, we can infer that the purpose of this program is to compute the sum of the first five odd numbers.



```
sum = 0;
num = 1;
while num <= 9
    sum = sum + num;
    num = num + 2;
end
disp(sum)
```

sum: 0
num: 1

sum: ~~0~~ 1
num: ~~1~~3

sum: ~~0~~ ~~1~~ 4
num: ~~1~~ ~~3~~ 5

sum: ~~0~~ ~~1~~ ~~4~~ 9
num: ~~1~~ ~~3~~ ~~5~~ 7

sum: ~~0~~ ~~1~~ ~~4~~ ~~9~~ 16
num: ~~1~~ ~~3~~ ~~5~~ ~~7~~ 9

sum: ~~0~~ ~~1~~ ~~4~~ ~~9~~ ~~16~~ 25
num: ~~1~~ ~~3~~ ~~5~~ ~~7~~ ~~9~~ 11

Figure 5.2: A program tracing example

Evidence from literature shows that tracing is a necessary pre-cursor skill to write, maintain and debug code. Previous research has shown that students who use tracing are able to better comprehend a given program (Xie et al., 2018; Cunningham et al., 2017; Lister et al., 2004). Cunningham et al. explored the relation between what students sketched and their ability to comprehend simple python programs. They found that students who used tracing performed better than students who did not trace at all (Cunningham et al., 2017). Xie et al. explicitly taught this tracing strategy to learners and found that program comprehension improved (Xie et al., 2018).

Tracing a program has several advantages. The state change of variables are written down in the form of memory tables or some other external representation. These representations help students manage the cognitive load of remembering values of variables during execution, especially if the program is large or complex. The external representation also serves as an artifact which exhibits students' understanding of the program. As students trace, they take an active role in describing the 'run' of the program, as opposed to reading through the code (Cunningham et al., 2017). Tracing also forces students to think what happens next in the program.

Developers use tracing while comprehending large programs as well. While tracing large programs, they focus their attention on areas of interest in the program, based on the context. For example, if a particular functionality has to be changed, developers read and locate the code where the change has to be performed (Roehm et al., 2012). As they trace specific parts of the program, they reason about the control flow (which components call which functions, how a function is reached, etc.) and data flow (how data flows through a program, what modifies a data structure etc.). As they trace the control flow and data flow of different parts of the program, they gradually develop a mental model of the given program (Détienne, 2001).

To summarize, tracing has shown to be an effective strategy to comprehend programs. Developers trace the control flow and data flow of relevant parts of the program, based on the task and context, to develop a mental model of the given program. In the next sub-section, we adapt the program tracing idea to the context of software design, and show how a similar tracing strategy can be used to simulate the control flow and data flow of various scenarios in the design.

5.2.4 Adapting the tracing strategy for software designs

In studies with novices (Chapter 4), we observed that students have difficulties in simulating alternate scenarios and detailed control and data flow of various scenarios in the design. Since tracing has been used as an effective strategy to understand the control and data flow of a given program, we believe that the idea of tracing can be extended to design diagrams as well. In this sub-section, we introduce the *"design tracing strategy"*, and explain how the strategy can be applied to scaffold students in simulating various scenarios in the design.

In design tracing, *students trace the control flow and data flow across different diagrams for a given scenario of system execution.* A scenario is a specific behaviour in the design which fulfils the given requirements. For example, consider the design of an automated door locking system which was described as part of Study 1b (Section 4.3.3). In the design, Requirement R3 states that *"When the user chooses the unlock option, and enters the correct passcode, the door should unlock. If the passcode is incorrect, the door remains locked."*. Based on this requirement, various scenarios can be generated, based on the initial state of the door and values of the passcode chosen. For example, a probable scenario can be - *"When the door* *is initially locked and the user selects the unlock option and enters the correct passcode, the door unlocks"*. In design tracing, for each scenario, students are scaffolded to trace the control flow and data flow by constructing a state diagram. The state diagram encompasses the flow of messages across different objects described in the sequence diagram (control flow) and change of variables as a result of the execution of these messages (data flow). Thus, the state diagram serves as the representational model of the scenario.

We now describe the design tracing strategy to trace the scenario - *"When the door is initially locked and the user selects the unlock option and enters the correct passcode, the door unlocks"*. The state diagram for this scenario is shown in Figure 5.3.

- Based on the scenario, students identify relevant data variables from the class diagram (Figure B.2). These are variables which undergo changes based on execution of the scenario. For example, *'optionSelected'*, *'userInputPasscode'*, *'systemPasscode'* and *'lock'* are relevant variables from the class diagram, which are included in all the states (S1 - S4), as shown in Figure 5.3.

- Students then identify the sequence diagram which realises this scenario. In this case, since the scenario describes a user choosing the unlock option, students focus their attention on the 'Unlock Sequence Diagram' (Figure B.5).

- Students then break the scenario into parts. For each part of the scenario, the relevant part of the sequence diagram is identified. For example, the sub-parts of the scenario are "E1: Set option to unlock", "E2: Enter correct passcode" and "E3: Door unlocks". Each of these sub-parts are then mapped to parts in the sequence diagram. For example E1 corresponds to Messages 1-2, E2 to Messages 3-7, and E3 to Message 8-10 in the Unlock sequence diagram (Figure B.5). These messages correspond to the events in the scenario, which are shown as arrows in Figure 5.3.

- Once the relevant data variables and events are identified, students are scaffolded to construct the state diagram which realises this scenario.

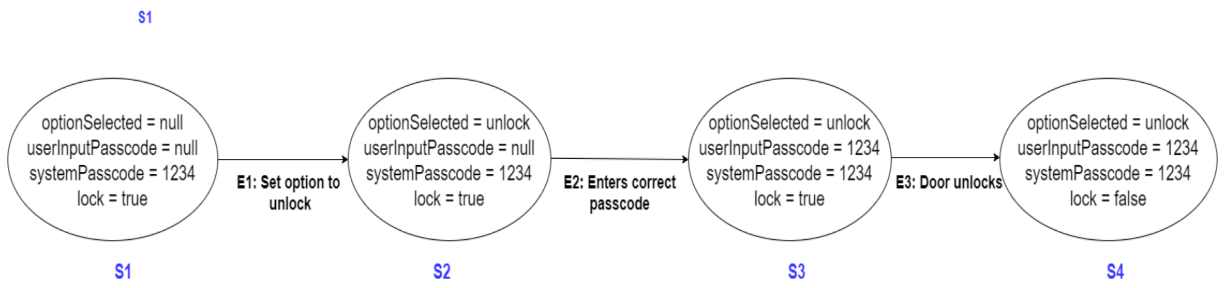


Figure 5.3: Design tracing example for a scenario in an automated door locking system design

The process of constructing the state diagram (Figure 5.3) is as follows:

- Students start with an initial state (S1), whereby initial values of relevant variables are chosen. In Figure 5.3, we see that the values of relevant variables are initialized, and the *lock* variable value is set to *true* since the scenario specifies that the door is initially locked.

- Based on $E1$, students simulate the execution of Messages 1-2 in the unlock sequence diagram (Figure B.5). They then construct the second state, and list down the data variables and their values as a result of this transition. This state corresponds to the state S2 in Figure 5.3. In S2, we see that the value of *optionSelected* changes from *null* to *unlock* as a result of the transition $E1$.

- Students proceed to construct the next state (S3) by simulating the execution of $E2$ (Enter correct passcode). We see that as the correct passcode is entered, the value of *userInputPasscode* changes to 1234, and thus matches the *systemPasscode*.

- Based on the execution of event $E3$ (Door unlocks), the value of *lock* changes from *true* to *false* in State S4.

The example above describes the process of the state diagram model construction for a single scenario for a given requirement. As students identify other scenarios and model them using design tracing, they can uncover defects in the design. For example, consider the following scenario for requirement R2 - *"When the door is initially unlocked and the user selects the unlock option and enters the incorrect passcode, the door remains unlocked"*. This scenario is plausible as there can be a case when the user chooses to unlock option even though it is unlocked. The state diagram for the given scenario is shown in Figure 5.4. From this figure, we see that in state $S4$, although the value of $lock$ should remain $false$, it changes to $true$ due to the invocation of $lock()$ (Message 11) in the Unlock sequence diagram (Figure B.5). Hence, there is an error in the Unlock sequence diagram, when the user enters an incorrect passcode. We see that the process of tracing the control flow and data flow resulted in identification of this defect.

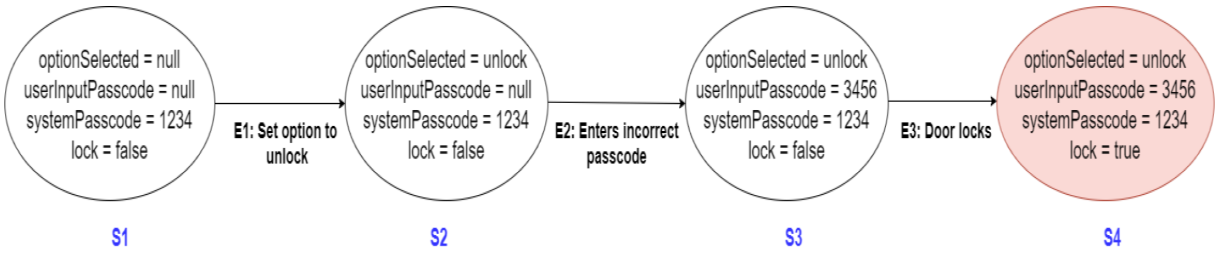


Figure 5.4: Design tracing example for a scenario which does not satisfy the requirement

From the above examples, we hypothesize that as students identify scenarios and model them using design tracing, they will be able to evaluate the given software design better. Firstly, by identifying different scenarios for each requirement, students broadly explore the design solution space. Secondly, by tracing each of these scenarios, they are forced to think deeply about the flow of data and events for each scenario. Hence, both the broad exploration of the design by identifying scenarios, and simulating the data and event flow for each scenario can help in an improved understanding of the design and effectively evaluate the design against the given requirements.

5.2.5 Summary: Theoretical basis of the VeriSIM pedagogy

To summarize, the theoretical basis of the VeriSIM pedagogy is drawn from *model-based learning*, *model progression*, and *program tracing*. The model-based learning paradigm has shown that constructing, manipulating and evaluating mental models improves understanding of scientific concepts. Therefore, *we infer that providing affordances to construct, manipulate, and evaluate models of scenarios in the design can lead to better mental models of the design, leading to better evaluation.* Program tracing has been shown to be an effective strategy to understand a given program. *We extend the program tracing idea to tracing scenarios in a given design.* Model progression can provide a structure to activities in the pedagogy. *Thus,* *progressively enabling students to explore a model, correct an incorrect model, and complete an incomplete model can enable them to effectively create a mental model of a given scenario.*

In the next sections, we describe the VeriSIM pedagogy and how it has been operationalised as a technology-enhanced learning environment to scaffold students in software design evaluation.

5.3 The VeriSIM pedagogy for teaching-learning of software design evaluation

The key idea of the VeriSIM pedagogy is that - *"scaffolding students to identify and construct models of relevant scenarios in the design can lead to effective evaluation of the design diagrams against the given requirements."* The VeriSIM pedagogy scaffolds students to construct models for a given scenario using the design tracing strategy. Using design tracing, they trace the control flow and data flow for a given scenario by constructing a state diagram. The pedagogy aids students to progressively construct models of various scenarios in the design by taking them through four activities -

1. Explore the model - In this activity, students observe an already created state diagram for a given scenario. This activity enables them to understand the different parts of the state diagram model, and how the state diagram corresponds to the control and data flow of the given scenario.
2. Correct the model - In this activity, students are required to correct an incorrect model. The model contains variable-value pairs that need to be corrected, in order to model the given scenario.
3. Complete the model - In this activity, the model contains relevant events, but the data variables and their values are missing. Students are required to add relevant data variables and specify their values in each state.
4. Construct the model - Finally, they construct the entire state diagram for a given scenario.

We hypothesize that this model progression of first exploring the model, then correcting an incorrect model, completing an incomplete model, and finally constructing the model enables students to model various scenarios in the design, thereby enabling them to effectively evaluate the design against the given requirements.

5.4 The VeriSIM Learning Environment

We have incorporated the VeriSIM pedagogy into the VeriSIM learning environment. VeriSIM is a web-based self-learning technology-enhanced learning environment (TELE) which trains learners to apply the design tracing strategy. VeriSIM is available online to use on the following link **https://verisim.tech**. An introduction and videos of various stages in VeriSIM is available at this link - **http://et.iitb.ac.in/ prajish/verisim.html**

In VeriSIM, learners are first introduced to the requirements and design diagrams for an “Automated Door Locking System”. They then trace scenarios in the design using the design tracing pedagogy. Finally, they reflect on their overall learning and how design tracing will be useful for them in the future.

After interacting with VeriSIM, learners should be able to:

1. Understand the purpose of the design tracing technique to trace a given scenario
2. Identify parts of the scenario and how these parts correspond to the control flow and the data flow from the design diagrams
3. Apply the design tracing technique to trace a given scenario by constructing a state diagram. As they trace a given scenario, they should be able to:
    (a) Identify relevant data variables from the class diagram for the given scenario
    (b) Identify relevant events from relevant sequence diagrams for the given scenario
    (c) Visualize control flow and data flow to trace the given scenario

Activities in VeriSIM are presented as various challenges to learners. VeriSIM takes learners through different stages that contain challenges. As learners attempt these challenges, they gain points and acquire skills. We give an overview of the stages and challenges in the next subsection.

5.4.1 Stages and Challenges in VeriSIM

An overview of the stages and challenges in VeriSIM is given in Figure 5.5. The three stages are - Problem Understanding Stage, Design Tracing Stage and Reflection Stage. Each stage comprises challenges, which are followed by evaluation and/or reflection activities (Ge and Land, 2004). The reflection activities make students reflect on what they have done and learnt in a challenge. The evaluation activities test them on what they learnt in the challenge. In VeriSIM, a pedagogical agent serves as a guide to the learner and helps understand the different features in the TELE. The agent also provides the goal of each challenge to the learner, and provides appropriate feedback when they encounter certain errors in the challenges.

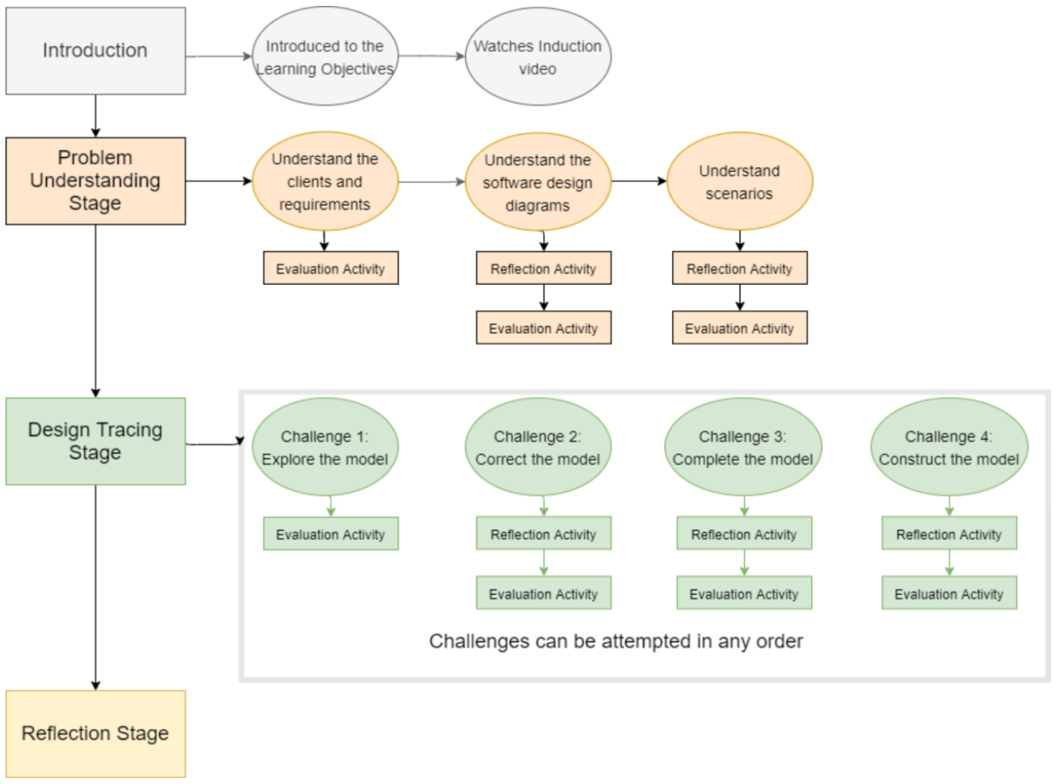

Introduction
Introduced to the Learning Objectives
Watches Induction video
Problem Understanding Stage
Understand the clients and requirements
Understand the software design diagrams
Understand scenarios
Evaluation Activity
Reflection Activity
Design Tracing Stage
Challenge 1: Explore the model
Challenge 2: Correct the model
Challenge 3: Complete the model
Challenge 4: Construct the model
Challenges can be attempted in any order
Reflection Stage

Figure 5.5: Overview of learning activities in VeriSIM

We now give an overview of the three stages and the challenges in each stage.

Problem Understanding Stage

In the Problem Understanding Stage, learners are introduced to the requirements and the software design of an "Automated Door Locking System". This stage comprises three challenges.

1. **Challenge 1 - Understand the client and requirements** - The learner is presented with the requirements of the automated door locking system. They then perform an evaluation activity which tests their understanding of the given requirements.

2. **Challenge 2 - Understand the software design diagrams** - The learner is then presented with the class diagram and 3 sequence diagrams of the door locking system. The TELE gives an overview of the class diagram and sequence diagrams. After the learner goes through these diagram, they attempt the reflection activity, wherein the learner is asked to identify defects (if any) in the diagrams. They then go through an evaluation activity, which tests their understanding of the design diagrams. These reflection and formative evaluation questions enable learners to reflect on and closely analyse the given diagrams.

3. **Challenge 3 - Understand scenarios** - In this challenge, the TELE explains what is meant by a scenario and displays a scenario based on the design. After this, they attempt a reflection activity asking them to identify other scenarios in the design, followed by an evaluation activity which tests their understanding of other scenarios in the design.

Design Tracing Stage

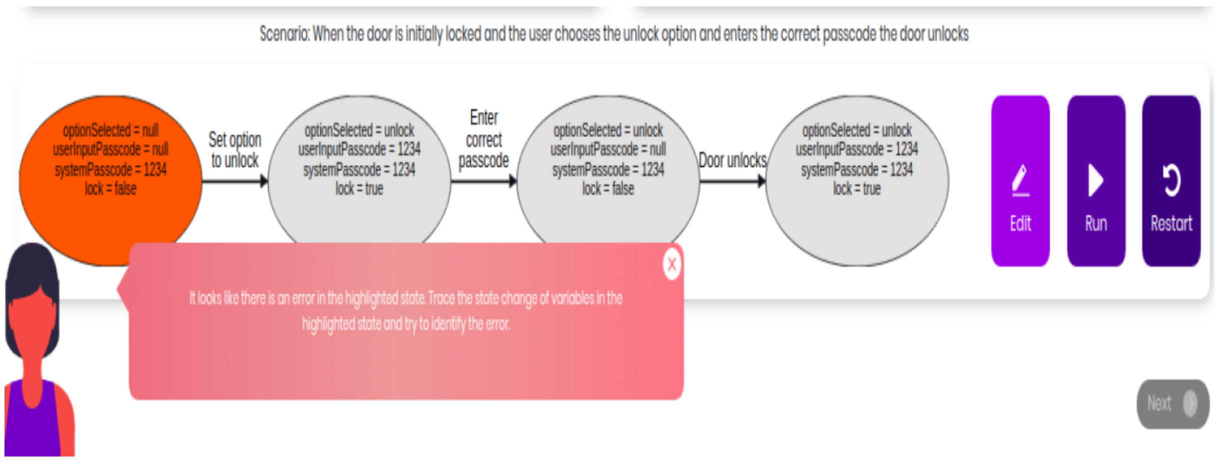
In the Design Tracing Stage, learners are introduced to the design tracing strategy. This stage comprises four challenges in increasing order of complexity. In each challenge, a different scenario of the same design is provided. Learners are free to attempt the challenges in any order.

In all the challenges in the design tracing stage, the objective of the learner is to trace the scenario using the state diagram. The state diagram has to match the expert model for the learner to successfully complete the challenge. Each state in the state diagram is compared with the corresponding state in the expert model and appropriate feedback is provided by the system.

The challenges in the design tracing stage are as follows:

1. **Challenge 1 - Explore the model** - In this challenge, learners are introduced to the state diagram for a scenario. The agent explains the different parts of the state diagram and how the scenario is being traced.

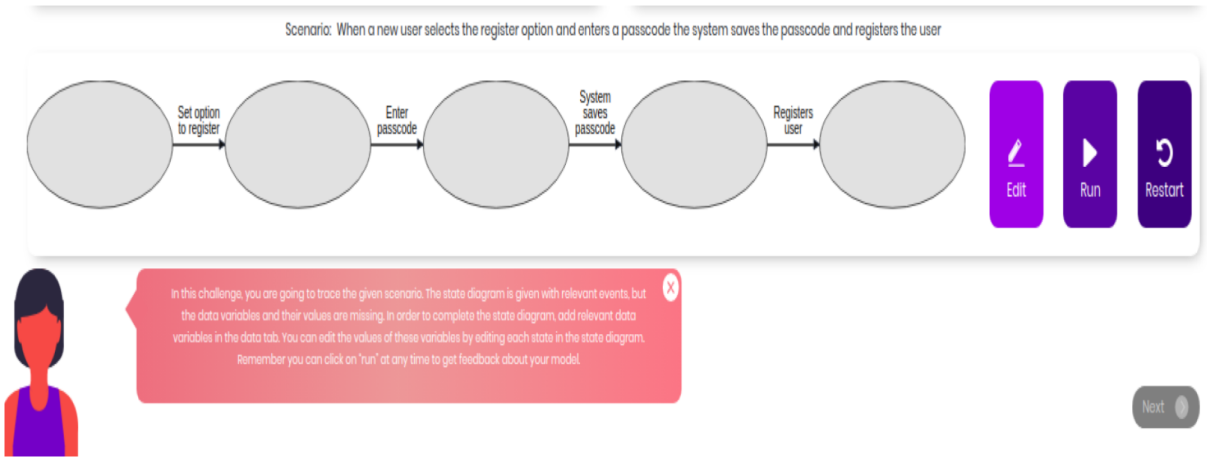
2. **Challenge 2 - Correct the model** - In this challenge, learners are presented with the state diagram for another scenario, but the state diagram contains certain errors. The objective of this challenge is to fix certain errors in the state diagram and thereby correctly trace the given scenario. A screenshot of the model of Challenge 2 is shown in Figure 5.6.


Scenario: When the door is initially locked and the user chooses the unlock option and enters the correct passcode the door unlocks

It looks like there is an error in the highlighted state. Trace the state change of variables in the highlighted state and try to identify the error.

Figure 5.6: An incorrect state diagram shown in Challenge 2

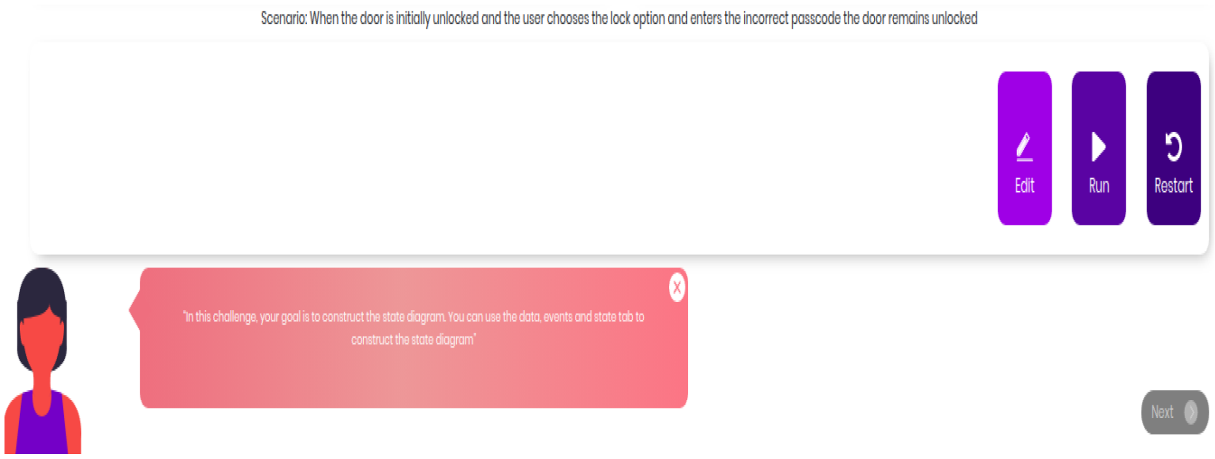
3. **Challenge 3 - Complete the model** - In this challenge, the state diagram is given with relevant events, but the data variables and their values are missing. Learners have to add relevant data variables and specify the values of these variables for each state in the state diagram. A screenshot of the model of Challenge 3 is shown in Figure 5.7.


Scenario: When a new user selects the register option and enters a passcode the system saves the passcode and registers the user

In this challenge, you are going to trace the given scenario. The state diagram is given with relevant events, but the data variables and their values are missing. In order to complete the state diagram, add relevant data variables in the data tab. You can edit the values of these variables by editing each state in the state diagram. Remember you can click on "run" at any time to get feedback about your model.

Figure 5.7: Incomplete model provided in Challenge 3

4. **Challenge 4 - Construct the model** - In this challenge, only the scenario is provided to the learner, and they have to construct the entire state diagram from scratch. A screenshot of the model of Challenge 4 is shown in Figure 5.8.



Scenario: When the door is initially unlocked and the user chooses the lock option and enters the incorrect passcode the door remains unlocked

"In this challenge, your goal is to construct the state diagram. You can use the data, events and state tab to construct the state diagram"

Figure 5.8: Construct the entire model in Challenge 4

Reflection Stage

In the Reflection Stage, the system provides learners with a reflection activity which helps them summarize what they learnt in the previous two stages and their perceptions of the usefulness of the design tracing pedagogy.

5.4.2 Pedagogical Features in VeriSIM

The main goal of VeriSIM is to enable students to effectively identify and construct models of various scenarios in the design, which is based on the model-based learning paradigm. The pedagogical features and activities in the VeriSIM TELE provide students affordances to construct and refine models of scenarios in the design, and visualize the execution of these models. VeriSIM provides scaffolds and feedback to assist learners in the model construction process, and also opportunities for learners to reflect on their learning while interacting with VeriSIM. We explain these pedagogical features in VeriSIM in detail below.

Affordances to construct and revise models of various scenarios in the design

In the 'Design Tracing Stage' challenges, learners go through four challenges which progressively scaffold them to trace a given scenario by constructing a state diagram. The interface in VeriSIM for the 'Design Tracing Stage' has certain prominent spaces which assist learners to construct, refine, and validate the model of a scenario. Figure 5.9 shows the interface for VeriSIM for Challenge 3 in the Design Tracing Stage. (The interface is similar for all four challenges in the Design Tracing Stage). The space on the top is the **design diagram space** which displays the design diagrams. The class diagram is shown on the left, the sequence diagrams are shown on the right. Based on the provided scenario, learners have to correct the incorrect model in Challenge 2, complete the incomplete model in Challenge 3, and construct the entire model in Challenge 4. The learners interact with the model (i.e. the state diagram) in the **model construction space**, as shown in the figure. Learners can construct and revise the model by clicking the 'Edit' button on the interface. On clicking the 'Edit' button, learners are provided with the **model attribute space**, as shown in Figure 5.10. As seen in this figure, learners can add, edit and delete data variables, events or states. These actions are reflected in the model state diagram. For example, when a new data variable is added with an initial value, the data variable and value gets added in all the states in the state diagram. Learners can then modify the values in each state of the state diagram by editing the respective states. Learners can 'execute' their model at any time by clicking on the 'run' button as shown in Figure 5.9. Based on the feedback provided by the model execution visualisation (which we explain in detail below), they can refine their model. In this manner, learners can construct and modify models of various scenarios in the design.

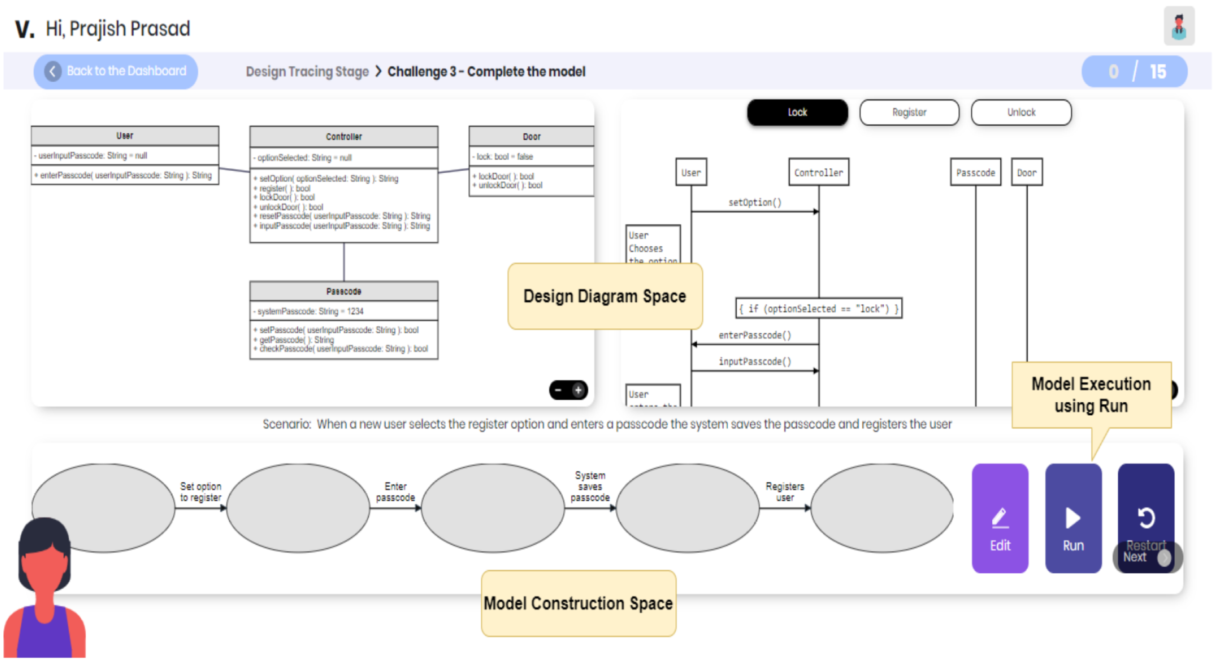
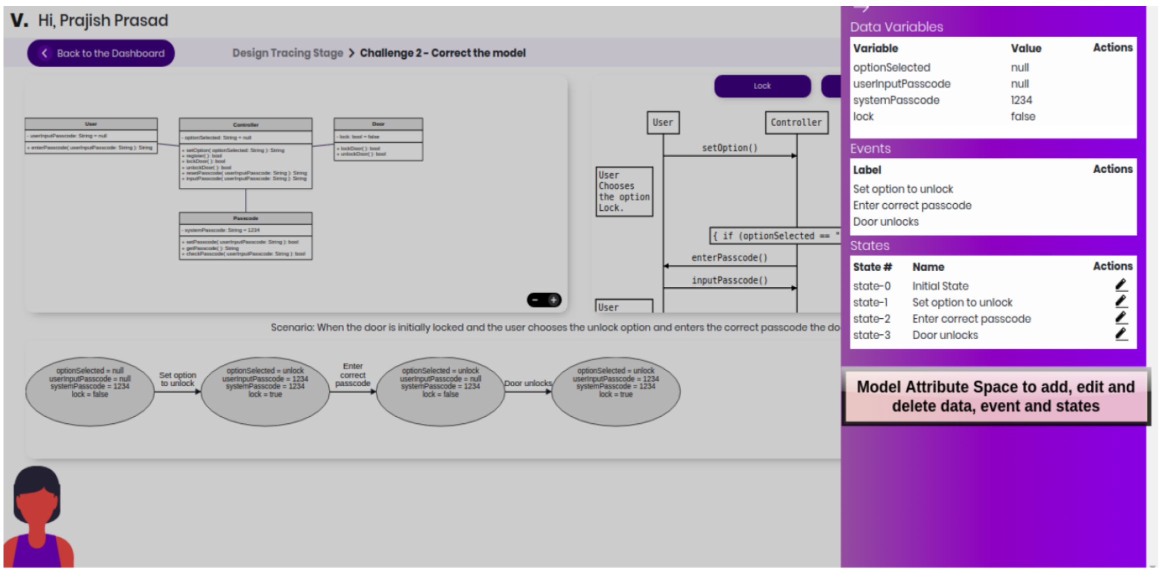

V. Hi, Prajish Prasad
Back to the Dashboard
Design Tracing Stage > Challenge 3 - Complete the model
0 / 15
Lock
Register
Unlock
Design Diagram Space
Model Execution using Run
Scenario: When a new user selects the register option and enters a passcode the system saves the passcode and registers the user
Set option to register
Enter passcode
System saves passcode
Registers user
Edit
Run
Restart
Next
Model Construction Space

Figure 5.9: Various spaces in the design tracing challenges


V. Hi, Prajish Prasad
Back to the Dashboard
Design Tracing Stage > Challenge 2 - Correct the model
Lock
Scenario: When the door is initially locked and the user chooses the unlock option and enters the correct passcode the do
Set option to unlock
Enter correct passcode
Door unlocks
Model Attribute Space to add, edit and delete data, event and states

Data Variables

Variable	Value	Actions
optionSelected	null	
userInputPasscode	null	
systemPasscode	1234	
lock	false	

Events

Label	Actions
Set option to unlock	
Enter correct passcode	
Door unlocks	

States

State #	Name	Actions
state-0	Initial State	
state-1	Set option to unlock	
state-2	Enter correct passcode	
state-3	Door unlocks	

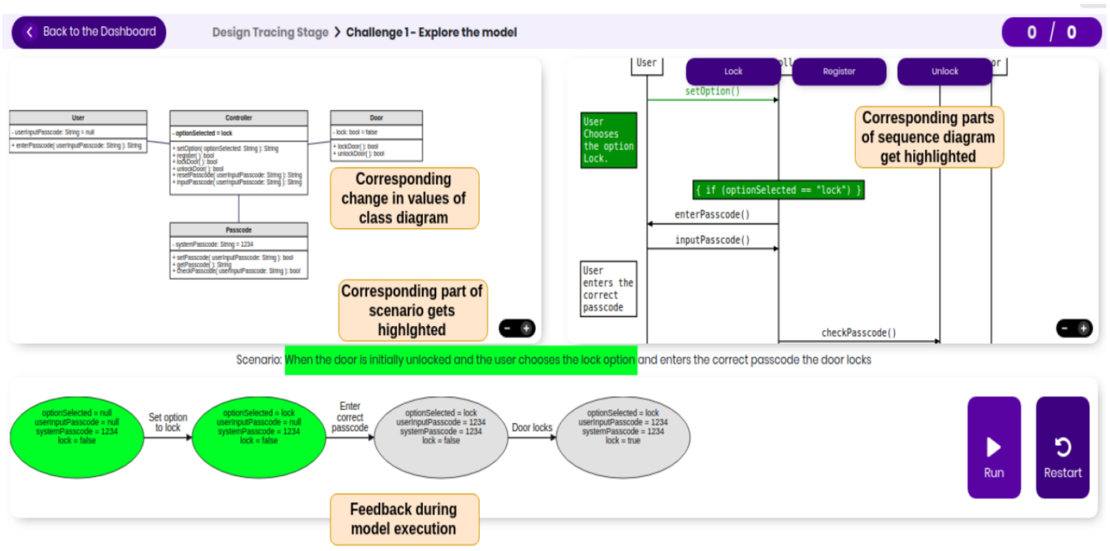
Figure 5.10: The model attribute space

Affordances to visualize execution of the model

VeriSIM provides the affordance to visualise the state-wise execution of the model, by clicking on the ‘run’ button in the interface. Each click of the ‘run’ button results in the system checking the correctness of a particular state in a linear order. When the learner clicks on ‘run’ after constructing the state diagram, the system matches the first state with the expert model. If correct, the state is highlighted in green and the system proceeds in checking the next state. If a state contains an error, the execution stops, the state is highlighted in red and the agent provides a feedback message indicating an error. Whenever a learner clicks on ‘run’ the next time, the execution starts from the first state.

Based on each click of the “Run” button, appropriate highlights are made in the class and sequence diagram as shown in Figure 5.11. With the help of these highlights, learners are able to visualise the relevant changes in the class and sequence diagrams in a particular state. A transition to a new state triggers a set of events in the sequence diagram. The visualization shows the corresponding changes in the class diagram (change of values of variables) for each event/message in the sequence diagram. The learner can replay this execution step, review previous execution steps and move forward. As the learner observes these visualizations and corresponding changes in the design diagrams, they are able to visualize the state change of different variables at a given state and also understand the relationship between the class and sequence diagram. Thus, the ‘run’ feature enables students to validate their model, and refine the model based on the feedback provided.

The ‘run’ feature is based on existing work in program visualization literature. Program visualization systems (PV) are used to display program executions automatically or semi-automatically. These systems allow teachers to demonstrate and students to explore the runtime behaviour of programs (Sorva et al., 2013). In our case, we have extended the program visualization idea to software design diagrams. In VeriSIM, although the execution is entirely simulated, and not automatic, we believe the simulation can help in students’ understanding of the execution of the scenario and also provide opportunities to validate and refine their models.


Back to the Dashboard
Design Tracing Stage > Challenge 1 - Explore the model
0 / 0
Corresponding change in values of class diagram
Corresponding part of scenario gets highlighted
Corresponding parts of sequence diagram get highlighted
Feedback during model execution
Lock
Register
Unlock
Run
Restart
Set option to lock
Enter correct passcode
Door locks

Scenario: When the door is initially unlocked and the user chooses the lock option and enters the correct passcode the door locks

Figure 5.11: Affordance of the run feature

Scaffolding and feedback provided by the pedagogical agent

Pedagogical agents are lifelike characters designed to facilitate learning in computer-based environments.The role of an agent is to promote student learning, such as guiding students' attention in interactive multimedia environments (Moreno, 2004), providing students with feedback, modeling, and guidance (Moreno et al., 2001). It can make the environment more human like, engaging and motivating (Moreno and Flowerday, 2006).

A summary of various types of scaffolds provided by the agent is summarized in Table 5.1. In VeriSIM, the agent serves as a guide to the learner and can help in easing the cognitive load of encountering a new learning environment. When learners encounter a new interface in VeriSIM, (for example: Challenge 1 in the Design Tracing stage), the agent provides a tour and explains various parts of the interface. The agent also provides the goal of each challenge to the learner. Key concepts in VeriSIM, like the design tracing pedagogy, is also explained by focusing the attention of the learner on various parts of the interface.

Another important function of the agent is to provide appropriate feedback to the learner in the challenges. As the learner uses the 'run' feature to validate their model, the agent provides appropriate feedback regarding the correctness of each state in the state diagram. The feedback messages do not directly tell learners what to do, but directs their attention to what can be done to resolve the error.

Message Type	**Example**
Explaining parts of the interface	*There are three stages comprising of different challenges.* *Complete each stage to unlock challenges in the next stage.* *Earn upto 2000 points as you attempt these challenges.* *You also gain skills in design tracing as you successfully complete these challenges*
Explaining concepts	*In design tracing, we trace the sequence of data and function changes across different diagrams for a given scenario of system execution.*
Introducing goals of a challenge	*In this challenge, your goal is to construct the state diagram.* *You can use the data, events and state tab to construct the state diagram*
Providing Feedback	*It looks like there is an error in the highlighted state.* *Trace the state change of variables and try to identify the error*

Table 5.1: Pedagogical agent message types in VeriSIM

Scaffolds for articulation and reflection

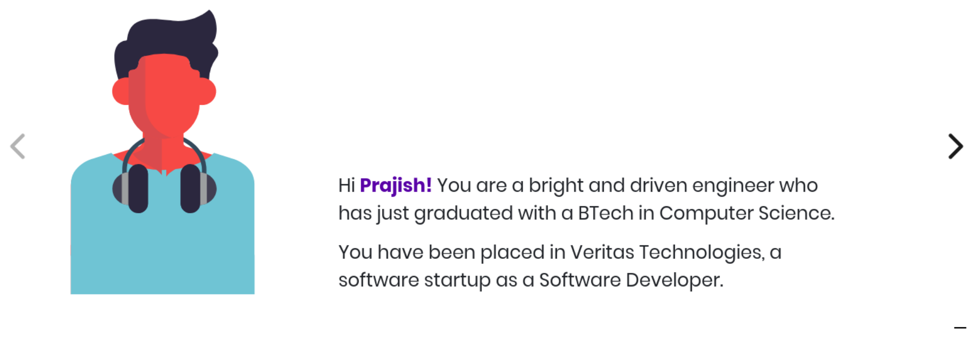
After every challenge, there are reflection activities which enable learners to articulate what they learnt in the challenge. These reflection activities are termed as reflection-on-action activities (Schön, 1987), where students evaluate their study process after completing the whole inquiry cycle. These are similar to elaboration prompts (Ge and Land, 2004), which are designed to prompt learners to articulate their thoughts and explicit explanations. Such elaboration prompts have been shown to be effective in facilitating knowledge building of learners (Lin and Lehman, 1999). These scaffolds also serve as reminders to plan their future activities and monitor their progress (Quintana et al., 2004). In VeriSIM, the reflection activities allow learners to articulate what they learnt and apply these learnings in future challenges.

5.5 A Walk-through of how Learners Interact with VeriSIM

In this section, we present how learners go through the various stages and challenges in VeriSIM.

Introduction to VeriSIM

After learners log in to the system, they are presented with the learning objectives of VeriSIM. They then view an introductory video. In the video, learners are presented with a situation where they have graduated and entered a software startup company as a software developer. They are introduced to their team and project manager. The project manager then explains the importance of design evaluation and why it is essential. The objective of this introduction is to set the context and situate the learner in a software team setting. A screenshot of the Introduction phase as shown in Figure 5.12.


Hi **Prajish!** You are a bright and driven engineer who has just graduated with a BTech in Computer Science.

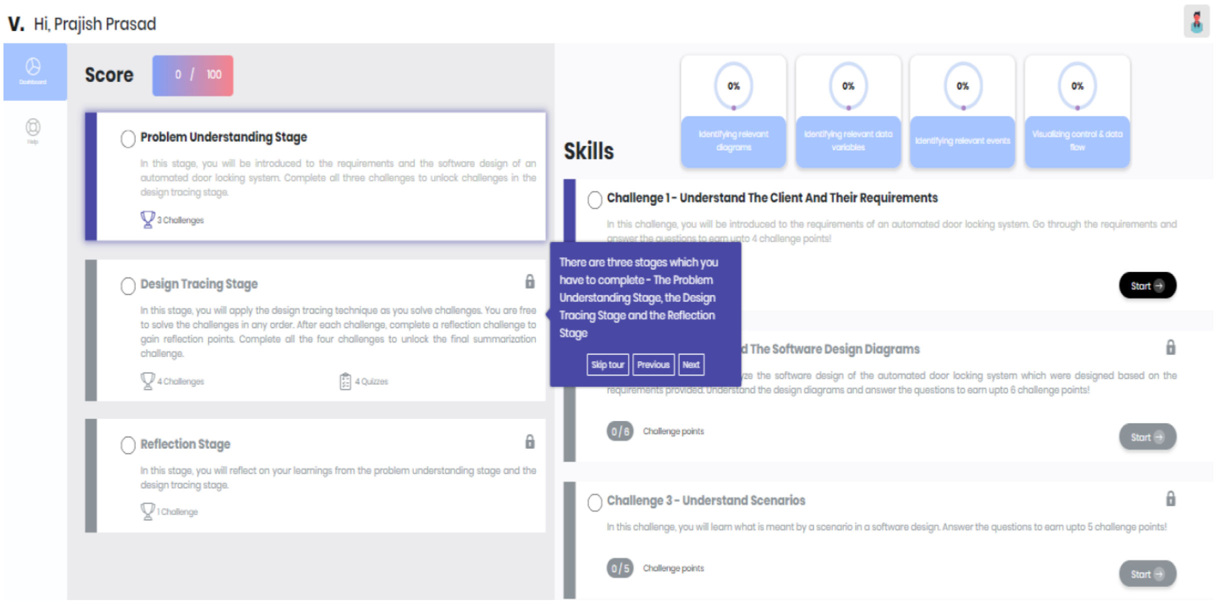
You have been placed in Veritas Technologies, a software startup as a Software Developer.

Figure 5.12: Introduction screen of VeriSIM

After watching the video, learners are directed to the VeriSIM dashboard as shown in Figure 5.13. The dashboard contains the three stages and corresponding challenges in each stage. When learners view the dashboard for the first time, they are provided with a tour (the purple box in Figure 5.13), which explains the broad objectives of various stages and challenges in VeriSIM. They can choose to take or skip the tour. Learners now proceed with the challenges in the first stage i.e. the 'Problem Understanding Stage'.

Problem Understanding Stage

The first challenge in the Problem Understanding Stage is the **'Understand the Client and their Requirements'** challenge. In this challenge, learners are introduced to the requirements of an automated door locking system, as shown in Figure 5.14.


V. Hi, Prajish Prasad

Score 0 / 100

Problem Understanding Stage

In this stage, you will be introduced to the requirements and the software design of an automated door locking system. Complete all three challenges to unlock challenges in the design tracing stage.

3 Challenges

Design Tracing Stage

In this stage, you will apply the design tracing technique as you solve challenges. You are free to solve the challenges in any order. After each challenge, complete a reflection challenge to gain reflection points. Complete all the four challenges to unlock the final summarization challenge.

4 Challenges 4 Quizzes

Reflection Stage

In this stage, you will reflect on your learnings from the problem understanding stage and the design tracing stage.

1 Challenge

Skills

Dashboard
Help
0%
Identifying relevant diagrams
0%
Identifying relevant data variables
0%
Identifying relevant events
0%
Visualizing control & data flow
Start
Start
Start
Skip tour
Previous
Next

Challenge 1 - Understand The Client And Their Requirements

In this challenge, you will be introduced to the requirements of an automated door locking system. Go through the requirements and answer the questions to earn upto 4 challenge points!

There are three stages which you have to complete - The Problem Understanding Stage, the Design Tracing Stage and the Reflection Stage

d The Software Design Diagrams

ze the software design of the automated door locking system which were designed based on the requirements provided. Understand the design diagrams and answer the questions to earn upto 6 challenge points!

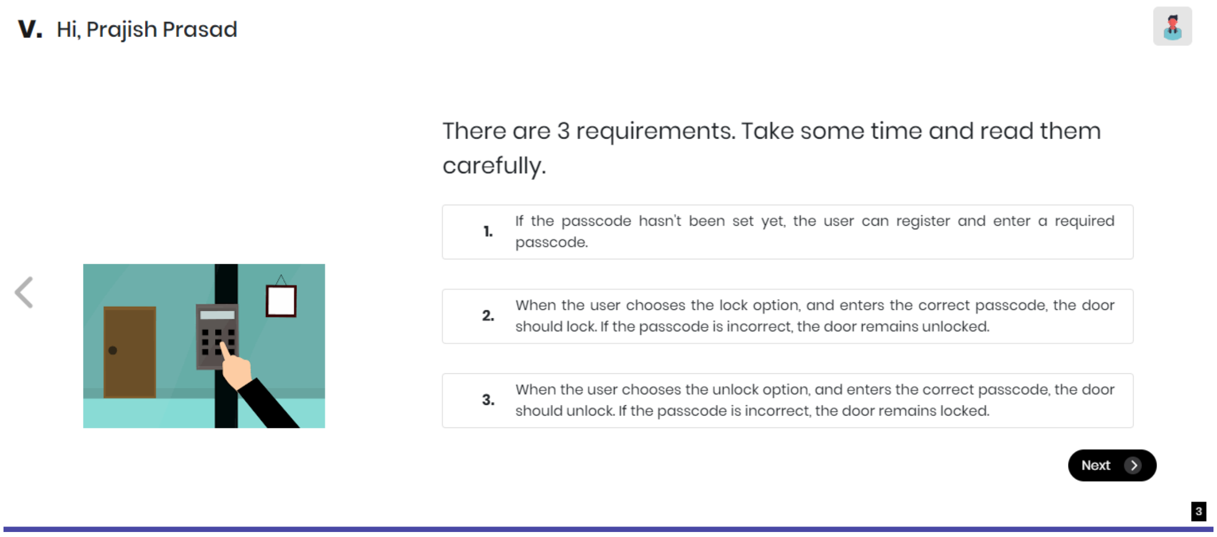
0 / 6 Challenge points

Challenge 3 - Understand Scenarios

In this challenge, you will learn what is meant by a scenario in a software design. Answer the questions to earn upto 5 challenge points!

0 / 5 Challenge points

Figure 5.13: The VeriSIM dashboard


V. Hi, Prajish Prasad

There are 3 requirements. Take some time and read them carefully.

1. If the passcode hasn't been set yet, the user can register and enter a required passcode.

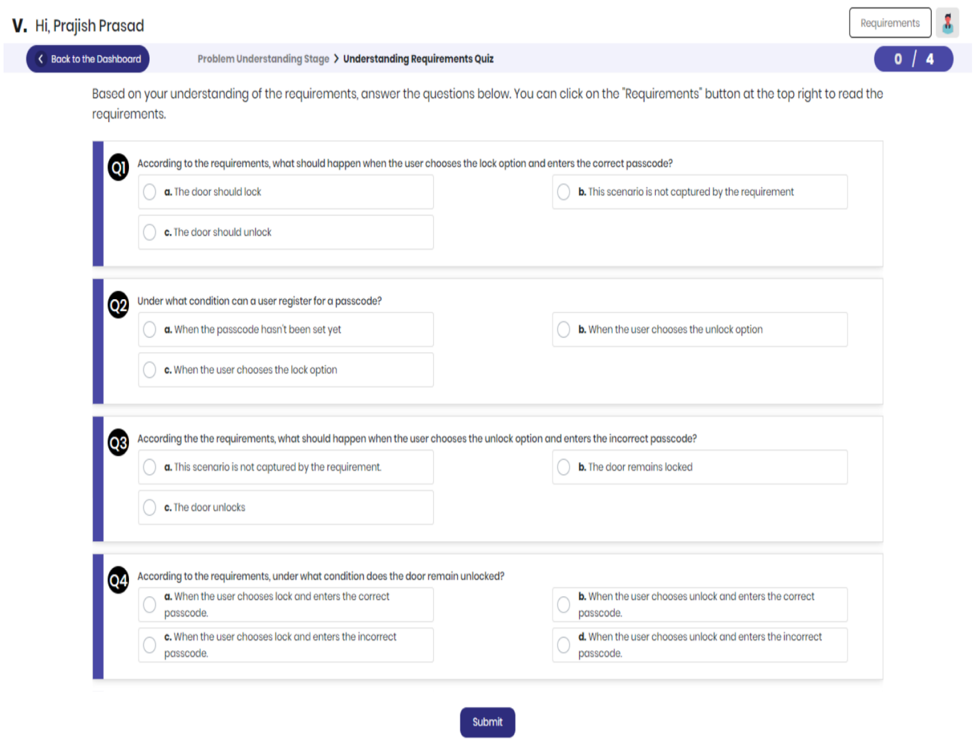
2. When the user chooses the lock option, and enters the correct passcode, the door should lock. If the passcode is incorrect, the door remains unlocked.

3. When the user chooses the unlock option, and enters the correct passcode, the door should unlock. If the passcode is incorrect, the door remains locked.

Figure 5.14: The requirements from 'Understand the Client and their Requirements' challenge

After that, they go through an evaluation activity as shown in Figure 5.15. This evaluation activity contains questions which test their understanding of the requirements, and provides feedback on their answers to the questions.

After attempting the evaluation activity, learners proceed to the second challenge in the Problem Understanding Stage i.e. the **'Understand the Software Design Diagrams'** challenge. In this challenge, they are provided with the class diagram and sequence diagrams of the automated door locking system, as shown in Figure 5.16. The agent provides a brief explanation of the class diagram and sequence diagrams, and asks learners to go through these design diagrams, before attempting the reflection and evaluation activities. After analysing the design diagrams, learners click on the ‘next’ button and attempt a reflection activity.


V. Hi, Prajish Prasad

Requirements

Back to the Dashboard — Problem Understanding Stage > Understanding Requirements Quiz — 0 / 4

Based on your understanding of the requirements, answer the questions below. You can click on the "Requirements" button at the top right to read the requirements.

Q1 According to the requirements, what should happen when the user chooses the lock option and enters the correct passcode?
- a. The door should lock
- b. This scenario is not captured by the requirement
- c. The door should unlock

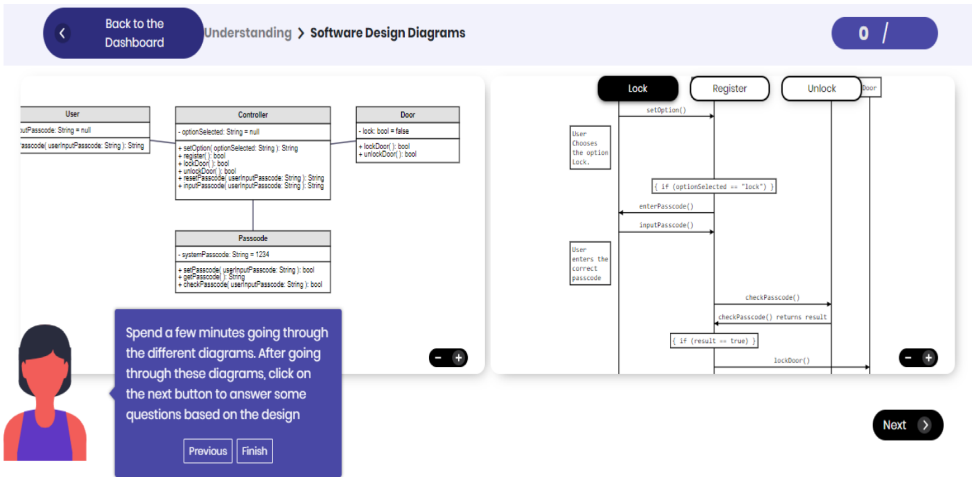
Q2 Under what condition can a user register for a passcode?
- a. When the passcode hasn't been set yet
- b. When the user chooses the unlock option
- c. When the user chooses the lock option

Q3 According the the requirements, what should happen when the user chooses the unlock option and enters the incorrect passcode?
- a. This scenario is not captured by the requirement.
- b. The door remains locked
- c. The door unlocks

Q4 According to the requirements, under what condition does the door remain unlocked?
- a. When the user chooses lock and enters the correct passcode.
- b. When the user chooses unlock and enters the correct passcode.
- c. When the user chooses lock and enters the incorrect passcode.
- d. When the user chooses unlock and enters the incorrect passcode.

Submit

Figure 5.15: The evaluation activity from ‘Understand the Client and their Requirements’ challenge

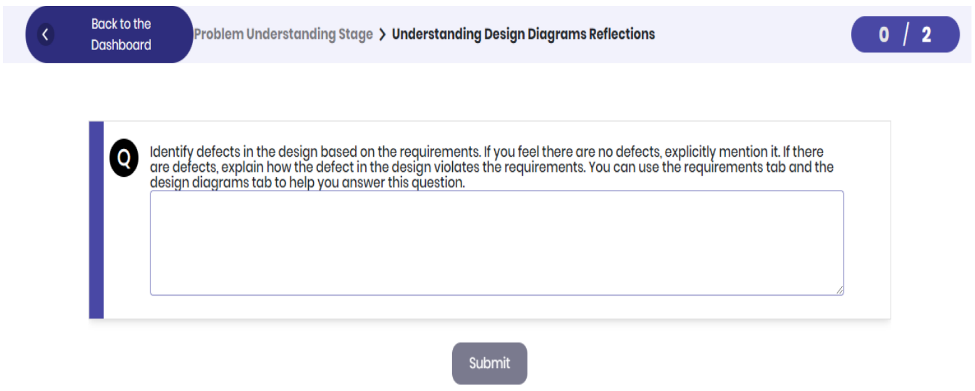


Back to the Dashboard
Understanding
Software Design Diagrams
0 /
User
putPasscode: String = null
asscode(userInputPasscode: String): String
Controller
- optionSelected: String = null
+ setOption(optionSelected: String): String
+ register( ): bool
+ lockDoor( ): bool
+ unlockDoor( ): bool
+ resetPasscode(userInputPasscode: String): String
+ inputPasscode(userInputPasscode: String): String
Door
- lock: bool = false
+ lockDoor( ): bool
+ unlockDoor( ): bool
Passcode
- systemPasscode: String = 1234
+ setPasscode(userInputPasscode: String): bool
+ getPasscode( ): String
+ checkPasscode(userInputPasscode: String): bool
Lock
Register
Unlock
Door
setOption()
User Chooses the option Lock.
{ if (optionSelected == "lock") }
enterPasscode()
inputPasscode()
User enters the correct passcode
checkPasscode()
checkPasscode() returns result
{ if (result == true) }
lockDoor()
Next
Previous
Finish

Spend a few minutes going through the different diagrams. After going through these diagrams, click on the next button to answer some questions based on the design

Figure 5.16: The ‘Understand the Software Design Diagrams’ challenge

In the reflection activity, learners are asked to identify defects in the design based on the requirements, as shown in Figure 5.17. After this reflection, they attempt an evaluation activity which tests their understanding of the design diagrams. The evaluation activity contains multiple choice questions (e.g. *'What does the inputPasscode() function return?'*, *'Which is the function used for creating a passcode during first-time registration?'*), which students can answer by analysing the class and sequence diagrams of the automated door locking system. For the evaluation and reflection activities, they can view the requirements and design diagrams at any time, by clicking on the 'Diagrams' and 'Requirements' button at the top right of the screen (Figure 5.17).

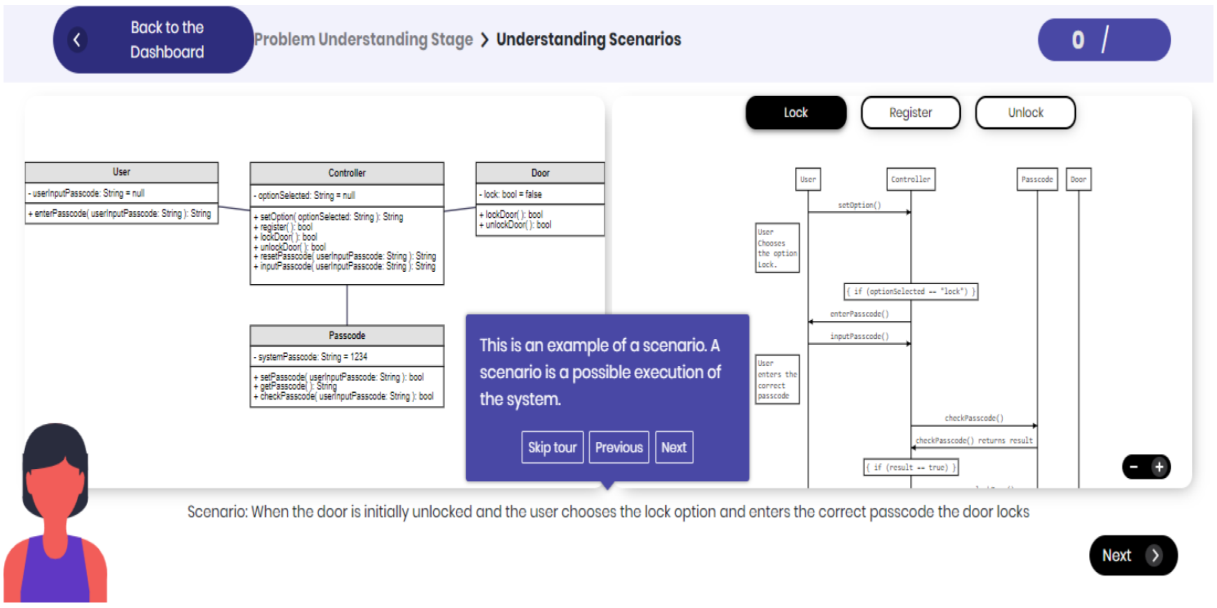

Back to the Dashboard
Problem Understanding Stage > Understanding Design Diagrams Reflections
0 / 2
Q
Submit

Identify defects in the design based on the requirements. If you feel there are no defects, explicitly mention it. If there are defects, explain how the defect in the design violates the requirements. You can use the requirements tab and the design diagrams tab to help you answer this question.

Figure 5.17: Reflection activity in 'Understand the Software Design Diagrams' challenge

They then move on to the third challenge in the Problem Understanding Stage i.e. the **'Understanding Scenarios'** challenge. In this challenge, the agent provides a description and example of a scenario to learners, as shown in Figure 5.18. They then proceed to a reflection activity, which asks them to list other scenarios in the design (*"What are other scenarios in the design? List at least three. You can use the requirements tab and the design diagrams tab to help you answer this question"*). After listing down the scenarios, they then go through an evaluation activity, which tests their understanding of different scenarios in the design. Examples of questions asked in the evaluation activity are shown in Figure 5.19.

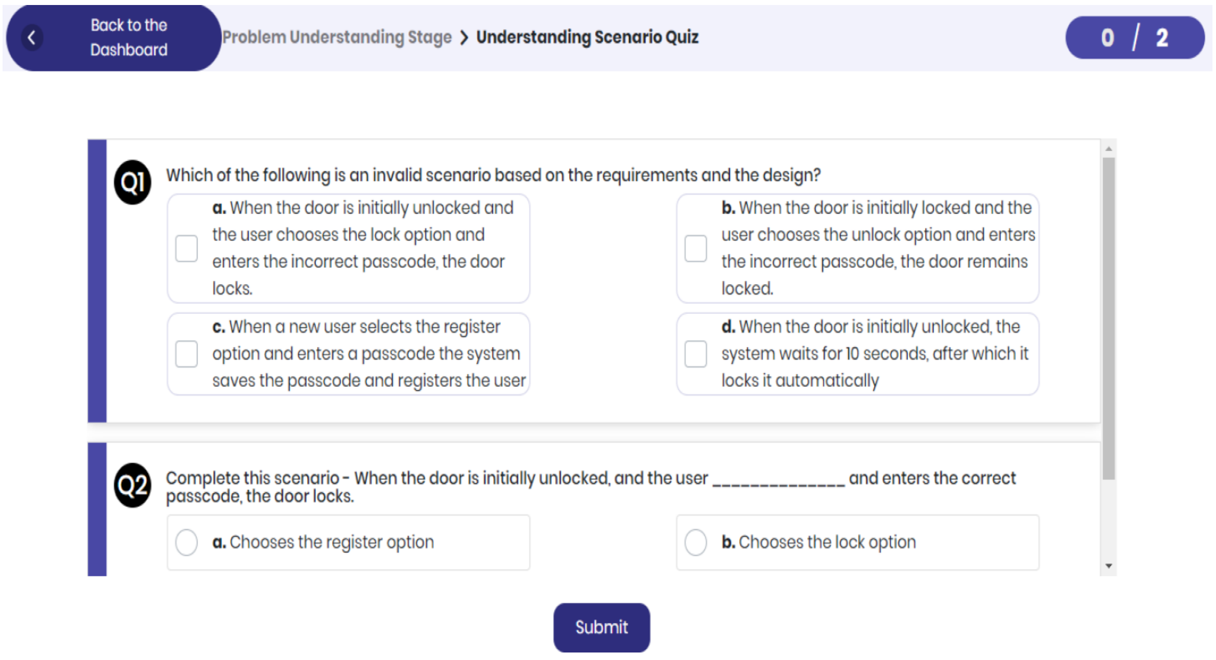
After successfully attempting the reflection and evaluation activities in the **'Understanding Scenarios'** challenge, learners have now unlocked the next stage, i.e. the Design Tracing Stage. They are directed towards the dashboard, and can attempt any of the challenges in the Design Tracing Stage.


Back to the Dashboard
Problem Understanding Stage > Understanding Scenarios
0 /
User
- userInputPasscode: String = null
+ enterPasscode(userInputPasscode: String): String
Controller
- optionSelected: String = null
+ setOption(optionSelected: String): String
+ register( ): bool
+ lockDoor( ): bool
+ unlockDoor( ): bool
+ resetPasscode(userInputPasscode: String): String
+ inputPasscode(userInputPasscode: String): String
Door
- lock: bool = false
+ lockDoor( ): bool
+ unlockDoor( ): bool
Passcode
- systemPasscode: String = 1234
+ setPasscode(userInputPasscode: String): bool
+ getPasscode( ): String
+ checkPasscode(userInputPasscode: String): bool
Lock
Register
Unlock
User
Controller
Passcode
Door
setOption()
User Chooses the option Lock.
{ if (optionSelected == "lock") }
enterPasscode()
inputPasscode()
User enters the correct passcode
checkPasscode()
checkPasscode() returns result
{ if (result == true) }
Skip tour
Previous
Next
Next

This is an example of a scenario. A scenario is a possible execution of the system.

Scenario: When the door is initially unlocked and the user chooses the lock option and enters the correct passcode the door locks

Figure 5.18: The 'Understanding Scenarios' challenge



Q1 Which of the following is an invalid scenario based on the requirements and the design?

a. When the door is initially unlocked and the user chooses the lock option and enters the incorrect passcode, the door locks.

b. When the door is initially locked and the user chooses the unlock option and enters the incorrect passcode, the door remains locked.

c. When a new user selects the register option and enters a passcode the system saves the passcode and registers the user

d. When the door is initially unlocked, the system waits for 10 seconds, after which it locks it automatically

Q2 Complete this scenario - When the door is initially unlocked, and the user _____________ and enters the correct passcode, the door locks.

a. Chooses the register option

b. Chooses the lock option

Figure 5.19: Evaluation activity in 'Understanding Scenarios' challenge

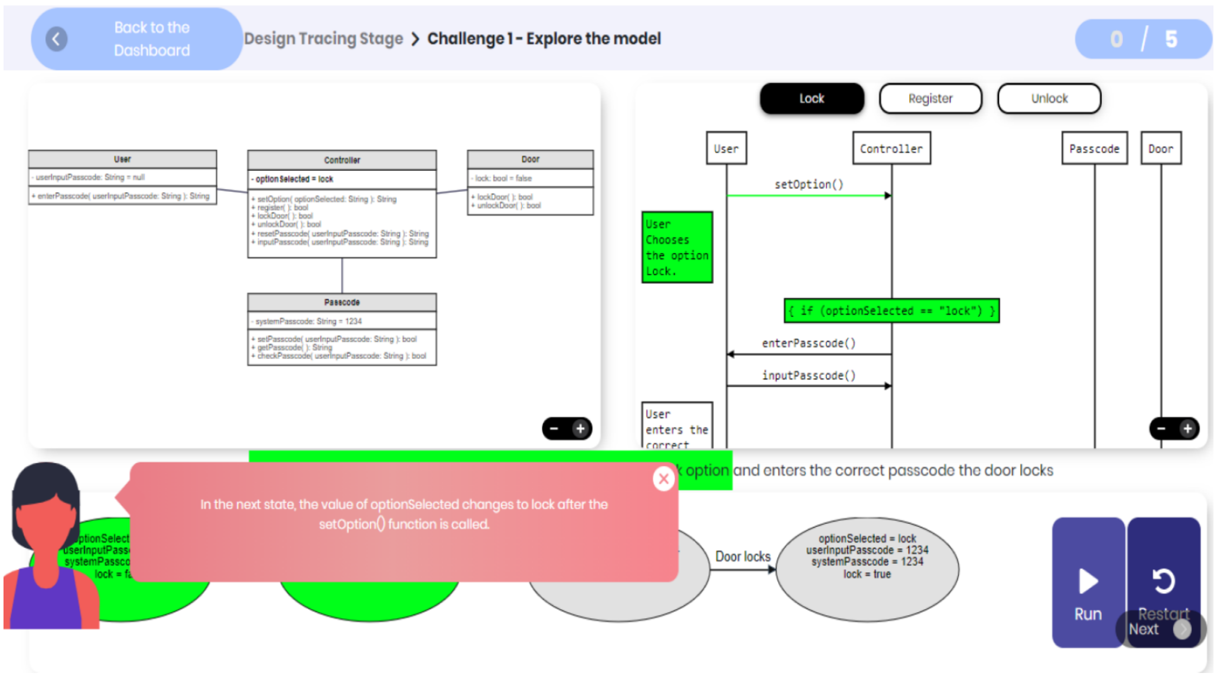
Design Tracing Stage

The Design Tracing Stage has four challenges - **'Challenge 1 - Explore the model', Challenge 2 - Correct the model', 'Challenge 3 - Complete the model', and 'Challenge 4 - Construct**

the model'. This is the prescribed order, although learners can go through them in any order.

Challenge 1 - Explore the Model

In this challenge, learners are introduced to the state diagram for the first time. In this challenge, the learner is presented with the correct state diagram which describes the given scenario. The agent describes the design tracing strategy and the model i.e. state diagram, and its different parts, such as the states and the transitions. The agent then suggests the learner to click the 'Run' button to view the execution of the model. For each click of the 'Run', the appropriate state is highlighted in green, and appropriate messages are provided by the agent. The agent describes the changes in the design diagrams when control transfers to a particular state, as shown in Figure 5.20. After the execution reaches the final state, the learner can proceed to an evaluation activity. The evaluation activity contains questions which test learners' understanding of the state diagram (e.g. *The state diagram contains relevant data variables from the ____ diagram*). After completing the evaluation activity, learners can proceed to Challenge 2.



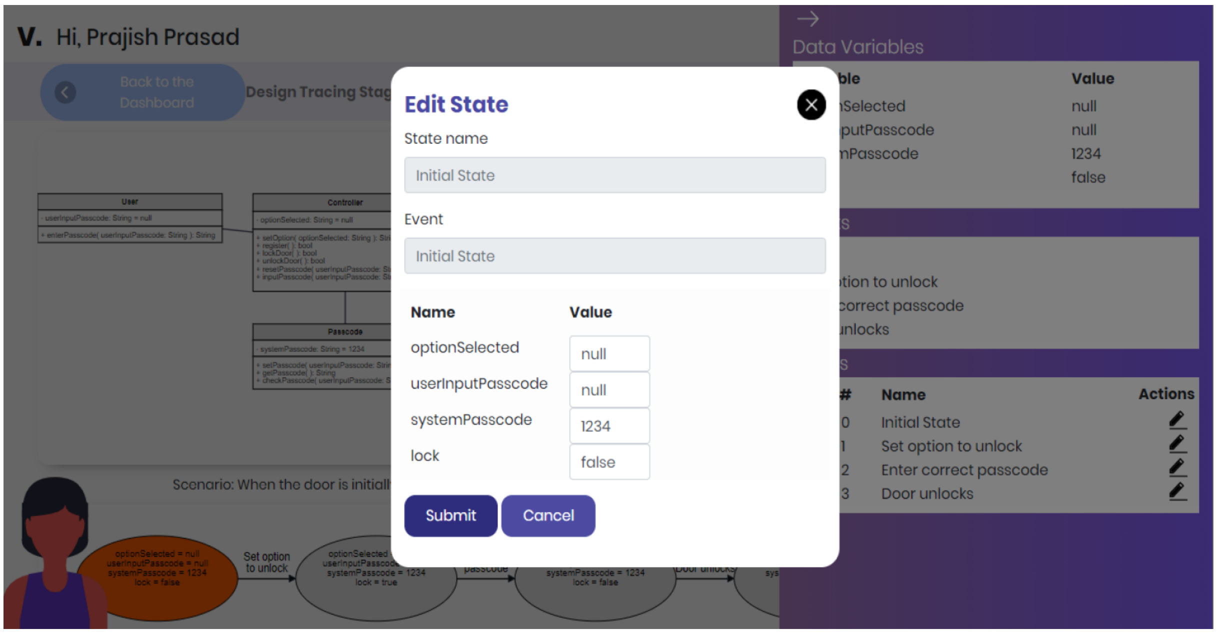
Back to the Dashboard
Design Tracing Stage > Challenge 1 - Explore the model
0 / 5
User
- userInputPasscode: String = null
+ enterPasscode(userInputPasscode: String): String
Controller
- optionSelected = lock
+ setOption(optionSelected: String): String
+ register( ): bool
+ lockDoor( ): bool
+ unlockDoor( ): bool
+ resetPasscode(userInputPasscode: String): String
+ inputPasscode(userInputPasscode: String): String
Door
- lock: bool = false
+ lockDoor( ): bool
+ unlockDoor( ): bool
Passcode
- systemPasscode: String = 1234
+ setPasscode(userInputPasscode: String): bool
+ getPasscode( ): String
+ checkPasscode(userInputPasscode: String): bool
Lock
Register
Unlock
User
Controller
Passcode
Door
setOption()
User Chooses the option Lock.
{ if (optionSelected == "lock") }
enterPasscode()
inputPasscode()
User enters the correct
option and enters the correct passcode the door locks
Door locks
optionSelected = lock
userInputPasscode = 1234
systemPasscode = 1234
lock = true
Run
Restart
Next

In the next state, the value of optionSelected changes to lock after the setOption() function is called.

Figure 5.20: The agent explaining the state diagram execution for Challenge 1

Challenge 2 - Correct the Model

In this challenge, learners are presented with another scenario and an incorrect state diagram which models this scenario. The agent explains the objective of this challenge, i.e. learners are expected to fix errors in the given state diagram. They can click the 'Run' button at any time to validate the model. If a state contains an error, and learners click on 'Run', the agent indicates an error in the state by highlighting that state in red, and asks learners to trace the state change of variables in the highlighted state to identify the error. To modify the state, learners can click on the 'Edit' button, which opens the model attribute space, where they can edit the appropriate state by changing the appropriate values (see Figure 5.21). For example, for the first state in Challenge 2, the learner is required to change the value of 'lock' from 'false' to 'true', since the door is initially at the locked state. Learners are free to edit states and check the execution of the model at any time during the challenge.


V. Hi, Prajish Prasad

Back to the Dashboard

Design Tracing Stag

Scenario: When the door is initiall

Edit State

State name

Initial State

Event

Initial State

Name	Value
optionSelected	null
userInputPasscode	null
systemPasscode	1234
lock	false

Submit Cancel

Data Variables

ble	Value
Selected	null
putPasscode	null
nPasscode	1234
	false

tion to unlock
correct passcode
nlocks

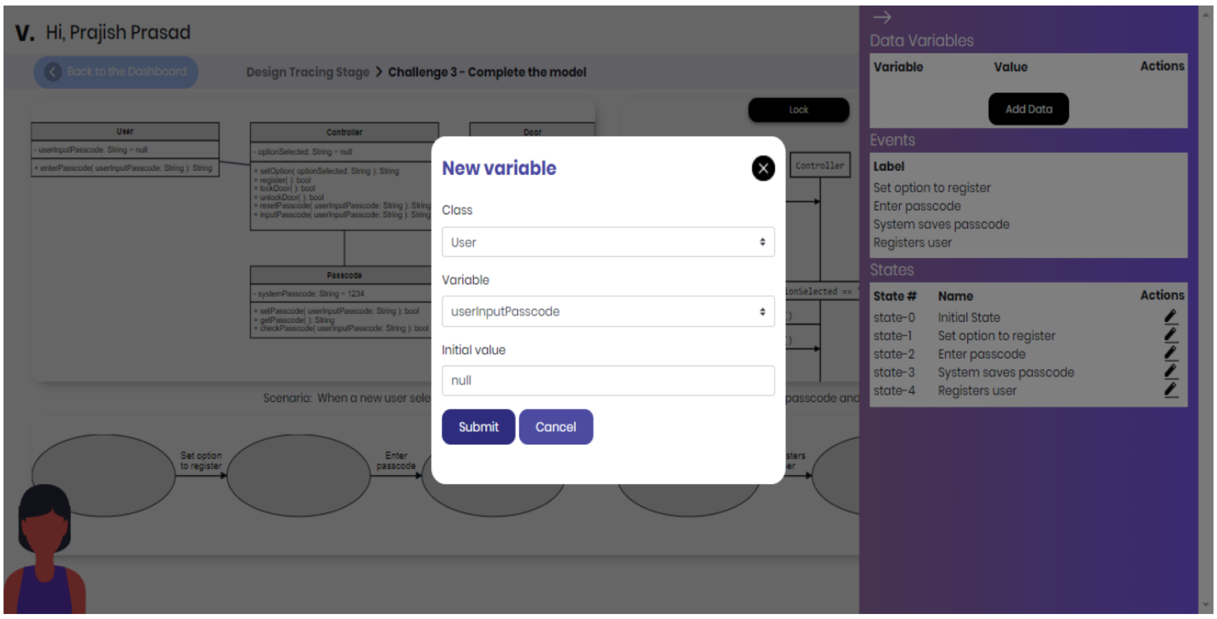
#	Name	Actions
0	Initial State	✎
1	Set option to unlock	✎
2	Enter correct passcode	✎
3	Door unlocks	✎

Figure 5.21: In Challenge 2, learners can change values of variables in appropriate states

After learners correct all the states in the state diagram, they can click on 'run' to validate their model. They then move onto a reflection activity, which asks them to summarize what they learnt and what they found difficult in Challenge 2. After answering the reflection question, they then attempt an evaluation activity. In the evaluation activity, they are provided with an incorrect state diagram. They attempt multiple-choice questions which ask them to identify incorrect states and also choose the appropriate correct states. After attempting this evaluation activity, learners move on to Challenge 3.

Challenge 3 - Complete the Model

In Challenge 3, learners are provided with the state diagram containing relevant events, but the data variables and their values are missing. Learners can use the 'edit' feature to add data variables in the data tab (see Figure 5.22), and edit appropriate states to reflect the change in values of variables in these states. Similar to previous challenges, they can execute the state diagram at any time to get feedback on their model. After completing the state diagram with appropriate data variables and values, they move on to reflection and evaluation activities, similar to the ones after Challenge 2.



V. Hi, Prajish Prasad
Back to the Dashboard
Design Tracing Stage > Challenge 3 - Complete the model
Lock
User
- userInputPasscode: String = null
+ enterPasscode(userInputPasscode: String): String
Controller
- optionSelected: String = null
+ setOption(optionSelected: String): String
+ register( ): bool
+ lockDoor( ): bool
+ unlockDoor( ): bool
+ resetPasscode(userInputPasscode: String): String
+ inputPasscode(userInputPasscode: String): String
Door
Passcode
- systemPasscode: String = 1234
+ setPasscode(userInputPasscode: String): bool
+ getPasscode( ): String
+ checkPasscode(userInputPasscode: String): bool
Scenario: When a new user sele
passcode and
Set option to register
Enter passcode
Submit
Cancel
Add Data

New variable

Class: User

Variable: userInputPasscode

Initial value: null

Data Variables

Variable	Value	Actions

Events

Label
Set option to register
Enter passcode
System saves passcode
Registers user

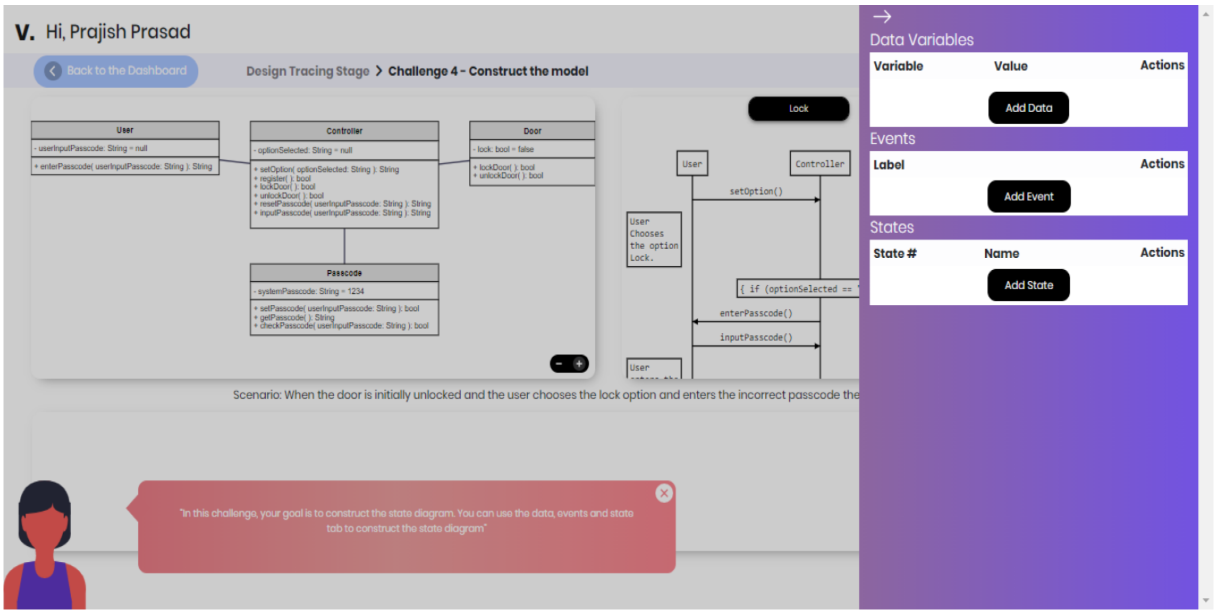
States

State #	Name	Actions
state-0	Initial State	✎
state-1	Set option to register	✎
state-2	Enter passcode	✎
state-3	System saves passcode	✎
state-4	Registers user	✎

Figure 5.22: In Challenge 3, learners add relevant data variables to the state diagram

Challenge 4 - Construct the Model

When learners start challenge 4, they are provided with an empty model. They have to add relevant data variables, events and states to model the given scenario (see Figure 5.23). After successfully completing this challenge, learners advance to the third stage in VeriSIM i.e. the 'Reflection Stage'


V. Hi, Prajish Prasad
Back to the Dashboard
Design Tracing Stage > Challenge 4 - Construct the model
User
- userInputPasscode: String = null
+ enterPasscode(userInputPasscode: String): String
Controller
- optionSelected: String = null
+ setOption(optionSelected: String): String
+ register( ): bool
+ lockDoor( ): bool
+ unlockDoor( ): bool
+ resetPasscode(userInputPasscode: String): String
+ inputPasscode(userInputPasscode: String): String
Door
- lock: bool = false
+ lockDoor( ): bool
+ unlockDoor( ): bool
Passcode
- systemPasscode: String = 1234
+ setPasscode(userInputPasscode: String): bool
+ getPasscode( ): String
+ checkPasscode(userInputPasscode: String): bool
Lock
User
Controller
setOption()
User Chooses the option Lock.
{ if (optionSelected ==
enterPasscode()
inputPasscode()
User
Data Variables
Variable
Value
Actions
Add Data
Events
Label
Actions
Add Event
States
State #
Name
Actions
Add State

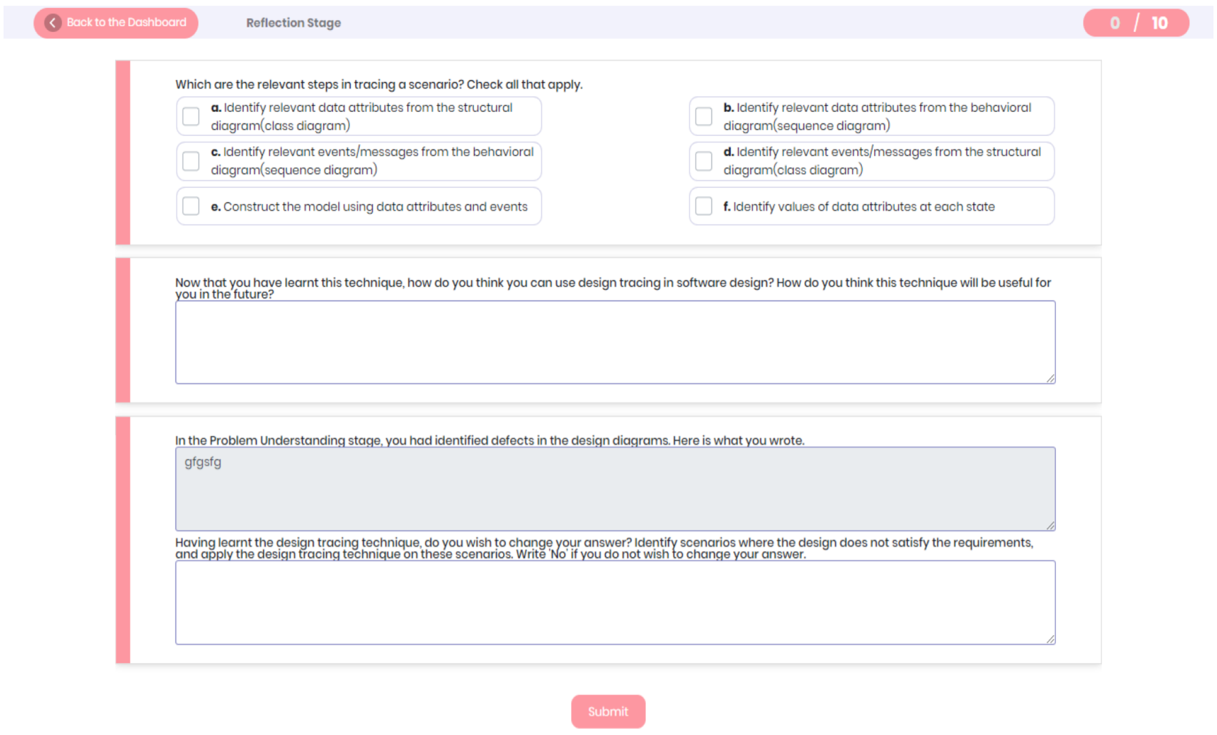
Scenario: When the door is initially unlocked and the user chooses the lock option and enters the incorrect passcode the

"In this challenge, your goal is to construct the state diagram. You can use the data, events and state tab to construct the state diagram"

Figure 5.23: In Challenge 4, learners construct the entire state diagram from scratch

Reflection Stage

In the reflection stage, learners go through reflection activities which asks them to identify relevant steps in tracing a scenario, and how the design tracing strategy will be useful for them in the future. Learners are also presented with the answer they provided to the reflection activity in the ‘Problem Understanding Stage’ i.e. of identifying defects in the design diagrams (see Figure 5.17). They are then asked whether they would like to change their answer to this question. The reflection activity is meant for them to apply the design tracing technique to come up with scenarios which do not satisfy the requirements. Questions in the ‘Reflection Stage’ are shown in Figure 5.24.



Which are the relevant steps in tracing a scenario? Check all that apply.

- [ ] **a.** Identify relevant data attributes from the structural diagram(class diagram)
- [ ] **b.** Identify relevant data attributes from the behavioral diagram(sequence diagram)
- [ ] **c.** Identify relevant events/messages from the behavioral diagram(sequence diagram)
- [ ] **d.** Identify relevant events/messages from the structural diagram(class diagram)
- [ ] **e.** Construct the model using data attributes and events
- [ ] **f.** Identify values of data attributes at each state

Now that you have learnt this technique, how do you think you can use design tracing in software design? How do you think this technique will be useful for you in the future?

In the Problem Understanding stage, you had identified defects in the design diagrams. Here is what you wrote.

gfgsfg

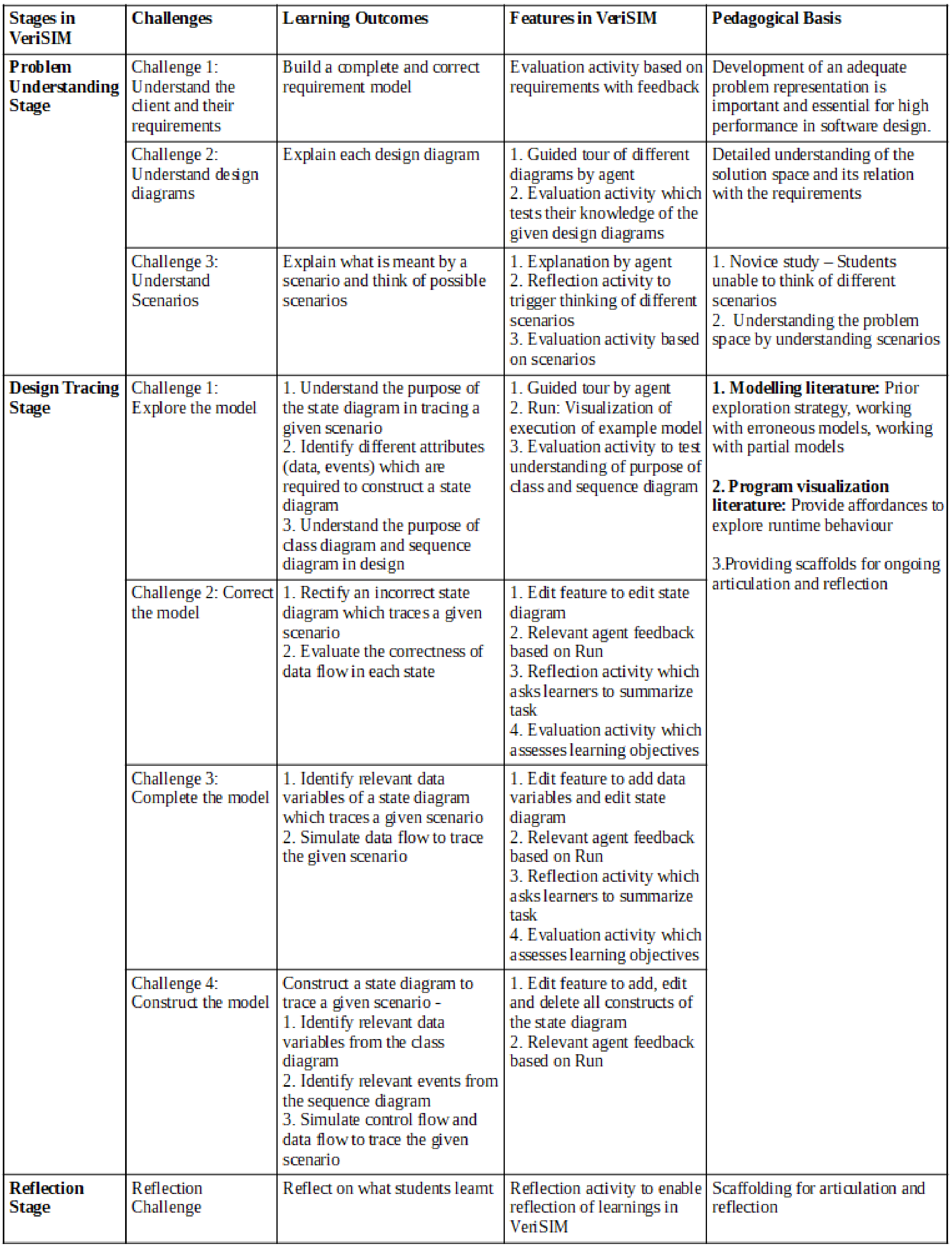
Having learnt the design tracing technique, do you wish to change your answer? Identify scenarios where the design does not satisfy the requirements, and apply the design tracing technique on these scenarios. Write 'No' if you do not wish to change your answer.

Figure 5.24: Questions asked to learners in the 'Reflection Stage'

In the next section, we describe the pedagogical basis of the challenges and stages in VeriSIM and they implement the VeriSIM pedagogy.

5.6 Connecting the Pedagogical Basis to Challenges and Features in VeriSIM

In this section, we discuss how the pedagogical basis connects to various activities in VeriSIM. We describe how the TELE features implement the VeriSIM pedagogy of helping students identify and construct models of various scenarios in the design. A summary of challenges in VeriSIM, and the corresponding features and its pedagogical basis is described in Figure 5.25.



Stages in VeriSIM	Challenges	Learning Outcomes	Features in VeriSIM	Pedagogical Basis
Problem Understanding Stage	Challenge 1: Understand the client and their requirements	Build a complete and correct requirement model	Evaluation activity based on requirements with feedback	Development of an adequate problem representation is important and essential for high performance in software design.
	Challenge 2: Understand design diagrams	Explain each design diagram	1. Guided tour of different diagrams by agent 2. Evaluation activity which tests their knowledge of the given design diagrams	Detailed understanding of the solution space and its relation with the requirements
	Challenge 3: Understand Scenarios	Explain what is meant by a scenario and think of possible scenarios	1. Explanation by agent 2. Reflection activity to trigger thinking of different scenarios 3. Evaluation activity based on scenarios	1. Novice study – Students unable to think of different scenarios 2. Understanding the problem space by understanding scenarios
Design Tracing Stage	Challenge 1: Explore the model	1. Understand the purpose of the state diagram in tracing a given scenario 2. Identify different attributes (data, events) which are required to construct a state diagram 3. Understand the purpose of class diagram and sequence diagram in design	1. Guided tour by agent 2. Run: Visualization of execution of example model 3. Evaluation activity to test understanding of purpose of class and sequence diagram	**1. Modelling literature:** Prior exploration strategy, working with erroneous models, working with partial models **2. Program visualization literature:** Provide affordances to explore runtime behaviour 3.Providing scaffolds for ongoing articulation and reflection
	Challenge 2: Correct the model	1. Rectify an incorrect state diagram which traces a given scenario 2. Evaluate the correctness of data flow in each state	1. Edit feature to edit state diagram 2. Relevant agent feedback based on Run 3. Reflection activity which asks learners to summarize task 4. Evaluation activity which assesses learning objectives	
	Challenge 3: Complete the model	1. Identify relevant data variables of a state diagram which traces a given scenario 2. Simulate data flow to trace the given scenario	1. Edit feature to add data variables and edit state diagram 2. Relevant agent feedback based on Run 3. Reflection activity which asks learners to summarize task 4. Evaluation activity which assesses learning objectives	
	Challenge 4: Construct the model	Construct a state diagram to trace a given scenario - 1. Identify relevant data variables from the class diagram 2. Identify relevant events from the sequence diagram 3. Simulate control flow and data flow to trace the given scenario	1. Edit feature to add, edit and delete all constructs of the state diagram 2. Relevant agent feedback based on Run	
Reflection Stage	Reflection Challenge	Reflect on what students learnt	Reflection activity to enable reflection of learnings in VeriSIM	Scaffolding for articulation and reflection

Figure 5.25: A summary of the pedagogical basis of various features in VeriSIM

The pedagogical basis of challenges in the Problem Understanding Stage is to help learners build an adequate understanding of the requirements and the design diagrams. This understanding is a pre-requisite to trace scenarios in the 'Design Tracing' stage. Literature also emphasises that development of an adequate problem representation is essential for high performance in software design (Sonnentag, 1998). These challenges and the evaluation activities ensure that learners have a detailed understanding of the problem (requirements) and solution (design diagram) space. Literature also identifies two categories of understanding the problem space - Problem comprehension by analysing requirements and problem comprehension by scenarios (Sonnentag, 1998). In the 'Understand Scenarios' challenge, learners are presented with a scenario, and are also triggered to think of different scenarios as they attempt the reflection and evaluation activities. Hence, the challenges in this stage ensure that learners have analysed the requirements and the design, and have examined various scenarios in the design.

In the 'Design Tracing Stage', learners are scaffolded to progressively construct models of scenarios in the design. The challenges in this stage incorporate various pedagogical features of the model-based learning paradigm. First, the VeriSIM interface provides learners with the 'Edit' feature, which enables them to manipulate the state diagram by adding/editing/deleting data, events and states. Second, the 'Run' feature provides learners with appropriate feedback on their models. Hence, these pedagogical features promote the model formulation, refinement, and evaluation practices posited by the model-based learning paradigm.

In the 'Design Tracing Stage', learners progressively learn to construct the state diagram. The underlying theoretical basis of the order of challenges is based on the model progression of activities which we described in Section 5.2.1. In the first challenge of the 'Design Tracing Stage', learners observe the run of the state diagram i.e. the execution of trace of the scenario. In the second challenge, there is an error in the data flow, which the learner has to correct. In the third challenge, the entire data flow has to be traced by the learner. In the fourth challenge, control flow and data flow has to be traced. Each of these modelling activities, such as prior exploration, learning from partial and erroneous models have been shown to be beneficial to students in the context of modelling in science. We have appropriately adapted it to the context of modelling scenarios in a given design.

In the 'Reflection Stage', as well as after each challenge in other stages, students are provided with adequate scaffolds for articulation and reflection. These scaffolds enable them to abstract their learnings from each challenge, and think of how they can apply it for new problems and contexts.

In the next section, we discuss how the activities in VeriSIM are enabling students to develop effective mental models of the design.

5.7 Role of VeriSIM Activities in Developing Effective Mental Models of the Design

As mentioned in Section 4.1, the block model not only characterizes the elements of the software design mental model, but can also inform the pedagogy of activities targetted at developing specific mental model elements of the design. Hence, we have used the mental model elements for design diagrams (which we adapted from the Block model in Section 4.1.2), to inform the design of the stages and activities in VeriSIM. This is summarised in Figure 5.26.

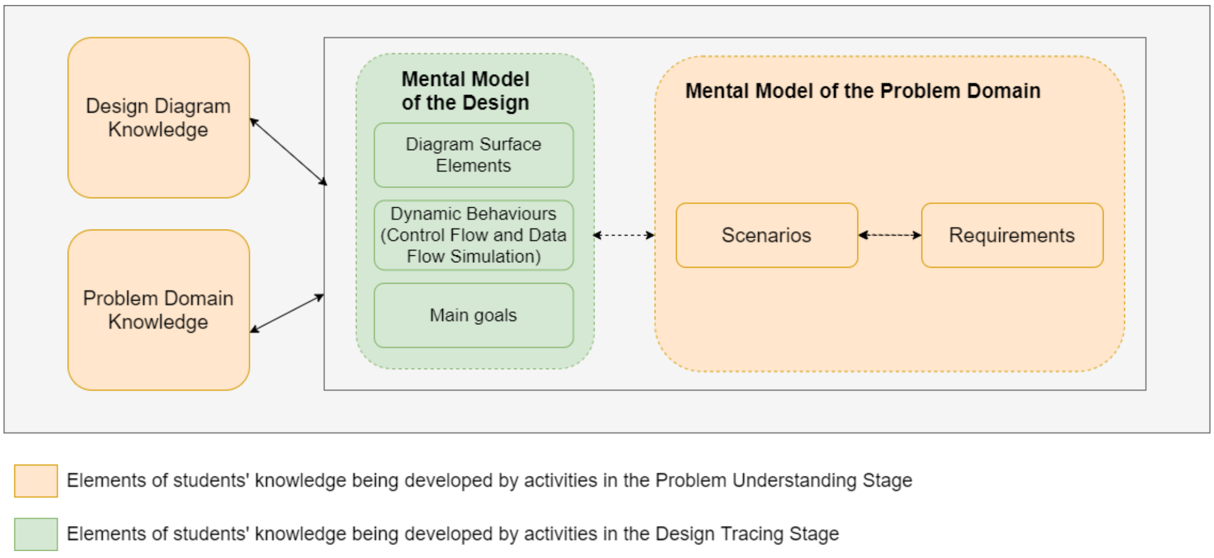

Design Diagram Knowledge
Problem Domain Knowledge
Mental Model of the Design
Diagram Surface Elements
Dynamic Behaviours (Control Flow and Data Flow Simulation)
Main goals
Mental Model of the Problem Domain
Scenarios
Requirements
Elements of students' knowledge being developed by activities in the Problem Understanding Stage
Elements of students' knowledge being developed by activities in the Design Tracing Stage

Figure 5.26: Elements of students' mental model being developed by activities in VeriSIM

The challenges in the 'Problem Understanding Stage' helps students build on their prior design diagram knowledge and problem domain knowledge, as seen in Figure 5.26. In Challenge 1 (Understand the client and requirements), as learners go through the requirements and answer the evaluation questions, they develop an understanding of the problem domain and what is required to be designed. This helps them visualize the door locking system, and simulate how a user will use the system based on the requirements. In Challenge 2 (Understand the software design diagrams), learners are provided with an overview of class and sequence diagrams, which along with the reflection and evaluation activities activates their prior knowledge of the design diagrams and enhances their understanding of the purpose of specific design diagrams in the design. In Challenge 3 (Understand scenarios), learners are introduced to scenarios, and they identify other scenarios in the reflection and evaluation activities. These activities help them develop a deeper understanding of the problem domain, since the scenarios describe various behaviours in the design. As seen in Figure 5.26, the activities in the 'Problem Understanding Stage' fosters development of the mental model of the problem domain, and also triggers their prior knowledge about the problem domain and design diagrams.

Once learners develop an understanding of the problem domain, the challenges in the 'Design Tracing Stage' follow a scenario-centric approach and help students develop a rich mental model of the design. The main goal of each challenge is to construct a state diagram which correctly models the given scenario. Each state in the state diagram contains the current values of variables. Each transition denotes an event and causes a change to the next state. The values in each state can be determined by analysing the class diagram, and transitions can be determined by analysing the sequence diagram. This ensures that learners are developing an understanding of the **diagram structural elements**.

As learners trace a given scenario, they also map a part of the scenario with a given state. This ensures that they identify the **main goals and sub-goals** in the design. Learners tracing the scenario using the state diagram ensures that they are able to **simulate the control flow** while making the transitions, and are able to **simulate the data flow** by explicitly mentioning the values of variables in each state.

Thus, as learners progressively construct the model for the given scenarios, they develop a deeper understanding of the diagram structural elements, such as the data members and functions in the class diagram, the goals and sub-goals in the sequence diagram, and simulate the control flow and data flow across the class and sequence diagram for the given scenarios.

We believe these activities and pedagogical features in VeriSIM foster development of the elements of students' mental models, thereby enabling them to effectively evaluate a software design against the given requirements.

5.8 Summary

In this chapter, we summarized findings from the analysis of literature and our studies with novices. Based on these findings, we proposed the VeriSIM pedagogy, which can scaffold students to effectively perform software design evaluation. The key idea of the VeriSIM pedagogy is to scaffold learners to identify and construct models of scenarios in the design. The underlying theoretical basis of the pedagogy is adapted from the model-based learning paradigm. We provided arguments for how model-based learning practices and scaffolds can be adapted for software design evaluation. We operationalised the pedagogy into the VeriSIM learning environment, which incorporated features of model construction, revision, visualization, and evaluation. We also described the pedagogical basis of various features in VeriSIM and how they contribute in developing effective mental models of the design in students.

In the next chapter, we investigate the effectiveness of VeriSIM, and how it helps students to perform software design evaluation.

Chapter 6

Evaluation of the VeriSIM Learning Environment

In this chapter, we describe Study 2, which we conducted to examine the effectiveness and usefulness of the VeriSIM learning environment.

6.1 Research Questions

The activities and features in the VeriSIM learning environment enable learners to simulate dynamic behaviours (control flow and data flow) in the design by tracing a given scenario. Tracing scenarios can help learners identify defects in the design by identifying scenarios which do not satisfy the requirements. Through this study, we investigate the effects of VeriSIM in students' ability to simulate dynamic behaviours, and in their ability to identify defects in the design diagrams. The research questions guiding this study are as follows:

- RQ 2.1: What are the effects of VeriSIM in students' ability to simulate dynamic behaviours in the design?

- RQ 2.2: What are the effects of VeriSIM in students' ability to identify defects in the design?

6.2 Study Procedure

Study 2 was conducted with 86 final year (fourth year) computer engineering and information technology engineering students (48 male and 38 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. 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 understanding of object-oriented design, class and sequence diagrams. The registration form also contained the consent form. Participation in the study was completely voluntary, and participants could withdraw from the study at any point during the study. (The registration form along with the consent form is given in Appendix C).

All participants had undergone a Software Engineering course in the previous semester, and hence were familiar with class diagrams and sequence diagrams. All participants were comfortable working on a computer. Due to space constraints, all the participants could not be accommodated in a single room at once. Hence, the study was divided into 2 sessions - one in the morning and the other in the afternoon. 60 students (30 male and 30 female) participated in the morning session. 26 students (18 male and 8 female) participated in the afternoon session.

A summary of the study procedure is shown in Figure 6.1. The study took place in a computer lab. I, along with two research interns were present during the study. I initially explained the main goals of the workshop and the activities participants will perform during the workshop. After this introduction, participants solved a pre-test. After solving the pre-test, they interacted with VeriSIM. Participants worked individually with VeriSIM at their own pace. I did not interfere while they interacted with VeriSIM, but I was available to answer any doubts which arose during their interaction. After participants finished interacting with VeriSIM, they solved a post-test. They then filled a feedback form, which contained usability and other feedback questions. After the study, I and one of the research interns conducted a semi-structured focus group interview in each session (8 participants (5 male and 3 female) in the morning session and 5 participants (2 male and 3 female) in the afternoon session). Based on the academic details filled by participants in the registration form, students who had varied prior academic performance were chosen as the participants of our focus group interview. On an average, participants took 1.5 hours to interact with VeriSIM. Each study session (pre-test, VeriSIM, post-test, feedback and focus-group interview) took around 3 hours.

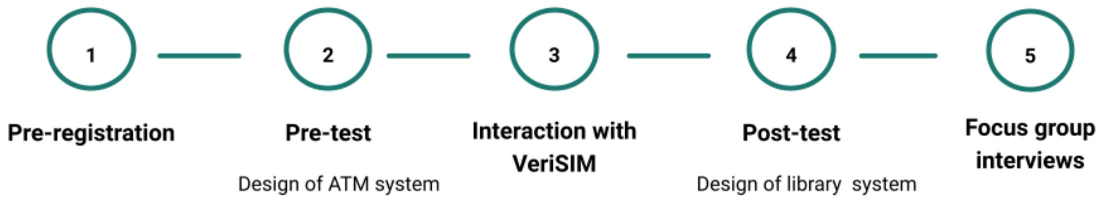


Figure 6.1: Summary of the procedure for Study 2

6.3 Data Sources

The data sources for the study are as follows:

1. Pre-test - In the pre-test, participants were provided with the requirements and design of an ATM system (1 class diagram and 3 sequence diagrams). They had to attempt 2 questions. In the first question, they were provided with the following incomplete scenario - *"The user has a balance of Rs.5000 in his account, enters the correct PIN and withdraws Rs.500"*. The participant had to explain the sequence of steps and changes that occur in the system from the beginning to the end on execution of this scenario. In the second question, they had to identify defects in the given diagrams. The question statement is as follows - *"Identify defects (if any) in the following design diagrams based on the requirements. For each defect, provide a logical explanation of why you think it is a defect."*. (The pre-test for Study 2 can be found in Appendix D.1)

2. Post-test - In the post-test, participants were provided with the requirements and design of an online library system (1 class diagram and 3 sequence diagrams). The questions were similar to the pre-test. In the first question, they had to trace the following scenario using the design tracing technique - *"When a new user selects the register option, and enters*

the username and password, the user registers successfully.”. While tracing, students had to identify relevant data variables, events, and change of state as a result of the execution of the above scenario. In the second question, they had to identify defects in the given diagrams.

We refer to the first question as the **‘tracing question’**, and the second as the **‘evaluation question’**. Questions in the pre-test and post-test tested students’ ability to identify relevant data variables, events and changes in state of these variables on execution of the scenario (RQ 2.1) and their ability to identify defects in the design diagrams (RQ 2.2). (The post-test for Study 2 can be found in Appendix D.2)

3. Semi-structured focus group interview - A semi-structured focus-group interview was conducted with students in order to gather their perceptions about the VeriSIM pedagogy, and what they learnt after interacting with VeriSIM. The list of questions asked are provided in Table 6.1. By answering these questions, we wanted students to elicit how VeriSIM helped them simulate the control flow and data flow of scenarios, and how this helped them evaluate the software design.

4. Feedback form - The feedback form asked participants to rate their confidence in class and sequence diagram understanding after interacting with VeriSIM. There were also open-ended questions asking them what they learnt in the workshop and also the features of VeriSIM which they found useful. (The feedback form for Study 2 can be found in Appendix D.3)

Table 6.1: Focus Group Interview Questions for Study 2

Focus Group Interview Questions
1. What are the main things you learnt from the workshop?
2. What according to you is design tracing?
3. What is the usefulness of constructing the state diagram? Is there value to it?
4. How do you think what you learnt in this workshop is useful for you in the future?

6.4 Data Analysis

We combined insights from students' responses to the tracing and evaluation questions, as well as their perceptions captured in the semi-structured interviews and feedback forms, to answer research questions RQ 2.1 (Students' ability to simulate dynamic behaviours) and RQ 2.2 (Students' ability to identify defects in the design). A summary of the data sources and analysis used to answer the RQs is shown in Table 6.2.

Table 6.2: Data source and analysis methods for Study 2

Research Question	Data Source	Data Analysis Method
RQ 2.1: Effects of VeriSIM in students' ability to simulate dynamic behaviours in the design	Tracing question in pre-test and post-test	1. Analysis using rubric 2. Thematic analysis of interview transcripts & feedback forms
RQ 2.2: Effects of VeriSIM in students' ability to identify defects in the design	Evaluation question in pre-test and post-test	1. Content analysis 2. Thematic analysis of interview transcripts & feedback forms

6.4.1 Analysis of tracing questions

I analysed the tracing question in the pre-test and post-test in order to examine how students are simulating the dynamic behaviour of a given scenario (RQ 2.1). Out of 86 participants, 11 did not stay till the end of the study and did not attempt the post-test. Hence I analysed only the remaining 75 responses in the pre-test and post-test. I designed a rubric in order to evaluate student responses in the pre-test and post-test (Table 6.3). The rubric contains three criteria which are needed to trace a given scenario: (1) identifying relevant data variables, (2) identifying events and (3) simulating change of state. Three levels (Missing, Almost and Target) have been chosen based on how many relevant data variables, events and state changes students are able to identify for a given scenario. For data variables, students are required to identify all relevant and correct variables from the class diagram. For events, students are required to identify all relevant events from the scenario and sequence diagrams, with explicit distinction between events specified. For states, students are required to simulate change of state with relevant variable value pairs. In the pre-test, participants were not aware of the design tracing strategy. Hence most of the answers were written in the form of sentences. In the post-test, most participants drew state diagrams to describe the execution of the scenario. Both these types of representations can be evaluated using the rubric.

Table 6.3: Rubric for evaluating tracing questions in the pre-test and post-test

	Missing (0)	**Almost (1)**	**Target (2)**
Identifying data variables	Missing all relevant data variables from the class diagram	Identifies some relevant variables Adds irrelevant data variables	Identifies all relevant data variables No irrelevant data variables added
Identifying relevant events	Missing all relevant events Separation of events is not seen	Identifies some relevant events Identifies some irrelevant events Separation of events is unclear	Identifies all relevant events No irrelevant events included Separation of events is clear
Simulating state change	No mention of state change of variables	State change of some variables are mentioned with variable-value pairs	State change of all variables are clearly mentioned with correct variable-value pairs

In order to establish content validity of the rubric, I discussed the criteria and levels of the rubric with another researcher who was not involved in the research study. Three answer sheets were used to refine wordings of certain criteria. In order to establish reliability of the grading using the rubric, another rater graded a subset of the questions. The rater and I independently graded 2-3 questions. We then discussed how our scoring was done, and differences in scoring was resolved. We then independently graded 10 questions. Inter-rater reliability of the rubric was established by calculating Cohen's Kappa (Cohen, 1960) for each criteria. I then graded the remaining students' questions based on the rubric. This procedure was done for both the pre-test and the post-test.

The Cohen's kappa for each criteria in the pre-test is as follows: $Kappa_{Data} = 1, Kappa_{Event} = 0.83, Kappa_{State} = 0.81$. The Cohen's kappa for each criteria in the post-test is as follows: $Kappa_{Data} = 0.85, Kappa_{Event} = 0.85, Kappa_{State} = 0.85$. (The range from 0.81˘1.00 can be interpreted as almost perfect agreement).

6.4.2 Analysis of evaluation questions

I analysed the 'evaluation' question in the pre-test and post-test to examine how students are identifying defects in the design (RQ 2.2). We used the content analysis method (Baxter, 1991) (described in Section 3.5.2) to come up with categories for the defects identified. This analysis is similar to what was done in Study 1a. The steps which we followed to analyse the data are 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. I 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. Many students listed multiple defects. I identified 168 answers from the pre-test and 71 answers from the post-test.

2. **Descriptive coding** - I assigned a descriptive code to each sentence. The objective of descriptive coding is to avoid any prejudices or preconceptions and stay close to the data. For example, for the following student answer - *"After a user returns a book, in the return function, the no. of books attribute issued by the user must be updated and the issueStatus and the issuedBy attributes of the book must be reset"*, the descriptive code assigned is *'When user returns a book, a. noOfBooks attribute should be updated and b. issueStatus and issuedBy attributes should be reset"*

3. **Generating categories** - I then inferred categories based on the patterns, explanations and relationships between the descriptive codes. The categories reflected the meaning inferred from the descriptive codes and can explain larger segments of data. For example, the descriptive code mentioned above refers to scenarios which are not present in the design diagrams, and hence do not satisfy the requirements. Hence the category assigned to the above descriptive code is - *"Identify scenarios which do not satisfy requirements"*

We then compared categories of student responses in the pre-test and post-test to examine differences in the types of defects students identified.

6.4.3 Analysis of interview transcripts and feedback form responses

We analysed the focus group interview transcripts and the feedback form responses using a thematic analysis approach (Braun and Clarke, 2012) (described in Section 3.5.3). I transcribed the focus group interviews. Another researcher and I independently coded the transcript, and came up with initial codes. We then individually checked for emerging patterns and categorised the codes into themes. After independently coming up with themes, the researcher and I discussed, reviewed and defined the themes and came to an agreement on the final themes.

I followed a similar thematic analysis procedure for responses in the feedback form. The question asked in the feedback form was - *"What are the main things you learned from the workshop?"*. We received 75 participant responses from the feedback form.

The themes which emerged from the focus group interviews and feedback form responses provided student perceptions of how the VeriSIM pedagogy helped them in simulating dynamic behaviour of scenarios, such as control flow and data flow (RQ 2.1), and how it helped them identify defects in the design (RQ 2.2).

6.5 Findings

6.5.1 RQ 2.1: Students' ability to simulate dynamic behaviours

Findings from analysis of tracing question

We answer our first research question by analysing the pre-test and post-test answers. A summary of the results is shown in Table 6.4. In the pre-test, the mean scores for the data, event and state criteria are $0.47(SD = 0.7), 1.16(SD = 0.62)$ *and* $0.44(SD = 0.68)$ respectively. In the post-test, the mean scores for the data, event and state criteria are $0.95(SD = 0.87), 1.28(SD = 0.88)$ *and* $0.84(SD = 0.84)$ respectively. We performed a paired t-test between the pre-test and the post-test scores. The results show a significant increase in the post-test score ($M = 3.07, SD = 2.09$), compared to the pre-test score ($M = 2.07, SD = 1.7$),

$t(74) = -4.03, p < 0.001$

We also performed a paired t-test between the pre-test and post-test scores for each criteria. There is a significant increase in the post-test score for the data criteria ($t(74) = -4.38, p < 0.001$) and state criteria ($t(74) = -3.91, p < 0.001$), but not for the event criteria ($t(74) = -1.05, p = 0.17$). We hypothesize that participants were able to identify events better than data or state, because identifying events can be done by observing the scenario itself as well as a superficial reading of the relevant sequence diagram. Identifying data variables and constructing the state diagram requires a more detailed analysis of the class diagram as well as the sequence diagram.

Table 6.4: Participants' scores for 'tracing question' in pre-test and post-test

Criteria for tracing scenarios	**Pre-test Mean(SD)**	**Post-test Mean(SD)**	**Paired t-test (p value)**
Identifying relevant data variables	0.47(0.70)	0.95(0.87)	0.00
Identifying relevant events	1.16(0.62)	1.28(0.88)	0.17
Simulating state change	0.44(0.68)	0.84(0.84)	0.00
Total Score (out of 6)			
	2.07(1.70)	3.07(2.09)	0.00

Figure 6.2 shows the bar graph of participants score in the data, event and state change criteria for pre-test and post-test. In the post-test, around 60% of the participants were able to identify almost/all the relevant variables (45/75) and state changes for the given scenario (42/75), as opposed to only around 35% in the pre-test (data - 26/75, state change - 25/75). However, more students were able to identify almost/all events in the pre-test (66/75), as opposed to the post-test (54/75).

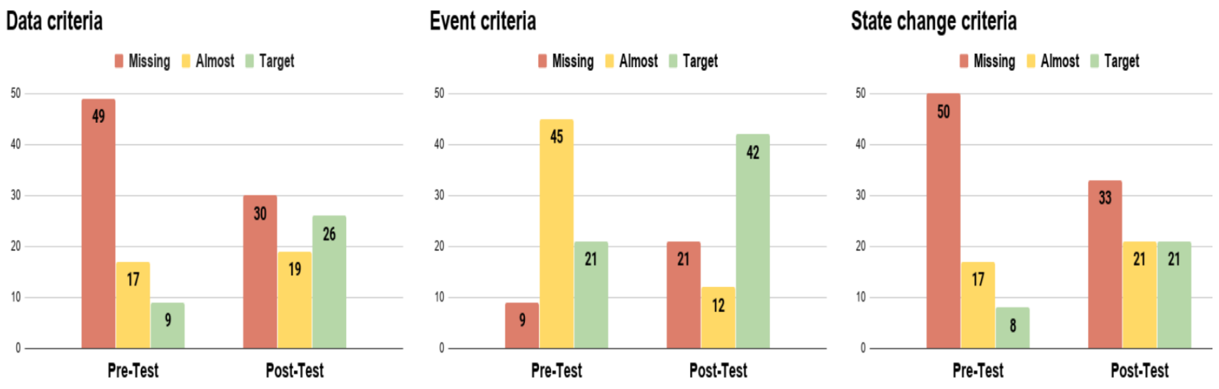


Figure 6.2: Bar graph of pre-test and post-test scores for the ‘tracing question’

Themes from interviews and feedback forms

Themes emerging from the interview transcripts and feedback forms indicate that the VeriSIM pedagogy helped students **simulate the control flow and data flow of scenarios in the design**. For example, consider the following instance where a participant is describing the execution of a scenario - *“We would have to first understand the sequence diagram system, to put forth values to put if its false, or if it is true, and there was one where the default values were given... Let’s say that at the first stage, .. what will be the default value of that locked function, it will always be false, we need to put that, so just understanding how all of these connect..”* In this example, we see that the participant is simulating possible values that the locked function returns, and also how the function is related to the sequence diagram.

Students also reported that design tracing helped them **understand the logical flow of the design**, and that the state diagram helped them understand **how the sequence of execution goes logically from one state to another**. For example - *“What design tracing helped was how to think in a flow, like a sequence. Given a scenario, ... it’s basically a set of steps,I do certain things, after that I proceed to this thing, after that I proceed to this thing.”*

Design tracing helped them **understand the relationship between different design diagrams**. For example, a participant said - *“I realised how they are all correlated, and how I need to traverse through that state diagram, I need to know the sequence diagram very well, or the class diagram very well. So it was useful to learn how it goes step by step.”* This shows that students had to understand the class and sequence diagram in order to complete the state diagram.

6.5.2 RQ 2.2: Students' ability to identify defects

Findings from analysis of the evaluation questions

Based on the content analysis of the answers for the evaluation questions in the pre-test and post-test, we identified categories similar to the categories in Study 1a (Section 4.2.6). The categories were - 'Identify scenarios which do not satisfy the requirements', 'Change existing functionalities and requirements', 'Add new functionalities in the design', 'Change data types, functions and structure of the class diagram' and 'Blank responses' and 'No defects'.

We categorized each student response based on the above categories. Differences in the category responses of students in the pre-test and post-test are shown in Figure 6.3. The number of responses in the post-test was lesser compared to the pre-test (Pre-test: 145 responses vs Post-test: 71 responses).

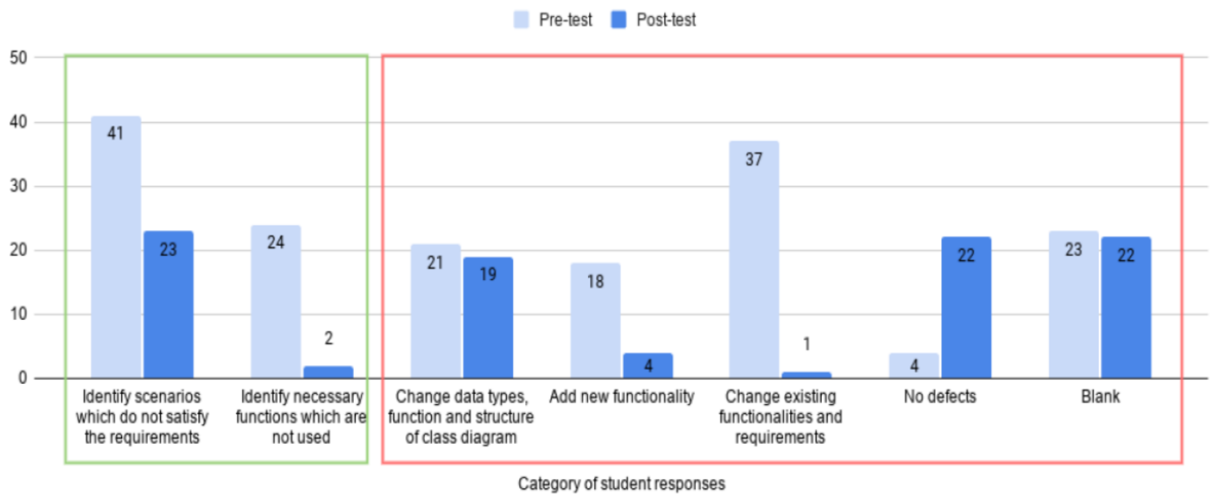


Figure 6.3: Bar graph of pre-test and post-test category responses for 'evaluation' question

Since there is a large difference between the number of responses in the pre-test and post-test, we calculated the percentage of responses in each category (Figure 6.4). This enabled us to compare the responses in each category for the pre-test and post-test. As seen in Figure 6.4, the percentage of student responses indicating identifying relevant scenarios which do not satisfy the requirement are similar in the pre-test (24.4%) and post-test (24.7%). The percentage of student answers indicating adding new functionalities reduced in the post-test (10.7%) as compared to the pre-test (4.3%). The same was true for the 'Changing existing functionalities and requirements' category (Post-test - 22% vs Pre-test - 1.1%). However, the percentage of students who explicitly specified that there were no defects increased in the post-test (23.7%) as compared to the pre-test (2.4%). The same holds true for students who left their answer sheet blank (Post-test - 22% vs Pre-test - 1.1%). In the next section, we discuss the probable reasons behind the reduced number of responses, as well as the percentage increase in number of blank and no defect categories.

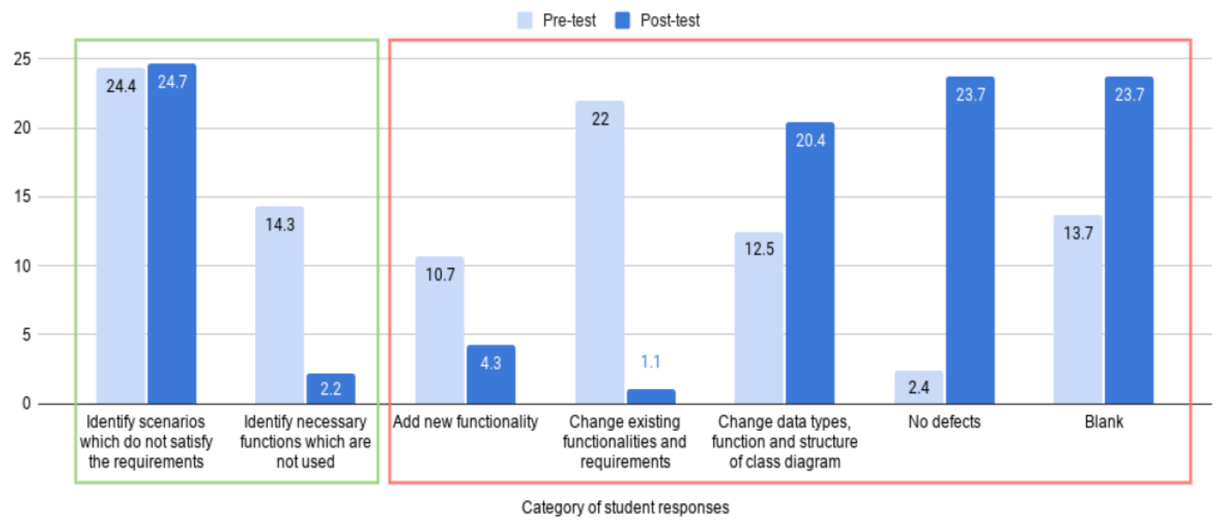


Figure 6.4: Bar graph of pre-test and post-test percentage responses for 'evaluation' question

Themes from interviews and feedback forms

Students perceived that VeriSIM helped them identify scenarios and detect defects in the system. Modeling different scenarios using the state diagram helped them **detect errors in the sequence diagram** and **identify user requirements not considered in the sequence diagram**. For example, one participant said that he considered scenarios similar to test cases. When asked about the importance of scenarios, a participant said - *"how do we validate the machine? Suppose we have to enter an age, and we have to identify, suppose the person enters like minus, negative numbers. So that .. those cases are not there only."*. In this example, we see that the student is able to explain how scenarios can help in identifying missed conditions and checks. He also gave another example from the question in the post-test - *".. in the post-test, there was a case where the sequence diagram missed whether a book was already issued or not. So that was like a user requirement, which was not considered in the sequence diagram."*

6.6 Discussion and Reflections

6.6.1 Improvements in simulating dynamic behaviours in the design

Findings of RQ 2.1 show that explicitly teaching students to trace scenarios leads to an improvement in their ability to simulate dynamic behaviours in the design. A statistically significant result in the tracing question indicates improvements in students' tracing ability. This finding also corresponded to their perceptions, that the VeriSIM pedagogy helped them simulate control flow and data flow of scenarios, understand the logical flow in the diagrams, and also understand the relationship between different diagrams.

6.6.2 Change in attitude towards software design

An insight which came out quite often in the focus group interviews, was the change in students' attitude towards creating software designs after interacting with VeriSIM. Although students were aware of how to create software designs, they did not use it explicitly while designing systems for their projects, and did not give it a lot of importance. Interactions with VeriSIM helped them understand the **importance of creating design diagrams at the start**, before coding (Example: *"I was the kind of student I used to make the project and then design the (system).. For this I got to know actually, I got to know the advantages of designing a class diagram before actually starting with the project."*). Students also realised that creating designs also helps them **manage time**, as well as **facilitate knowledge transfer among team members** (Example: *"But I get why it's so necessary in bigger teams, ... if one person leaves and another person needs to come, he needs to be in touch with what's happening, so show him these diagrams, he will figure it out"*).

We also see that students showed an increased confidence in their understanding of the purpose of class diagrams and sequence diagrams after interactions with VeriSIM. In the registration and feedback form, there were two Likert scale questions (ranging 0-5) asking students to rate their confidence in understanding the purpose of class diagrams and sequence diagrams in design (see Appendix D.3 for the feedback form). A Wilcoxon signed-rank test showed that there was a statistically significant improvement in students' perception of their confidence in understanding the purpose of class diagrams (Z = -4.26, p <0.001) and sequence diagrams (Z = -4.42, p <0.001).

Based on these findings, we believe that interactions with VeriSIM can help students overcome common reported difficulties of understanding the purpose and relationship of different diagrams as discussed in the Chapter 2 (Section 2.2).

6.6.3 Students have difficulty in identifying alternate scenarios

Students also reported that design tracing helped them identify missing features and errors in the design. However, this does not correspond to their performance in the evaluation question in the post-test. Firstly, there was a reduction in the number of responses in the post-test as compared to the pre-test (168 vs 71). A plausible explanation for this performance is due to fatigue in participants. The study lasted for around 3 hours, and the 'identifying defects' question was the last question in the post-test. Based on this finding, *we decided to make students interact with VeriSIM and solve the post-test over multiple days, in order to avoid fatigue.*

Secondly, we did not see an increase in the number of responses which identified scenarios not satisfying the requirements. Although VeriSIM improved students' ability to trace the given scenarios (RQ 2.1) and students perceived its usefulness in evaluating designs, they could not identify scenarios in the design which do not satisfy the requirements. Our hypothesis was that activities in VeriSIM would help them identify alternate scenarios. For example, in the 'Problem Understanding Stage' of VeriSIM, students were introduced to what scenarios are, and had also attempted a question asking them to identify scenarios in the design. The analysis of students' answers to this question also show that they identified relevant scenarios in the design. In the 'Design Tracing Stage', students had to trace scenarios which were already provided to them. Our assumption was that identifying scenarios in the 'Problem Understanding Stage', and tracing the given scenarios in the 'Design Tracing Stage' would trigger students to identify and trace alternate scenarios in the design to determine which scenarios do not satisfy the requirements. However, based on students' responses in the 'Reflection Stage' in VeriSIM and pre-post differences in the evaluation question, this assumption does not hold. In the ‘Reflection Stage’ of VeriSIM (Figure 5.24), learners were given the opportunity to change their answer to the question of identifying defects in the design, now that they had learnt about identifying and tracing scenarios. However, we found that most learners did not change their answer and did not identify and trace alternate scenarios. The post-test answer-sheets also did not show applications of design tracing in the evaluation question. We believe that students have difficulty in understanding how to apply the design tracing strategy for evaluating design diagrams. *This gives us indicators that students need explicit help to generate and identify scenarios which do not satisfy the requirements.*

These findings can also help us identify what elements in students’ mental models need to be improved, based on the mental model elements for design diagrams. This has been summarised in Figure 6.5. Activities in VeriSIM enabled students to build effective mental models of the design. By going through the model progression activities in the ‘Design Tracing Stage’, students were able to identify the main goals of a given scenario, identify appropriate diagram structural elements (such as data variables and events), and simulate the control and data flow of the given scenario. Findings from the post-test also show that students were able to *model an already given scenario*. However, they face difficulties in *generating scenarios which do not satisfy the requirements.* Hence, the pedagogy needs to be revised, to include explicit scaffolds to help students identify scenarios which do not satisfy the requirements. They can then model these identified scenarios by tracing the control flow and data flow of the scenario’s execution using the design tracing strategy.

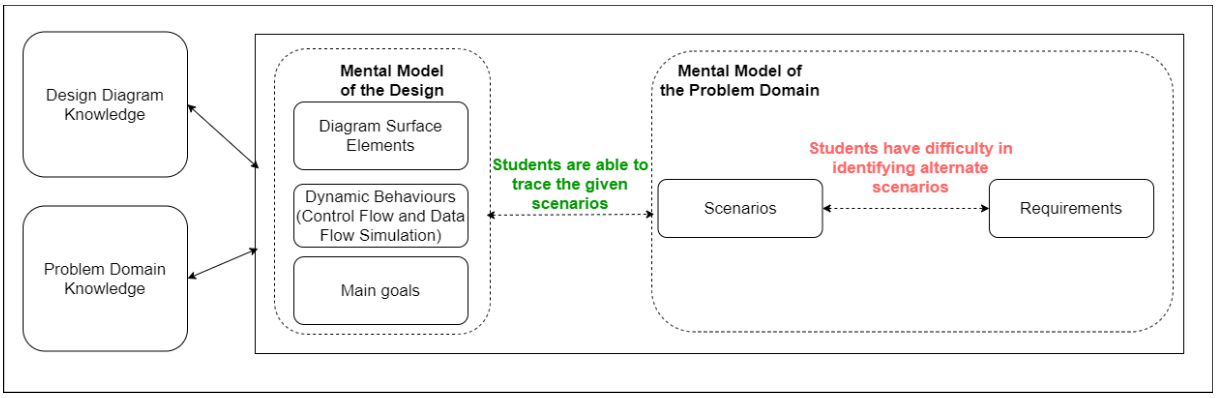

Design Diagram Knowledge
Problem Domain Knowledge
Mental Model of the Design
Diagram Surface Elements
Dynamic Behaviours (Control Flow and Data Flow Simulation)
Main goals
Students are able to trace the given scenarios
Mental Model of the Problem Domain
Students have difficulty in identifying alternate scenarios
Scenarios
Requirements

Figure 6.5: Connecting findings of Study 2 to the mental model for design diagrams

6.7 Summary

In this chapter, we investigated the effectiveness of VeriSIM in helping students in the software design evaluation process. Our findings show that students show improvements in simulating dynamic behaviours in the design, but face difficulties in identifying alternate scenarios which do not satisfy the requirements. These findings and reflections feed into the research directions we explored in the next DBR Cycle (Table 6.5).

Table 6.5: Research directions for the next cycle

Findings and inferences from Study 2	**Research directions for next cycle**
Students show improvements in simulating dynamic behaviours in the design	Delve deeper into what features in VeriSIM are leading to improvements in learning
Students have difficulty in identifying alternate scenarios which do not satisfy the requirements	Refine the VeriSIM pedagogy and provide explicit scaffolds for identifying scenarios in the design

In Chapter 8, we investigate what features in VeriSIM are contributing towards learning of software design evaluation. In the next chapter, we refine our pedagogy by providing explicit scaffolds to help students identify relevant scenarios violating the given requirements.

Chapter 7

Redesign and Evaluation of VeriSIM

In this chapter, we build on the reflections from the findings of the previous DBR Cycle. To recap, the previous chapters (Chapter 4 - Chapter 6), corresponds to the work done in DBR Cycle 1. As seen in Figure 7.1, in the first DBR Cycle, we reviewed literature and conducted studies with students to identify difficulties they faced during software design evaluation (Problem Analysis and Exploration). These insights led to the design of the VeriSIM pedagogy and its operationalisation into the VeriSIM TELE (Solution Design and Development). We investigated the effectiveness of the TELE for evaluating software design diagrams against the given requirements. Although we observed a significant difference in students' ability to trace an already given scenario, they had difficulties in identifying alternate scenarios which violated the given requirements. (Evaluation and Reflection).

These reflections from DBR Cycle 1 feed into the next cycle. In DBR Cycle 2, we analyse the problem, and revise the pedagogy in order to address these difficulties in students. We then conduct studies based on the revised pedagogy and measure its effectiveness.

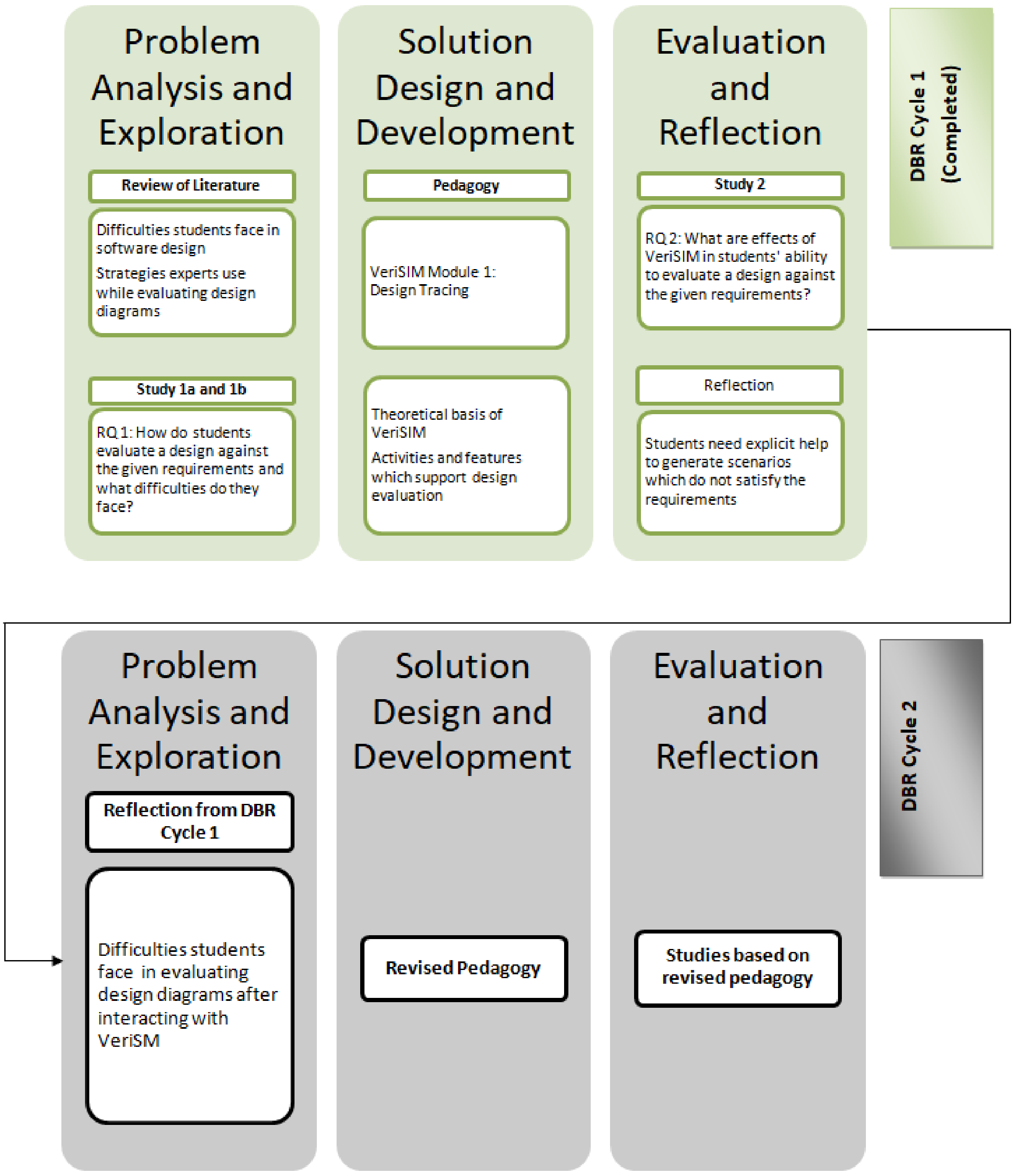

Problem Analysis and Exploration
Review of Literature
Difficulties students face in software design
Strategies experts use while evaluating design diagrams
Study 1a and 1b
RQ 1: How do students evaluate a design against the given requirements and what difficulties do they face?
Solution Design and Development
Pedagogy
VeriSIM Module 1: Design Tracing
Theoretical basis of VeriSIM
Activities and features which support design evaluation
Evaluation and Reflection
Study 2
RQ 2: What are effects of VeriSIM in students' ability to evaluate a design against the given requirements?
Reflection
Students need explicit help to generate scenarios which do not satisfy the requirements
DBR Cycle 1 (Completed)
Problem Analysis and Exploration
Reflection from DBR Cycle 1
Difficulties students face in evaluating design diagrams after interacting with VeriSM
Solution Design and Development
Revised Pedagogy
Evaluation and Reflection
Studies based on revised pedagogy
DBR Cycle 2

Figure 7.1: A recap of the DBR Cycles in this thesis

7.1 Theoretical basis of VeriSIM redesign

In this cycle, our main goal is to provide explicit guidance to students in identifying scenarios in the design. We can look at this goal from the perspective of model-based learning. In order to model scenarios in the design, students need to understand "what" to model, which in this case, is identifying different scenarios in the design. Once they identify different scenarios in the design, the design tracing strategy described in Chapter 5, can address the question of "how" to model the identified scenarios.

What should students do to identify relevant scenarios? They should be able to analyse the given requirements in an effective manner. An understanding of the requirements can help them (1) Generate scenarios from the requirements and (2) Compare the generated scenarios with scenarios in the design.

Analysing requirements is an essential practice of software designers as well. During the initial phases of software design, designers identify and simulate different scenarios which helps them understand and also infer new requirements (Guindon, 1990). Hence, techniques and strategies which software designers use in understanding and analysing requirements can be used to train students as well. We draw the theoretical basis of our proposed strategy from software engineering literature, specifically from cognitive mapping techniques in requirement analysis.

7.1.1 Cognitive mapping techniques in requirement analysis

In the initial phases of software design, designers represent and validate requirements which they have received from various stakeholders. They represent the requirements using conceptual models (Siau and Tan, 2005a), and validate these requirements in order to ensure that it is correctly specified. Most often, requirements analysis is a highly complex task, owing to the ill-structured nature of the initial design phase. There are various tools and strategies which have been used to analyse requirements (Siau and Tan, 2008). Cognitive mapping is a strategy which has been used widely in requirements analysis (Montazemi and Conrath, 1986; Sheetz and Tegarden, 1998). It is primarily used to overcome cognitive problems and facilitate understanding among stakeholders in the development of information systems (Siau and Tan, 2005b).

Cognitive mapping is a technique used to identify subjective beliefs and to portray these beliefs externally (Siau and Tan, 2005a). In cognitive mapping, meaningful concepts from the given domain are extracted. The relationships between these concepts are also specified which are represented in some kind of visuospatial layout (Siau and Tan, 2005b). For example, Figure 7.2 represents the cognitive map which describes the requirements for a learning management system. The map shows the various actors (such as student and instructor), and describes features which are required in the system. In the process of creating the cognitive map, a designer is also able to relate different concepts in the requirements, such as the 'Grade' and 'Course' concepts. Thus, as a designer creates and refines this visual representation, they are able to develop a better understanding of the requirements, and also establish relationships between them.

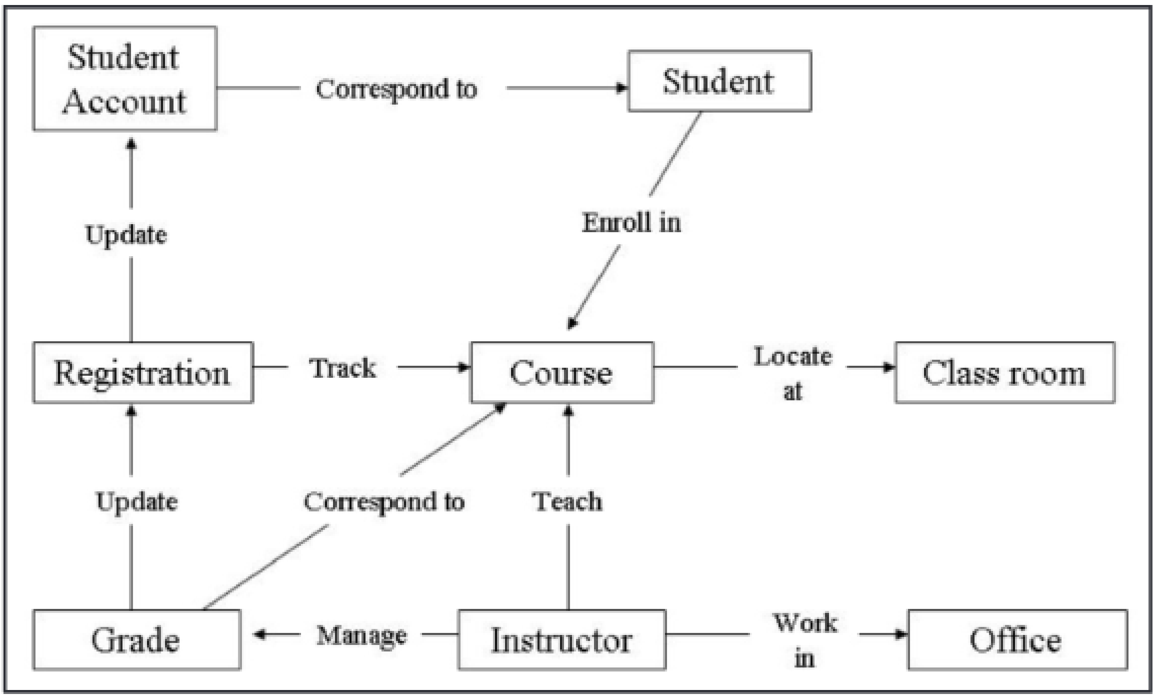


Figure 7.2: An example of a cognitive map

There are primarily three types of cognitive maps - (1) Causal maps - which focus on the cause-effect structure of concepts, (2) Semantic maps - which organize a number of sub-ideas around a main idea and (3) Concept maps - which create semantic-rich links among various concepts. Cognitive mapping has shown to have several advantages. These mapping strategies can focus attention, highlight priorities, supply missing information and reveal gaps in reasoning (Siau and Tan, 2005b). It can be used as a modelling language to conceptually model the given requirements, and also verify that the requirements have been accurately represented (Siau and Tan, 2006).

7.1.2 Adapting cognitive mapping to software design diagram evaluation

While cognitive mapping has been used to analyse requirements, we adapt this strategy as a means to train students in identifying scenarios. In the case of evaluating design diagrams, the requirements are already provided. In the proposed strategy, for each requirement, students construct a tree-like visual representation. They identify sub-goals for each requirement, and identify different possibilities for each sub-goal. This representation which they construct can aid them in identifying relevant scenarios which are not satisfying the given requirements. We explain this strategy in detail in the next section.

7.2 Solution: Scenario Branching Strategy

In this section, we describe the scenario branching strategy, which is an adaptation of cognitive mapping techniques used in requirement analysis. The scenario branching strategy helps students identify scenarios for each requirement in the design, thereby enabling them to identify relevant scenarios which do not satisfy the requirements. We explain the scenario branching strategy with an example, based on the requirements and design of the ATM system (Section 4.2.2). One of the requirements in the ATM system is: *"A user with a valid account can register his/her ATM 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."* For each requirement, learners go through four key steps.

1. **Identify subgoals in the requirement -** A subgoal is a part of the requirement which corresponds to the execution in the system. Identifying subgoals require learners to divide the requirement into logical parts and examine each part with the given design diagrams.

For example, the subgoals for the given example requirement are (1) User with valid account (2) Sets a PIN if a PIN hasn't been set yet and (3) PIN should be of length 4 and should contain only numbers.

2. **Identify relevant variables and different possibilities of these variables -** For each subgoal, learners identify relevant data variables and simulate different values for each of these variables. They construct a scenario tree which captures these subgoals and different values for each of these variables. The data variables and their values are specified inside the nodes, and the links correspond to different possibilities of each sub-goal. The example of the scenario tree is shown in Figure 7.3. For example, for the sub-goal - *"User with valid account"*, the account can be either valid or invalid. Each of these possibilities correspond to a branch in the root node as shown in the top part of the figure.

3. **Identify relevant scenarios based on the requirement -** Once learners construct the scenario tree, they can then identify relevant scenarios. Using the scenario tree, learners start from the root node and traverse each possible path till a leaf is reached. Each path corresponds to a scenario in the design based on the given requirement. For example, the scenarios which can be specified based on traversing the scenario tree in Figure 7.3 are -
Scenario 1: User with a valid account has already set a PIN
Scenario 2: User with a valid account has not set a Pin and sets a valid Pin
Scenario 3: User with a valid account has not set a Pin and sets an invalid Pin
Scenario 4: User has an invalid account.

4. **Identify scenarios which are not satisfying the requirement -** Now that learners have identified possible scenarios based on the given requirement, they then try to map each scenario to the design. They simulate the execution of each scenario in the given design, and examine if any of the scenarios are absent in the design, or if certain conditions are violated. Such scenarios which do not satisfy the requirement are identified. For example, the Register sequence diagram (Figure 4.4) corresponds to the given example requirement. When learners map the scenarios to the register sequence diagram they realise that Scenario 1, Scenario 3, and Scenario 4 (the paths corresponding to the leaf nodes highlighted in yellow, in Figure 7.3) have not been specified in the sequence diagram.

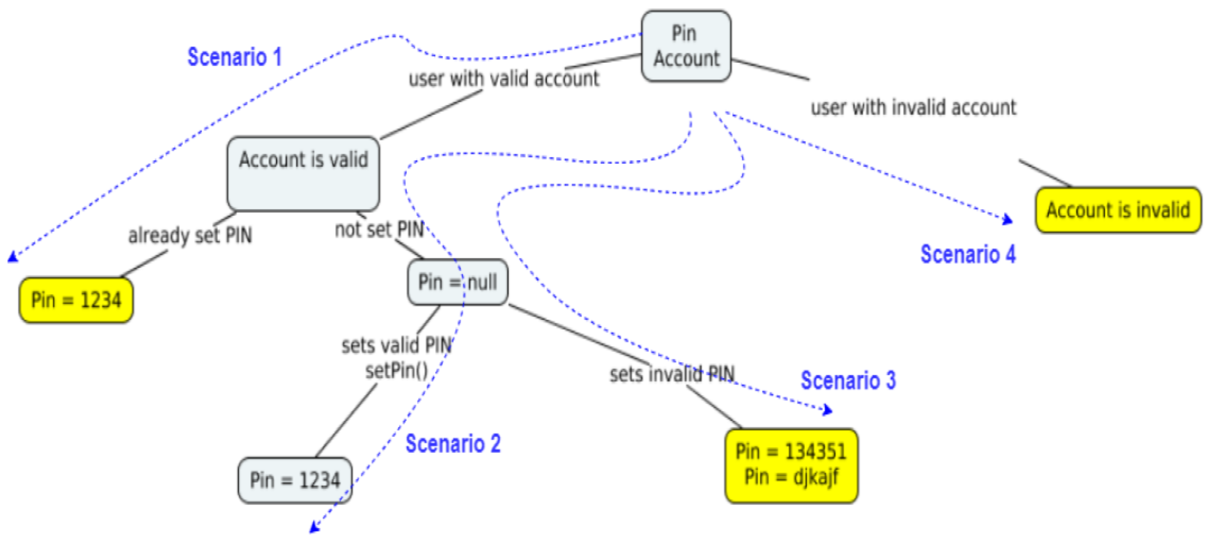


Figure 7.3: The scenario branching tree for a given requirement

To facilitate construction of the scenario branching tree, students use the Cmap [1] concept mapping tool. The Cmap tool enables users to create and share knowledge models represented as concept maps. The tool has affordances for adding nodes, and establishing links between nodes. In the context of scenario branching, each node denotes a set of values of the identified variables. The links denote different possible scenarios for the sub-goals. Using the Cmap tool, students can add, modify and delete nodes and links. Hence, the Cmap tool helps students incrementally construct the scenario tree for a given requirement. It also provides a means for learners to explore the design space by thinking of different possible scenarios in the design based on the requirements.

The scenario branching strategy and affordances of the Cmap tool also support the model-based learning practices outlined in Section 5.2.1. Learners *construct a model* of the given requirements by constructing a scenario branching tree. Students can edit the nodes and links of the tree, thereby facilitating *model revision*. Students perform *model comparison and evaluation* when the identified scenarios are compared and evaluated with the appropriate design diagrams, and missing scenarios in the design are identified. Hence, constructing the scenario branching tree using the Cmap tool assists learners in constructing an accurate mental model of the requirements, and enables them to map these requirements to the given design.

[1] https://cmap.ihmc.us/cmaptools/

Advantages of the scenario branching strategy

We hypothesize that the scenario branching strategy can help students identify relevant scenarios in the design. There are several advantages of using the scenario branching strategy. First, the visual representation of the scenario tree can help relieve the cognitive load of analysing the given requirements. The graphical layout can help students reason about each part of the requirement, and also provide them an approach to analyse the given requirements to come up with various scenarios.

Second, the scenario branching strategy can promote deeper reflection about the scenarios in the requirements and the design. Students are made to think about possible branches for each sub-goal. This helps students focus their attention on a particular part of the requirement, and think of all possible scenarios for that part. As students incrementally construct the tree for each part, they are able to identify all possible scenarios for each requirement.

Third, constructing the scenario branching tree minimizes the tendency to add new functionalities and requirements. The starting point for students to construct the scenario branching tree is the sub-goals of the given requirement. Focusing on the sub-goals enable students to focus their attention on the given requirements and not on new requirements and functionalities.

7.3 The revised VeriSIM pedagogy

The revised VeriSIM pedagogy trains students to **identify scenarios** in the design and **model these scenarios** in order to effectively evaluate a software design against the given requirements. The VeriSIM pedagogy has two key phases. Phase 1 of the revised VeriSIM pedagogy is identical to the previous DBR cycle. Phase 2 incorporates the scenario branching strategy as the learning activity. A summary of the revised VeriSIM pedagogy is shown in Figure 7.4.

1. **Phase 1: Modelling scenarios -** In this phase, learners are presented with scenarios, and construct models of these scenarios. They are scaffolded to use the design tracing strategy to trace the control flow and data flow of the scenario execution. They progressively trace these scenarios by exploring a model, correcting an incorrect model, completing an

incomplete model, and then constructing the model.

2. **Phase 2: Identifying scenarios -** Having learnt how to construct models of the given scenario, students identify various scenarios for the given requirements and design. Activities in this phase progressively scaffold them to apply the scenario branching strategy for each requirement in the design and generate scenarios. Students are provided with a step-by-step example of how to construct the scenario tree for a requirement in the given design. They are then required to apply scenario branching for the remaining requirements. For each requirement, they simulate scenario execution to identify scenarios which do not satisfy the requirements.

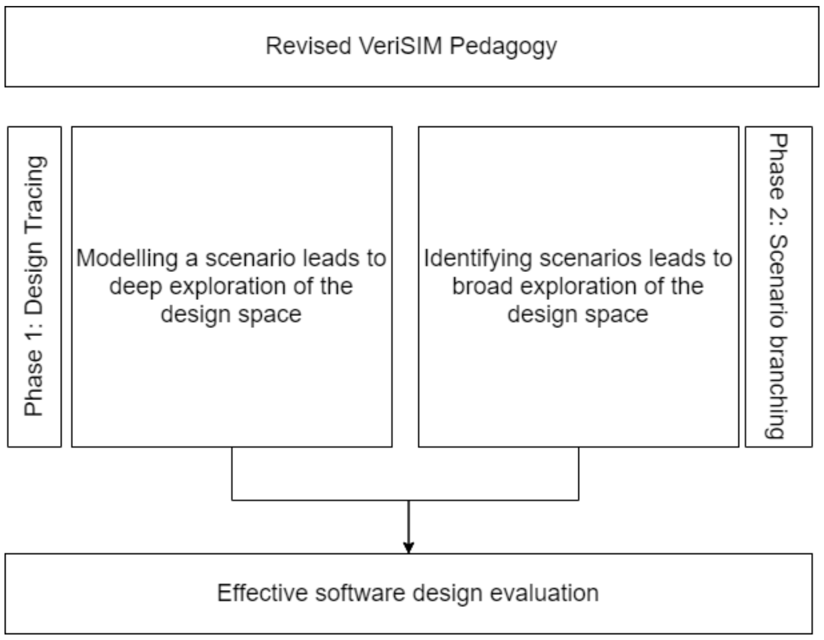

Revised VeriSIM Pedagogy
Phase 1: Design Tracing
Modelling a scenario leads to deep exploration of the design space
Identifying scenarios leads to broad exploration of the design space
Phase 2: Scenario branching
Effective software design evaluation

Figure 7.4: The revised VeriSIM pedagogy

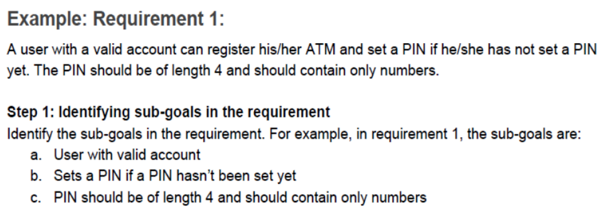
In the revised pedagogy, students are first scaffolded to model scenarios, and then provided explicit guidance on identifying scenarios. The rationale for this order is that we first want students to think deeply about the design by simulating the control and data flow for already given scenarios. Once they are trained to do this, they are then guided to identify scenarios. As they identify scenarios, they are also able to simulate the execution of these scenarios, which is similar to the design tracing strategy they learnt in Phase 1. Thus, activities in both phases help students identify and model scenarios in order to effectively evaluate a software design against the given requirements.

The revised pedagogy has been operationalised into the second version of the VeriSIM learning environment - VeriSIM 2.0, which we explain in the next section.

7.4 The VeriSIM 2.0 Learning Environment

VeriSIM 2.0 contains two modules. Module 1 and Module 2 correspond to Phase 1 and Phase 2 of the revised VeriSIM pedagogy respectively. In Module 1, learners go through the VeriSIM TELE, where they understand and apply the design tracing strategy to model the given scenarios in the design. This module is identical to the intervention provided in the previous cycle (Section 5.4). Learners interact with the VeriSIM TELE, and go through the 'Problem Understanding Stage', 'Design Tracing Stage', and the 'Reflection Stage'. These interactions enable them to effectively trace given scenarios in a software design.

Module 2 has been added outside the TELE to incorporate the scenario branching strategy. Module 2 is facilitated by an instructor. Learners are provided with the requirements and design diagrams (class diagram and three sequence diagrams) of an ATM system (described in Section 4.2.2) in a paper worksheet. The worksheet is divided into two parts. Part 1 of the worksheet describes the 4 steps outlined in Section 7.2. In the first step, the worksheet describes the relevant sub-goals for the first requirement, as shown in Figure 7.5a. In the next step, the worksheet describes how to progressively construct the scenario branching tree. The worksheet describes how to construct the intermediate scenario tree for the first sub-goal (*User with a valid account*). The intermediate scenario tree has two branches as shown in Figure 7.5b. For the next sub-goal (*Sets a PIN if a PIN hasn't been set yet*), two more branches are added to the scenario tree, as shown in Figure 7.5c. For the last sub-goal (*PIN should be of length 4 and should contain only numbers*), two more branches are added, as shown in Figure 7.5d. Students follow these steps and construct the scenario branching tree using the Cmap tool. The worksheet then explains how to identify scenarios from the scenario tree, and lists down all the scenarios. It then describes which scenarios have not been described in the design diagrams. The steps outlined in Part 1 are explained by the instructor. In Part 2, learners need to construct the scenario tree for the remaining three requirements, and identify scenarios which do not satisfy the requirements. (The scenario branching worksheet is provided in Appendix E).



Example: Requirement 1:

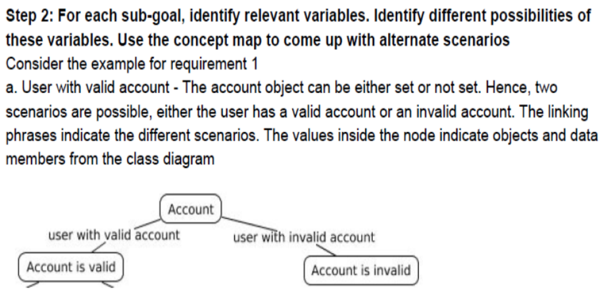
A user with a valid account can register his/her ATM and set a PIN if he/she has not set a PIN yet. The PIN should be of length 4 and should contain only numbers.

Step 1: Identifying sub-goals in the requirement

Identify the sub-goals in the requirement. For example, in requirement 1, the sub-goals are:

a. User with valid account
b. Sets a PIN if a PIN hasn't been set yet
c. PIN should be of length 4 and should contain only numbers

(a) Scenario branching worksheet: Identifying sub-goals in the requirement

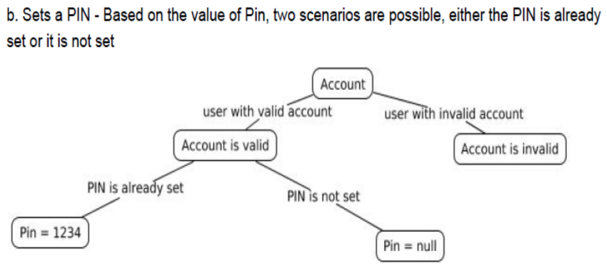


Step 2: For each sub-goal, identify relevant variables. Identify different possibilities of these variables. Use the concept map to come up with alternate scenarios

Consider the example for requirement 1

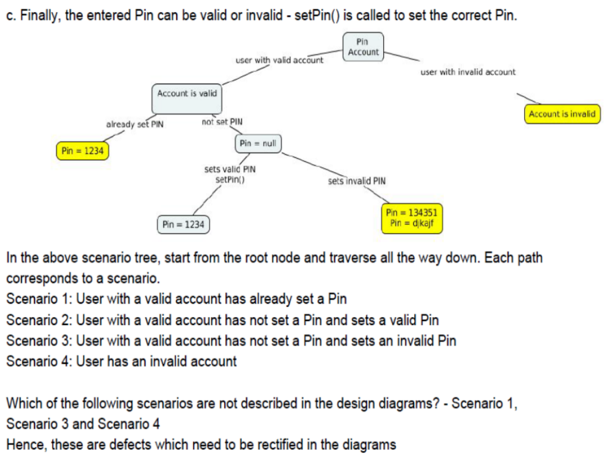
a. User with valid account - The account object can be either set or not set. Hence, two scenarios are possible, either the user has a valid account or an invalid account. The linking phrases indicate the different scenarios. The values inside the node indicate objects and data members from the class diagram

(b) Scenario branching worksheet: Identifying possibilities for the first sub-goal



b. Sets a PIN - Based on the value of Pin, two scenarios are possible, either the PIN is already set or it is not set

(c) Scenario branching worksheet: Identifying possibilities for the second sub-goal



c. Finally, the entered Pin can be valid or invalid - setPin() is called to set the correct Pin.

In the above scenario tree, start from the root node and traverse all the way down. Each path corresponds to a scenario.
Scenario 1: User with a valid account has already set a Pin
Scenario 2: User with a valid account has not set a Pin and sets a valid Pin
Scenario 3: User with a valid account has not set a Pin and sets an invalid Pin
Scenario 4: User has an invalid account

Which of the following scenarios are not described in the design diagrams? - Scenario 1, Scenario 3 and Scenario 4
Hence, these are defects which need to be rectified in the diagrams

(d) Scenario branching worksheet: Identifying scenarios from the scenario tree

Figure 7.5: Steps outlined in the scenario branching worksheet

Hence, the VeriSIM 2.0 learning environment operationalises the revised VeriSIM pedagogy, enabling students to identify and model scenarios in the given design. We believe this enables them to build an effective mental model of the requirements and the design. In the next section, we position the activities in VeriSIM 2.0 based on the mental model elements for design diagrams, and the discuss how these activities are developing specific elements in their mental models.

7.5 Role of VeriSIM 2.0 in Developing Effective Mental Models of the Design

VeriSIM 2.0 enables students to identify and model scenarios in the design, thereby fostering effective evaluation of software design diagrams. Figure 7.6 situates the strategies in the revised pedagogy in the mental model for design diagrams. The scenario branching strategy enables students to build an effective mental model of the problem domain. Using the scenario branching strategy, they are scaffolded to identify different scenarios based on the requirements. This allows them to broadly explore the mapping between the problem domain and the design solution space. The design tracing strategy enables students to closely examine each of these scenarios by analysing the elements in the design. Design tracing scaffolds students to identify the diagram structural elements and the main goals, and simulate the dynamic behaviours corresponding to each of these scenarios.

Hence, both the broad exploration of the design by identifying scenarios, and deeply thinking about each scenario by simulating the data and control flow simulation for each scenario can lead to effective evaluation of the given software design.

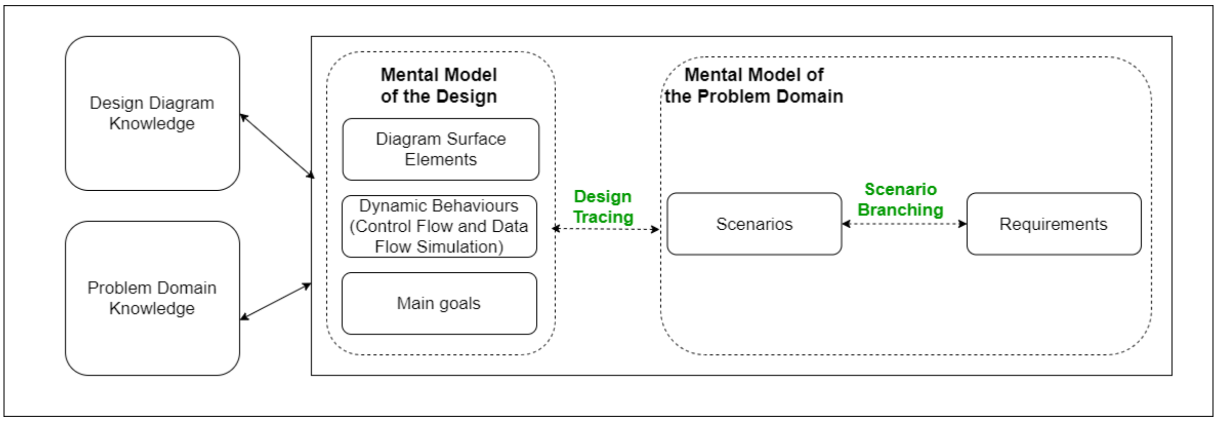


Figure 7.6: Situating the revised VeriSIM pedagogy in the mental model for design diagrams

7.6 Study 3: Effectiveness and Usefulness of the revised VeriSIM pedagogy

In Study 3, we investigate the effects of VeriSIM 2.0 on students' ability to effectively evaluate design diagrams against the given requirements.

7.6.1 Research Questions

In DBR cycle 2, we focussed on providing explicit scaffolds to help students identify scenarios in the design. Identifying alternate scenarios will enable them to identify scenarios which do not satisfy the requirements. Hence, the focus of Study 3 is to understand the effects of VeriSIM 2.0 in students' ability to identify scenarios, and whether this is leading them to identify defects in the design.

The research questions guiding this study are:

- RQ 3.1: What are the effects of VeriSIM 2.0 in students' ability to identify scenarios in the design?
- RQ 3.2: What are the effects of VeriSIM 2.0 in students' ability to uncover defects in the design?

7.6.2 Study Procedure

I conducted Study 3 in an urban private engineering institute. The study was spread over two days. 22 students (m=16, f=6) in the second year of their undergraduate degree in CS and IT, were part of the study. Prior to the study, they attended a workshop where they were taught basic UML diagrams like use case, class and sequence diagrams. Hence all students were familiar with UML diagrams prior to the study. The participants signed a consent form prior to the study, and were free to withdraw from the study at any point in time.

A summary of the study procedure is shown in Figure 7.7. I initially explained the main goals of the study, and what students will learn as they interact with VeriSIM 2.0. After providing this information, students solved a pre-test. In the pre-test, students were given the requirements and design diagrams of an ATM system, and were asked to identify defects in the design diagrams based on the requirements. After solving the pre-test, students interacted with the Module 1 of VeriSIM 2.0. I was available to answer any doubts which students had while interacting with Module 1. After a break of one hour, students were introduced to Module 2. I facilitated this session by explaining how to construct a scenario branching tree for the first requirement. Students interacted with the mapping tool and constructed scenario trees for the remaining requirements. The next day, students solved a post-test. The format of the post-test was similar to the pre-test. The requirements and design diagrams were for a video streaming website. (The design problem context was different from the library system in Study 2, as students were already exposed to the design of the library system in the workshop.) The complexity of the design problem in the pre-test and post-test was similar, as both problems contained the same number and types of requirements, sequence diagrams, and classes in the class diagram (The design problem and the questions for pre-test and post-test can be found in Appendix F). We conducted focus-group interviews with students in between Module 1 and 2, and at the end, in order to elicit their perceptions of the revised VeriSIM pedagogy.

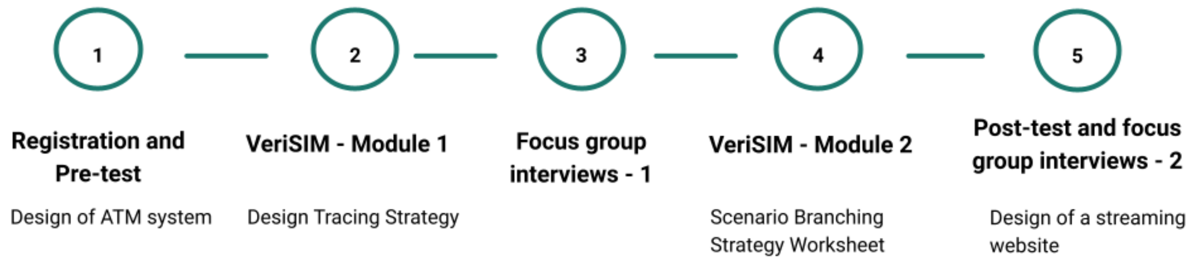

1
2
3
4
5
Registration and Pre-test
Design of ATM system
VeriSIM - Module 1
Design Tracing Strategy
Focus group interviews - 1
VeriSIM - Module 2
Scenario Branching Strategy Worksheet
Post-test and focus group interviews - 2
Design of a streaming website

Figure 7.7: Summary of Study 3 procedure

7.6.3 Data Sources and Analysis

The data sources and analysis are similar to Study 2. A summary is provided in Table 7.1. To answer RQ 3.1, we analysed student responses to the following 'identify scenarios' question in the pre-test and post-test - " *For each requirement, list all possible scenarios based on the* *design."* In order to answer RQ 3.2, we analysed student responses to the following 'identify defects' question in the pre-test and post-test - *"Identify defects (if any) in the following design diagrams based on the requirements. For each defect, provide a logical explanation of why you think it is a defect."*.

We used the content analysis method (Baxter, 1991) to answer both RQ 3.1 and RQ 3.2. Details of the analysis procedure is similar to what has been described in Study 2 (Section 6.4.2), where we followed descriptive coding, and generated categories and sub-categories to infer what types of scenarios (RQ 3.1), and types of defects (RQ 3.2) students identified in the pre-test and post-test. We also analysed the transcripts of the focus group interviews to infer themes about the effects of the scenario branching strategy in students' ability to identify scenarios and defects in the given design diagrams.

Table 7.1: Data source and analysis methods for Study 3

Research Question	Data Source	Data Analysis Method
RQ 3.1: Ability to identify scenarios	Responses to 'identify scenarios' question in the pre-test and post-test	Content analysis
RQ 3.2: Ability to identify defects	Responses to 'identify defects' question in the pre-test and post-test	Content analysis

7.6.4 Findings

RQ 3.1 - Students' ability to identify scenarios

The categories of student responses we identified are as follows:

- **Scenarios based on the requirements -** In this category, students correctly identified relevant scenarios which satisfy/do not satisfy the requirement. They also identified scenarios based on an error condition or alternate condition of the provided requirement. Example: *"There is a defect in withdraw sequence diagram where the condition whether the balance is less than 1000 is not mentioned. The check balance condition is not present here"*

- **Scenarios based on new functionality or change in requirement** - In this category of responses, students thought of scenarios not related to the requirements, but on new functionalities or change in requirements. Example: *"After payment for premium pack, after some months the user wishes to cancel its subscription"*
- **Restating the requirement -** Students merely restated the requirement, but did not provide a scenario. Example: *"The user wants to change the PIN number."*
- **Incomplete/Incorrect Scenario -** Students provided an incomplete or incorrect scenario.

Differences in the category responses of students in the pre-test and post-test is shown in Figure 7.8. In both pre-test and post-test, students identified majority of the scenarios based on the requirements. Hence, we saw no major differences between the pre-test and post-test responses.

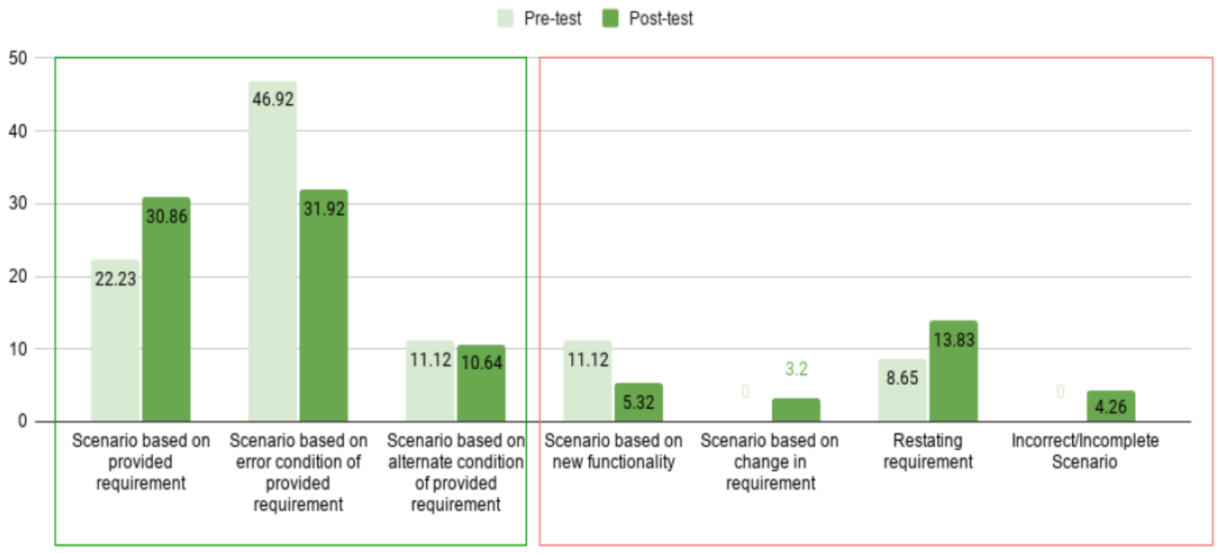


Figure 7.8: Pre-test and post-test differences in student responses to identifying scenarios

RQ 3.2 - Students' ability to identify defects

The categories of student responses we identified were similar to that of Study 1a and 2. The categories are - "Identify scenarios which do not satisfy the requirements", "Change existing functionalities and requirements", "Add new functionalities in the design", '‘Change data types, functions and structure of the class diagram", and "Blank responses and no defects". Details of each of these categories have been explained in Section 4.2.6. We categorized each student response based on the above categories. Differences in the category responses of students in the pre-test and post-test is shown in Figure 7.9.

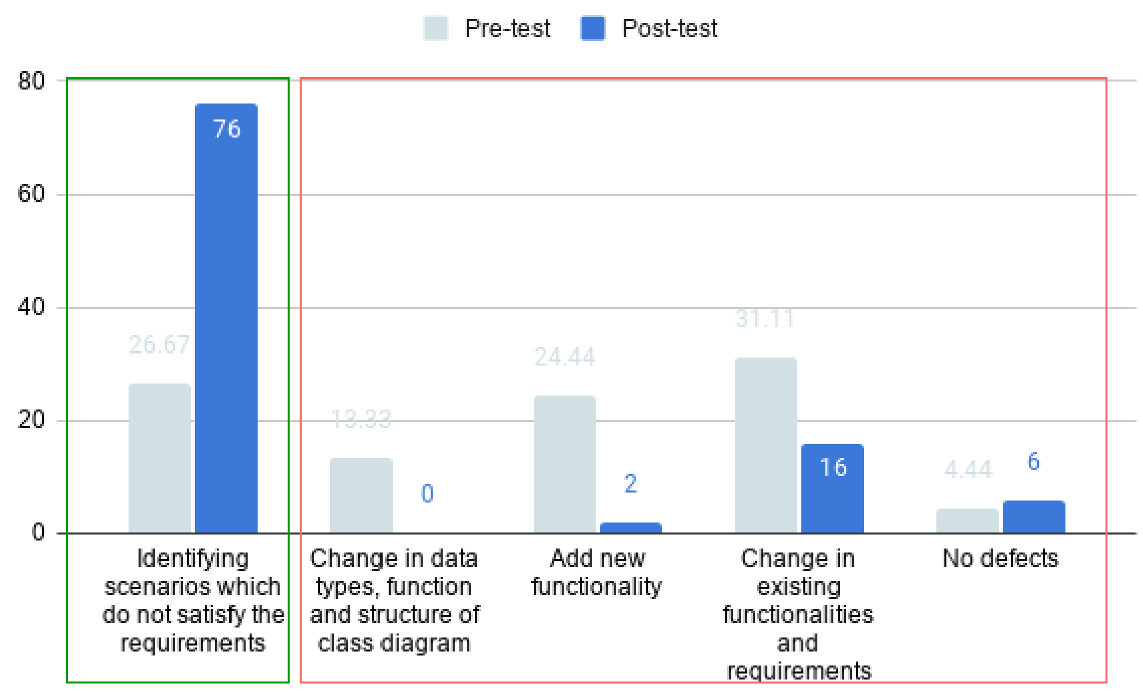


Figure 7.9: Pre-test and post-test differences in student responses to identifying defects

As seen in Figure 7.9, in the pre-test, students focused on changing the functionalities or requirements (31.1%), adding a new functionality (24.4%), and change in data types or functions (13.3%). 26.7% of the student responses involved identifying relevant scenarios which do not satisfy the requirements. In these responses, students are evaluating the design diagrams against the requirements by identifying alternate scenarios and simulating function execution.

In the post-test, we see that a majority of student responses involved identifying relevant scenarios which do not satisfy the requirement (76%). The number of student responses indicating change in functionality/requirement reduced to 16%, and only 2% of student responses involved adding a new functionality. Hence, students were able to identify more defects by identifying scenarios which do not satisfy the requirement in the post-test. Hence students' adding new functionalities and referring to them as defects have reduced in the post-test. These results indicate that students have improved in evaluating the design against the given requirements.

7.6.5 Reflections

VeriSIM 2.0 enabled students to relate the identified scenarios with defects based on the requirements

Based on the findings of RQ 3.1, we see that in both pre-test and post-test, students identified majority of the scenarios based on the requirements. However, findings of RQ 3.2 show that in the pre-test, they did not identify defects based on these scenarios. We see a shift in the post-test, as students identified more defects based on the requirements. We see that students were able to associate the identified scenarios with defects based on the requirements, which they were unable to do in the pre-test. We believe that this change was brought about by the intervention.

This finding is also corroborated by students' perceptions of the scenario branching strategy, which we elicited from the focus group interviews. Students perceived that scenario branching helped them think of possible alternate scenarios, which in turn helped them identify defects in the design diagrams. Working with the Cmap tool helped them externalize their thinking, by making them explicitly mention all possible scenarios based on the requirements. They reflected on how the scenario branching notation was missing in traditional UML diagrams, and helped them visualize different scenarios. (*E.g. "UML diagrams just give us the attributes, functions and sequence of a particular scenario. But it does not tell us about all the possible scenarios involved"*).

VeriSIM 2.0 brought about a change in students' approach towards software design

Similar to the findings of Study 2 (Section 6.6.2), students perceived that interactions with VeriSIM 2.0 brought about a change in their approach towards the software design process. In the focus-group interviews, students were asked if there were any differences in their approach in evaluating designs in the pre-test and post-test. Students mentioned that VeriSIM 2.0 enabled them to think of a **structured way to solve the problem**. (e.g. *"We didn't know which way to go to solve the problem. Now we know a structured way to follow up problems, now I could directly go for another problem."*) The emphasis on identifying and modelling scenarios enabled them to perform the design evaluation task. (e.g. *"I was able to think about more scenarios. In the ATM problem, in that problem I was not able to think what can be done new, but after learning this new session, I was able to elaborate more about what those scenarios can be."*).

Students were also asked to reflect on what they learnt and how it will be useful for them in the future. Students reflected on how interactions with VeriSIM 2.0 brought about a change in their perception about the **importance of creating a software design before jumping into writing code**. (e.g. *"For me I thought firstly before this workshop, I thought like, programming was the main thing behind all the project. But right now, I've learnt that how a blueprint is made for a project, and what software designing is. That was quite useful, it was something new."*). It broadened their perspective of thinking of the system as a whole, before jumping into specifics. For example, consider the following excerpt from an interview -

"If you ask me, 5 days before, like how will you solve this problem, I'm pretty sure, I'll jump directly onto the coding part, then I'll get stuck, and then I'll scratch my brain for like, days, and still, I'm not sure whether I'll be able to get the solution correct or not.

But now, after learning all these processes, I can design the entire problem, I can have the entire picture in front of me, and I know like, it is so close to the code, that I know how the code will like, what has to be the flow, and that's like I know and in this node, like if the variable's changing, I know that here some condition exists, so I know that I'll have to use the control flow logic, or the if-else statements here. So basically, designing, it makes it very very clear, like coding is then, just, you just have to learn the syntax, nothing else."

This excerpt shows how activities in VeriSIM 2.0 encouraged the student to focus on the overall design of the system, and specify relevant parts like the control flow and data flow in the design. The student was also able to reflect on the benefit of designing first, which would lead to a better and accurate translation to code.

To summarize, we see that VeriSIM 2.0 enabled students to identify and model scenarios in the design, thereby enabling them to effectively perform software design evaluation. This gives us certain indicators that the features and affordances of VeriSIM 2.0 are beneficial and are supporting students in their design evaluation process. In the next chapter, we delve deeper into how VeriSIM 2.0 features are contributing towards student learning, and propose an account of how learning occurs while students interact with VeriSIM 2.0.

Chapter 8

Local Learning Theory based on the VeriSIM Pedagogy

In previous chapters, we described the VeriSIM pedagogy (the initial and revised pedagogy) and how it has been operationalized into two modules of VeriSIM. Module 1 corresponds to the web-based VeriSIM learning environment, where learners are scaffolded to apply the design tracing strategy. Module 2 corresponds to the worksheet and mapping tool which supports students in applying the scenario branching strategy.

Studies with students interacting with VeriSIM (Study 2 and 3), give indicators for improvement in students' ability to trace as well as identify scenarios which do not satisfy the given requirements. These improvements would have likely been due to the role of various features in VeriSIM. The mediating role of these features and how they contributed to learning is an important aspect to consider, especially in the design-based research paradigm. An important aim of the design based research methodology is to identify underlying design principles which can be extended and used in other similar contexts and settings (Barab and Squire, 2004). These principles can contribute to a "local learning theory" i.e. an account of how learning is happening in the local context of the given intervention. In this chapter, we delve deeper into how learners are interacting with VeriSIM, and in what ways it is leading to effective student learning. Findings from this chapter are used to answer RQ 4 in DBR Cycle 2 (see Section 3.4):

RQ 4: How are features in the VeriSIM learning environment contributing towards student learning?

8.1 Data Sources and Analysis Method

In order to develop the local learning theory based on students' interactions with VeriSIM, we further analysed the data collected from Study 2 and Study 3. We primarily draw inferences from the analysis of two data sources - (1) Interaction log data captured by the VeriSIM learning environment, and (2) Focus-group interviews with students after their interaction with VeriSIM. From the analysis of these data sources, we inferred how features in VeriSIM contributed to student learning. We describe these inferences in detail in subsequent sections.

Analysis of Interaction Logs

Since students interacted with VeriSIM Module 1 in both Study 2 and Study 3, we analysed the interaction logs from both studies in order to build upon the local learning theory. In both Study 2 and Study 3, I specifically asked students for consent to use their interaction logs in the consent form. A total of 48 students gave consent to use their interaction logs.

As students interacted with VeriSIM Module 1, specific user interactions were logged, such as which challenge the user was at a particular time, and how they interacted with the state diagram model. Interactions with the state diagram included action sequences while constructing, editing, and executing the state diagram model in different challenges, and what feedback was provided by the agent as a result of these actions. The logs also captured their answers to the evaluation and reflection questions. The process which we followed to extract these interaction logs is explained below.

After conducting Study 2 and Study 3, I downloaded all user interaction logs from the server. These interactions of all users were stored in a single CSV file on the server, ordered by timestamp. I used the pandas library [1] in Python for the pre-processing and analysis. From the single CSV file, I extracted all interaction logs for each user, ordered by timestamp. Hence, I now had a separate file which contained the interaction logs of each user. I added additional columns like 'display_name', and 'duration' for each user file. I then combined all user files and generated a new CSV file. A snapshot of this CSV file is shown in Table 8.1. The fields of this CSV file are:

- **id**- Each row in the file has a unique id which distinguishes it from other entries in the file.
- **timestamp**- The timestamp corresponds to the time in which a particular event was logged by the system.
- **duration**- The duration of an event is the difference between the timestamps of this event and the previous event. This field was added mainly to aid readability and analysis of the logs (I found it easier to compare integers rather than timestamps)
- **screen**- This field denotes the particular screen that the user was in when the event was logged. For example, Table 8.1 is a snapshot which captures the interactions of a user while they attempted the following activities: Challenge 1, Evaluation questions of Challenge 1 and Challenge 2, in this order.
- **user_id** - Each user of the system is assigned a unique id.
- **display_name** - This field denotes the first name and last name of the registered user.
- **action_name** - This field denotes the type of action logged by the system, based on the activity the user is performing in the system. The following action types are possible -
  - **run** - When the user clicks on the 'run' button in the challenges, the action_name is set to 'run'
  - **system_action** - A row has the 'system_action' action_name associated with it when the pedagogical agent provides feedback to the user on clicking the 'run' button in the challenges.

[1]https://pandas.pydata.org/

– **data** - When a user adds/edits/deletes a data variable in the challenges, the action_name is set to ‘data’

– **event** - When a user adds/edits/deletes an event in the challenges, the action_name is set to ‘event’

– **state** - When a user adds/edits/deletes a state in the challenges, the action_name is set to ‘state’

– **question** - When a user attempts a reflection or evaluation activity, the action_name is set to ‘question’

Table 8.1: A snapshot of VeriSIM pre-processed interaction logs

	id	timestamp	duration	screen	user_id	display_name	action_name	data_score	event_score	state_score
1	5d884dea18919072731c05d5	2019-09-23 04:39:27.385000+00:00	22	Challenge 1	5d88493618919072731c014f	XXX	system_action	0	0	0
2	5d884e4e18919072731c06dd	2019-09-23 04:41:05.264000+00:00	97	Challenge 1	5d88493618919072731c014f	XXX	run	4	3	19
3	5d884e4e18919072731c06de	2019-09-23 04:41:05.266000+00:00	0	Challenge 1	5d88493618919072731c014f	XXX	system_action	4	3	19
4	5d884e5d18919072731c0716	2019-09-23 04:41:18.361000+00:00	13	Challenge 1	5d88493618919072731c014f	XXX	run	4	3	19
5	5d884e5d18919072731c0717	2019-09-23 04:41:18.363000+00:00	0	Challenge 1	5d88493618919072731c014f	XXX	system_action	4	3	19
6	5d884e5d18919072731c0718	2019-09-23 04:41:20.335000+00:00	1	Challenge 1	5d88493618919072731c014f	XXX	run	4	3	19
7	5d884e5d18919072731c0719	2019-09-23 04:41:20.337000+00:00	0	Challenge 1	5d88493618919072731c014f	XXX	system_action	4	3	19
8	5d885074cabef674ecdbe8a5	2019-09-23 04:50:16.570000+00:00	0	Challenge 1	5d88493618919072731c014f	XXX	run	4	3	19
9	5d885074cabef674ecdbe8a6	2019-09-23 04:50:16.571000+00:00	0	Challenge 1	5d88493618919072731c014f	XXX	system_action	4	3	19
10	5d885083cabef674ecdbe8fb	2019-09-23 04:50:30.352000+00:00	13	Challenge 1	5d88493618919072731c014f	XXX	system_action	4	3	19
11	5d885132cabef674ecdbec51	2019-09-23 04:53:27.196000+00:00	176	Q - Challenge 1	5d88493618919072731c014f	XXX	question	0	0	0
12	5d885132cabef674ecdbec52	2019-09-23 04:53:27.196000+00:00	0	Q - Challenge 1	5d88493618919072731c014f	XXX	question	0	0	0
13	5d885132cabef674ecdbec53	2019-09-23 04:53:27.196000+00:00	0	Q - Challenge 1	5d88493618919072731c014f	XXX	question	0	0	0
14	5d885132cabef674ecdbec54	2019-09-23 04:53:27.197000+00:00	0	Q - Challenge 1	5d88493618919072731c014f	XXX	question	0	0	0
15	5d885132cabef674ecdbec55	2019-09-23 04:53:27.197000+00:00	0	Q - Challenge 1	5d88493618919072731c014f	XXX	question	0	0	0
16	5d88516ecabef674ecdbed2f	2019-09-23 04:54:24.089000+00:00	56	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	14
17	5d88516ecabef674ecdbed30	2019-09-23 04:54:24.092000+00:00	0	Challenge 2	5d88493618919072731c014f	XXX	system_action	4	3	14
18	5d885173cabef674ecdbed4d	2019-09-23 04:54:31.040000+00:00	6	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	14
19	5d885173cabef674ecdbed4e	2019-09-23 04:54:31.043000+00:00	0	Challenge 2	5d88493618919072731c014f	XXX	system_action	4	3	14
20	5d885173cabef674ecdbed4f	2019-09-23 04:54:32.248000+00:00	1	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	14
21	5d885173cabef674ecdbed50	2019-09-23 04:54:32.249000+00:00	0	Challenge 2	5d88493618919072731c014f	XXX	system_action	4	3	14
22	5d885173cabef674ecdbed51	2019-09-23 04:54:33.056000+00:00	0	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	14
23	5d885173cabef674ecdbed52	2019-09-23 04:54:33.057000+00:00	0	Challenge 2	5d88493618919072731c014f	XXX	system_action	4	3	14
24	5d8851b4cabef674ecdbeee0	2019-09-23 04:55:34.713000+00:00	61	Challenge 2	5d88493618919072731c014f	XXX	state	4	3	15
25	5d8851b9cabef674ecdbeef8	2019-09-23 04:55:40.759000+00:00	6	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	15
26	5d8851b9cabef674ecdbeef9	2019-09-23 04:55:43.155000+00:00	2	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	15
27	5d8851b9cabef674ecdbeefa	2019-09-23 04:55:43.157000+00:00	0	Challenge 2	5d88493618919072731c014f	XXX	system_action	4	3	15
28	5d885223cabef674ecdbf087	2019-09-23 04:57:27.068000+00:00	103	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	15
29	5d885223cabef674ecdbf088	2019-09-23 04:57:27.876000+00:00	0	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	15
30	5d885223cabef674ecdbf089	2019-09-23 04:57:27.876000+00:00	0	Challenge 2	5d88493618919072731c014f	XXX	system_action	4	3	15
31	5d885254cabef674ecdbf13e	2019-09-23 04:58:14.444000+00:00	46	Challenge 2	5d88493618919072731c014f	XXX	state	4	3	14
32	5d885254cabef674ecdbf13f	2019-09-23 04:58:17.620000+00:00	3	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	14
33	5d885259cabef674ecdbf150	2019-09-23 04:58:18.883000+00:00	1	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	14
34	5d885259cabef674ecdbf151	2019-09-23 04:58:18.884000+00:00	0	Challenge 2	5d88493618919072731c014f	XXX	system_action	4	3	14
35	5d885259cabef674ecdbf152	2019-09-23 04:58:19.123000+00:00	0	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	14
36	5d88529acabef674ecdbf216	2019-09-23 04:59:26.171000+00:00	67	Challenge 2	5d88493618919072731c014f	XXX	state	4	3	14
37	5d88529fcabef674ecdbf21c	2019-09-23 04:59:30.173000+00:00	4	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	14
38	5d88529fcabef674ecdbf21d	2019-09-23 04:59:30.175000+00:00	0	Challenge 2	5d88493618919072731c014f	XXX	system_action	4	3	14
39	5d8852b8cabef674ecdbf24f	2019-09-23 04:59:57.674000+00:00	27	Challenge 2	5d88493618919072731c014f	XXX	state	4	3	16
40	5d8852bdcabef674ecdbf25d	2019-09-23 05:00:00.325000+00:00	2	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	16
41	5d8852bdcabef674ecdbf25e	2019-09-23 05:00:01.699000+00:00	1	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	16
42	5d8852bdcabef674ecdbf25f	2019-09-23 05:00:01.956000+00:00	0	Challenge 2	5d88493618919072731c014f	XXX	run	4	3	16
43	5d8852bdcabef674ecdbf260	2019-09-23 05:00:01.957000+00:00	0	Challenge 2	5d88493618919072731c014f	XXX	system_action	4	3	16
44	5d885331cabef674ecdbf3a8	2019-09-23 05:01:56.800000+00:00	114	Challenge 2	5d88493618919072731c014f	XXX	state	4	3	16

In VeriSIM, a score is associated for each constructed state diagram. This score is stored in the logs and is not visible to learners. The purpose of this score for the state diagram is to progressively see how learners are building the state diagram model. For each design tracing challenge, there are three types of scores which are logged (as shown in Table 8.1) - data_score, event_score and state_score.

- **data_score** - When the learner adds a correct variable, the data_score is incremented by 1. When the learner adds an irrelevant variable, the score is decremented by 1.
- **event_score** - When the learner adds a correct event, event_score is incremented by 1. When the learner adds an incorrect event, the score is decremented by 1.
- **state_score** - Each state (except the initial state) has an incoming event. The state_score is incremented by 1 if the incoming event is correctly chosen. Each state contains a list of variables and its corresponding values. For each correct variable, value pair, the score is incremented by 1.

Based on each add, edit or delete action on the state diagram, the score gets updated in the corresponding field. Hence, the progression of the score for a particular challenge is an indicator of how a learner is constructing the state diagram. For example, based on the interaction log of XXX shown in Table 8.1, XXX starts Challenge 2 by clicking on 'run' multiple times and observes the feedback from the agent (Rows 16-23). After this, he/she edits a state, as shown in row 24. This results in the state score changing from 14 to 15, which shows that the learner modified a state and corrected the value of an incorrect variable. XXX then clicks on 'run' again and observes the feedback (Rows 25-30). However, we see in row 31 that he incorrectly changes the value of a variable in a state, resulting in the state score changing from 15 to 14. In row 39, we see that the state score increases to 16, which indicates that the learner corrected the variables based on the feedback. In this manner, the internal data, event and state scores coupled with the 'action_names', 'duration', and 'screen' fields, give indicators of the sequence of actions a user performed in VeriSIM Module 1.

Analysis of Student Interviews

In Study 2 and 3, I had conducted focus-group interviews with students, where I asked questions related to the usefulness of the design tracing and scenario branching strategy. In addition to these questions, I had also asked questions regarding how students went about interacting with VeriSIM. I followed the same thematic analysis procedure as Study 2 and 3, to analyse their responses to these questions (Details can be found in Section 6.4.3). The focus of these questions was to delve deeper into how students were interacting with VeriSIM, and to gather their perceptions about the features in VeriSIM.

In the next sections, we examine how each of the features in VeriSIM contributed to student learning.

8.2 VeriSIM helps students progressively trace scenarios in the design

The 'Design Tracing Stage' activities in VeriSIM help students progressively trace scenarios in the design. In Chapter 5, we described the challenges in the 'Design Tracing Stage' of VeriSIM. The challenges were ordered in increasing order of complexity, where students first explored the model, then corrected an incorrect model, completed an incomplete model, and finally constructed the model from scratch. We derived this order of modelling activities from literature (Mulder et al., 2011), and it has shown to be beneficial to learners. Although this is the recommended ordering, it was not imposed strictly, and students were free to attempt the activities in any order.

What are certain indicators that students found the progression beneficial? We examined the following attributes from the interaction log data and interviews to answer this question.

- **The order students followed -** Using the interaction logs, we determined the order in which students interacted with different activities. We looked at what are possible paths learners took as they interacted with the VeriSIM learning environment.

- **Time spent in each challenge -** The interaction logs captured the timestamp of the start and end of each activity. Thus, the time spent in each challenge can serve as a proxy for the difficulty for that challenge.

- **Students perceptions -** During the focus group interviews, we asked students their perception of the order of activities. These responses complemented the results from the log data, and helped us gauge the effectiveness of the order of activities.

8.2.1 Findings from interaction logs

From the interaction logs, we observed that all 48 students followed the prescribed order of challenges. We also looked at the average time spent by students in each task. We chose the metric of average time spent, as the time spent on a task can be considered as a reasonable estimate of the difficulty of that task. Table 8.2 and Figure 8.1 shows the average time spent in each challenge by students. We observe that students spent the least time on Challenge 1. This is obvious, as learners had to only explore the state diagram and observe its execution in Challenge 1.

However, we observed that students spent the most time in Challenge 3. Based on the model progression of activities, Challenge 4 was the most difficult one among the challenges, since students had to construct the entire state diagram from scratch. Hence, it is reasonable to assume that students will spend the most time in Challenge 4. However, we observed that students spent more time in Challenge 3 rather than Challenge 4. Students spending lesser time in Challenge 4 than Challenge 3, gives an indication that the model progression helped students construct the model of a scenario. By the time students reached Challenge 4, they had already observed a correct trace, corrected an incorrect trace, and completed an incomplete scenario trace. Hence, they required lesser time to construct the model of a scenario from scratch in Challenge 4.

Table 8.2: Average time spent by students in the design tracing activity in VeriSIM

Challenge 1	Q - Challenge 1	Challenge 2	Q - Challenge 2	Challenge 3	Q - Challenge 3	Challenge 4	Q - Reflection
3.36	2.40	8.43	5.03	11.32	5.63	8.80	5.62

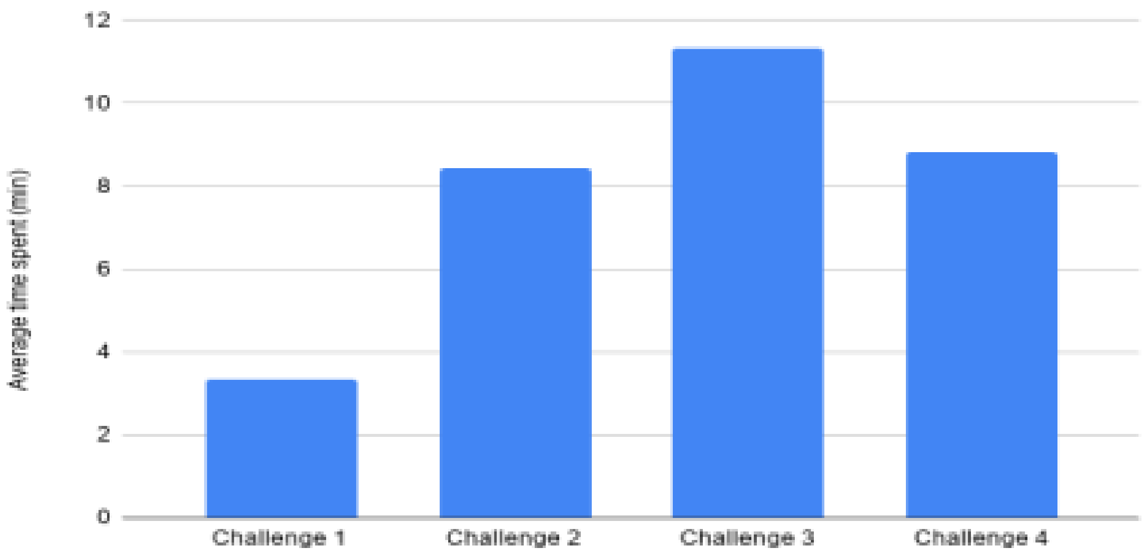


Figure 8.1: Average time spent by 48 students in each challenge

8.2.2 Findings from focus group interview

Most of the students in the focus group interview also mentioned that they followed the challenges in order. The key perception by students was that they found the **challenges in increasing order of difficulty**. Each challenge **added some more complexity** than the previous challenge (*"Firstly, the whole diagram was given, then correct the mistake... We were going step by step"*) This **enhanced their understanding** of the design diagrams (*"We were going step by step - understood how the system really work"*) as well **as enabled them to construct the state diagram at the end**. (*Like we solved it sequentially. So .. And when nothing was given, it was the last question. So I basically had the idea how to go about solving the last challenge*). These perceptions give us indicators that the model progression of the challenges enabled students to model the scenarios in the design. A more open-ended approach (introducing Challenge 3 or Challenge 4 at the start) may not have achieved the desired effect (e.g.:*"1. Rather than just pushing us to create state diagrams, it helped us move step by step. 2. It (solving Challenge 4) became easier for us by exploring the model in the first state"*)

8.3 VeriSIM helps students visualize execution of scenarios in the design

The model editing and execution features in VeriSIM help students visualize execution of scenarios in the design. The VeriSIM learning environment provides students opportunities to visualize the execution of the state diagram by clicking on a “Run” button. In Challenge 1, learners explore an already constructed state diagram. During the run of the state diagram, learners observe corresponding changes in the class diagram and sequence diagram. In Challenge 2, students are required to correct an incorrect state diagram. When learners click on the “Run” button for the given state diagram, the states which are incorrect are highlighted in red. They are supposed to edit the states by changing the values of the variables in the state. In Challenge 3, the states are empty, and relevant data variables and their values have to be added. Finally, in Challenge 4, learners have to construct the state diagram from scratch. Learners can add, edit, and delete data, events, and states by clicking on the “Edit” button. We examined the interaction logs and interviews to explore the usefulness of the model execution visualization.

- **Patterns of using ‘run’ in each challenge -** Actions such as adding/editing/deleting data, events and states are logged by the system in the interaction logs. We analysed patterns of such actions in each user. We excluded Challenge 1, since students did not modify the state diagram in this challenge. For each challenge, a user’s actions were plotted onto a graph, with the x-axis corresponding to the time, and the y-axis corresponding to the actions in that challenge (examples in Figure 8.2 and 8.3). From the 48 students who gave consent for use of their interaction logs, 39 students completed all 4 challenges in the design tracing stage. Hence we manually examined the 117 graphs (39 x 3) in order to identify different strategies which students used.

- **Student perceptions -** From the focus group interviews, we examined what were students’ perceptions about the model execution visualization and how it helped them in the challenges.

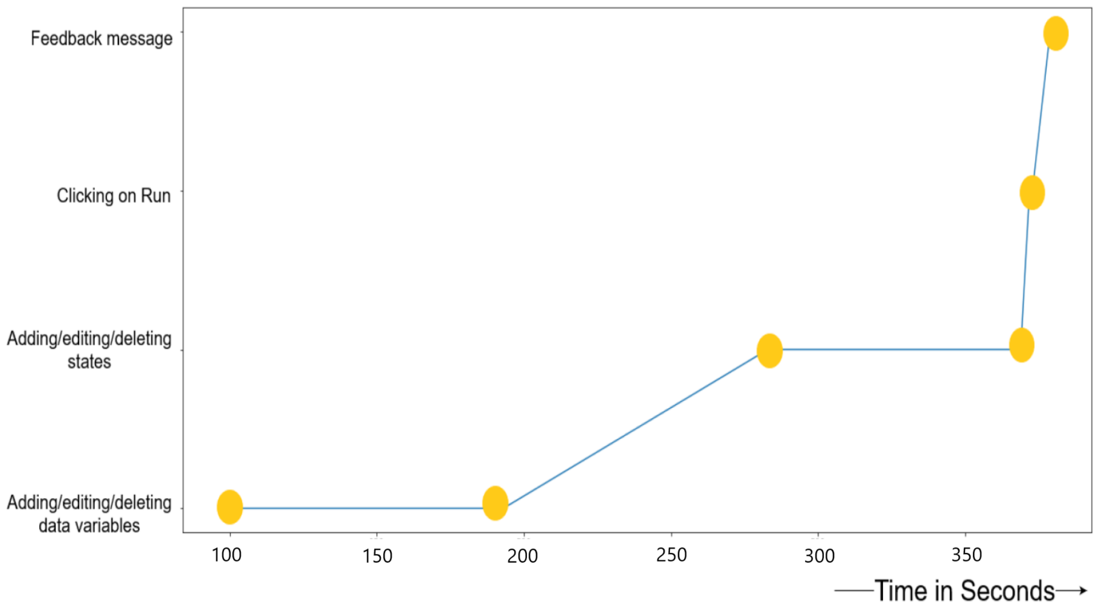


Figure 8.2: Example graphs of the Single Run strategy

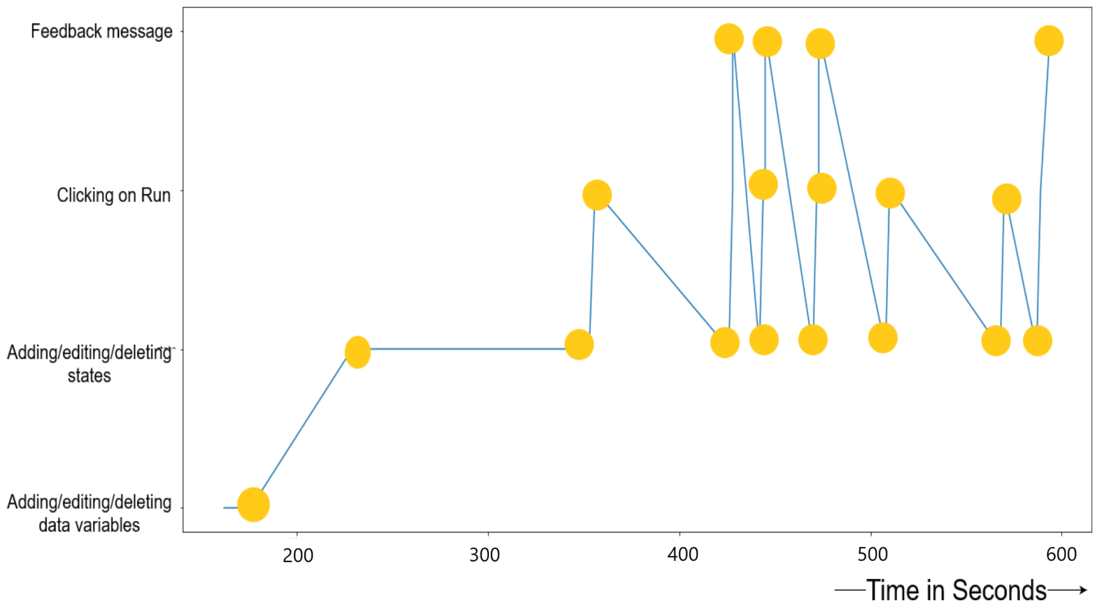


Figure 8.3: Example graph of the Multiple Runs strategy

8.3.1 Patterns of using the run visualization in challenges

Manual examination of the graphs led to identification of three distinct strategies across the three challenges.

Single Run Strategy: All modifications followed by single run at the end: In this strategy, students worked on the state diagram by modifying (adding/editing/deleting) data variables, and modifying (adding/editing/deleting) states, and clicked on 'run' only at the end to verify their correctly constructed state diagram (an incorrect state diagram would have led to further edits and runs). An example graph of this strategy is shown in Figure 8.2. We saw 37 instances (out of 117) across the three challenges of students using the Single Run strategy.

Few Runs Strategy: Between one to three cycles of modification and run: In this strategy, students modified data variables and states, clicked on 'run', and modified the data/states again. Students who followed upto three cycles of this behaviour were classified to this strategy. We saw 28 instances (out of 117) across the three challenges of students using this strategy.

Multiple Runs Strategy: Multiple cycles of modification and run: In this strategy, students performed multiple cycles of modifying data variables and states, and clicking on run, as shown in Figure 8.3. We saw 52 instances (out of 117) across the three challenges of students using this strategy.

We analysed the predominant strategies used in each challenge, and how transitions of strategies are happening in different challenges. The transitions corresponding to the interaction logs for the 39 students is shown in Figure 8.4. From the figure, we see that 54% of students (21 out of 39) attempted Challenge 2 by employing multiple cycles of modification and run (Multiple runs strategy). But by the time they reached Challenge 4, 56% (22 out of 39) shifted to using the 'run' only at the end (Single run strategy), and only 21% (8 out of 39) still used the multiple runs strategy.

What can this shift from 'Multiple Runs' to 'Single Run' indicate? Students execute the state diagram to get feedback on their model. Hence, at the start (Challenge 2), they used the 'run' scaffold to frequently validate their model. Based on this feedback, they refined their model. By the time they reached Challenge 4, the previous challenges had helped them visualize the execution of the control flow and data flow of a given scenario. Thus, in Challenge 4, they did not explicitly use the ‘run’ scaffold, but could internalise the control flow and data flow for the given scenario. They constructed their model, and verified it at the end.

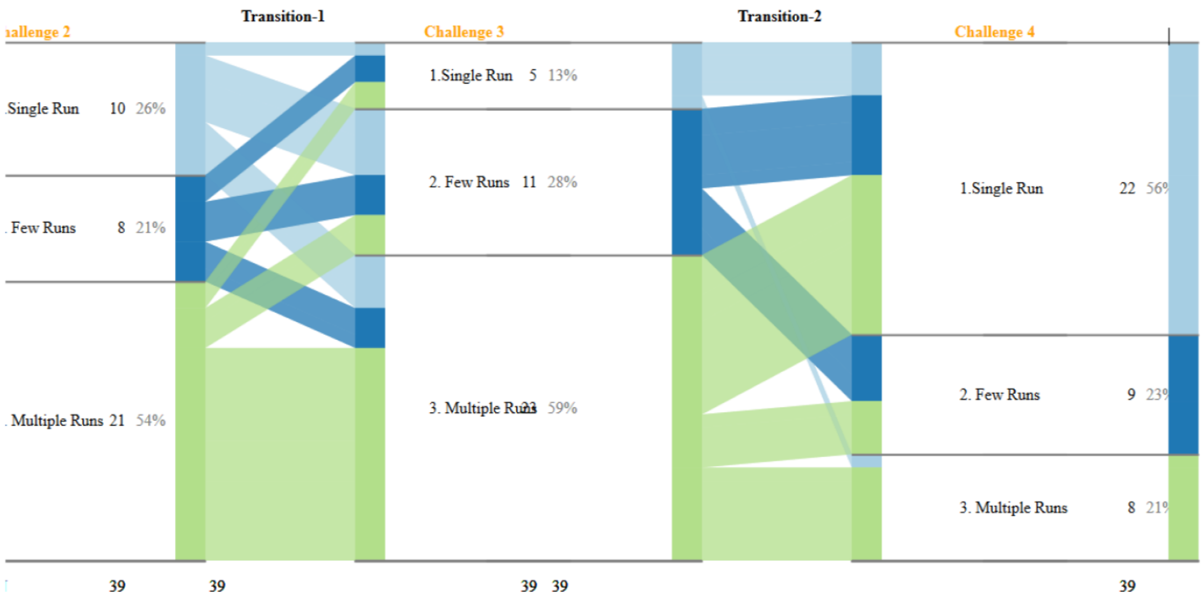


Figure 8.4: Transition diagram of the strategies used by students in each challenge

We also examined how students are transitioning to Challenge 4 from other challenges. Out of the 8 students who used the multiple run strategy in Challenge 4, 7 of them used the same strategy in Challenge 2 and 3 as well. This shows that students who used the multiple run strategy stick to it across the challenges. We can infer that these students require the ‘run’ scaffold to repeatedly validate their model.

We also observed that out of the 22 students who used the single run strategy in Challenge 4, 12 of them used ’multiple runs strategy’, 6 of them used ’few runs’, and 4 of them used ‘single run’ in Challenge 3. Hence, most number of transitions have happened from multiple runs to single runs between Challenge 3 and 4. This finding also supports our earlier inference that students were able to simulate the control and data flow in their mind, and did not require the ‘run’ scaffold by the time they reached Challenge 4.

8.3.2 Findings from focus group interview

Students perceived that the model execution feature in VeriSIM was useful. First, **the state diagram execution helped students map a particular state to the corresponding part of**

the scenario (*" if we pressed the run button, it showed what part of the scenario covers which state... Basically, it linked the scenario with the state."*). Students also found that the highlighting of the scenario as well as the sequence diagram was useful in **understanding the relationship between the scenario and different diagrams** (*E.g. "It was mapping the step-by-step scenario with the sequence diagram"*).

Visual feedback helped learners identify which parts had errors. (*" If there was something wrong, then it would be in a different colour"*). Students compared the "run" feature to debugging in programming, and this helped them build the model and rectify it in case of errors. **Positive feedback also served as an indicator that a sub-part of the model was correctly modelled,** and students could move on to the next part (*".. it (clicking on run) indicates that the first step is now completed. We can move onto the second step"*). Students found the step-wise run more beneficial as compared to a 'run' at the end, because it helped them rectify errors in their model. (*E.g. "Because at the first step, we need to know where we are wrong. After doing whole execution and getting to know and going back on that same point is more difficult and tedious and correcting it at that point."*). Thus, the step-wise execution and evaluation of the model helped students break down the task of modelling the scenario into smaller tasks, thereby making the task easier.

When students were asked how they executed the model, we found instances of both the 'Single run strategy' and 'Multiple run strategy' in their responses. (Example of Single run strategy - *"After I completed like whatever I thought was correct, after that I tested.. if there was a mistake, then it showed me in red color, that something was wrong in this part. I once again looked at it, and then if I found a mistake, I corrected it."* Example of Multiple run strategy - *"I directly read the question, and I edited the first state, and I pressed the run button. There was some mistakes, I rectified it, and again I pressed run.."*) These responses also serve as a triangulation of the patterns identified in the interaction logs.

8.4 VeriSIM helps students identify scenarios in the design

The scenario branching strategy helps in identifying scenarios in the design. In VeriSIM Module 2, students interacted with the mapping tool to construct a scenario branching tree for each requirement. We analysed their perceptions in order to understand the role of the mapping tool in identifying scenarios.

Students found the scenario branching strategy useful in identifying scenarios. They commented on how it was useful in **structuring the design problem** (*"We didn't know which way to go to solve the problem. Now we know a structured way to follow up problems"*). The strategy helped in **breaking down the problem** into scenarios (*"problem can be broken down into scenarios and can be represented in a single diagram"*). The visual representation of the scenario tree enabled them to view all possible scenarios for each requirement (*" CMAP I think, like in a very broad manner, is a connection of state diagrams. Like it entirely shows the whole picture"*), and hence helped them get a **macro-view of the design problem** (*"It gives a bigger picture about the task that we are trying to solve."*). The simplicity of the representation aided them to **identify scenarios which were missing in the design diagrams** (*"In CMAP, we can include all the scenarios. But in sequence diagrams and all, we sometimes, fail to do that, sometimes some scenarios get left out, because the diagram is very complex. But in CMAP, it becomes easy to find out the scenario is remaining, let's add it there."*)

Hence, the explicit focus on scenarios helped students **evaluate the given design**, which they found difficult during the pre-test (*" I was able to think about more scenarios. In the ATM problem, in that problem I was not able to think what can be done new, but after learning this new session, I was able to elaborate more about what those scenarios can be."*). It also helped them **identify missing functions in the sequence diagrams** (*"If one function is missing, by noting down the scenarios we can add them... I identified one function like, when the user types the username and password, then there was no function to check whether the username is already there or not."*)

8.5 VeriSIM facilitates strengthening of concepts learnt

The reflection and evaluation activities in VeriSIM facilitate strengthening of concepts learnt in each challenge. After each challenge in the design tracing stage, students were asked to reflect on what they learnt in the previous challenge. The objective of the reflection and evaluation activities is to make students aware of what they learnt in an activity. We examined the following attributes from the interaction log data and interviews to explore the role of these reflection and evaluation activities.

- **Time spent in activities -** Similar to the challenges, the interaction logs also captured the timestamp of the start and end of the reflection and evaluation activities.
- **Responses to the reflection and evaluation questions -** The reflection and evaluation questions were mandatory and students could not progress further until they submitted a response. We examined what responses students provide for these questions. A thematic analysis of the responses helped us determine whether students responded to these questions in a meaningful way.
- **Student perceptions -** During the focus group interviews, we also asked students about their perception of the reflection and evaluation activities.

8.5.1 Findings from interaction logs

From Table 8.2 and Figure 8.5, we see that students spent on average of around 5 minutes in the reflection and evaluation questions for Challenge 2, Challenge 3 and the final reflection stage. (Students spent around 2.5 minutes in Challenge 1, which had only evaluation questions). We analysed and grouped the responses to prominent themes, which is shown in Table 8.3. We see that for the reflection questions, students gave responses summarizing the activity, describing what they learnt and what was difficult. We also observed that certain students left the answer blank, or provided non-meaningful responses. However, these type of responses were not many.

We also analysed students' performance in the evaluation activities of the Problem Understanding Stage. We specifically focussed on their scores in this stage, as these evaluation activities were the primary means to determine whether students understood the requirements, design diagrams and scenarios. We found that average scores of students in the evaluation activities of 'Understand the Client and their Requirements' was 3.79 out of 4, 'Understand the Software Design Diagrams' was 3.6 out of 4, and in 'Understand Scenarios' was 1.79 out of 2. These scores indicate that the evaluation activities enabled them to build an adequate understanding of the requirements and the design diagrams.

Hence, the time students spent in the reflection and evaluation questions, as well as their responses give indicators that these questions were useful for students in their learning.

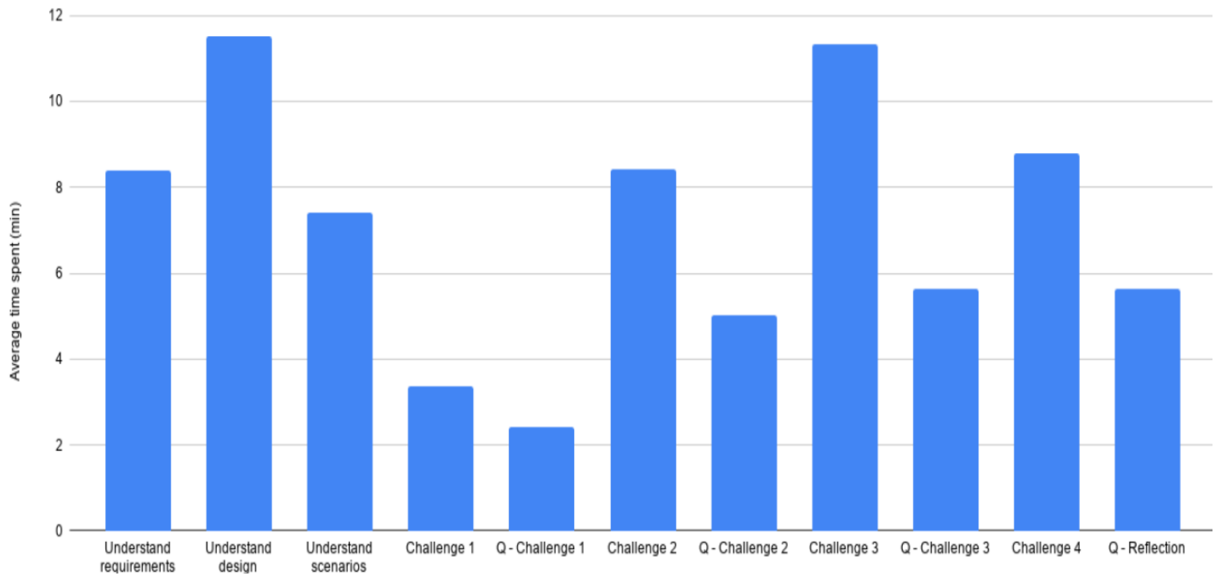


Figure 8.5: Average time spent by students in all activities

Table 8.3: Themes emerging from student reflection responses in VeriSIM

Main themes emerging from reflection activity responses	Examples
Summary of activity	"The main goal was to set the correct input so that it follows a correct sequence and unlock the door."
	"the main goal was to create a state diagram in accordance with the variables , classes, an the scenario given."
What I learnt	"I learnt how a particular scenario is traced from one state to another."
	"I learnt the use of state diagrams which is definitely a crucial part of software development"
What was difficult	"The challenging fact was to complete the state diagram with only the data variables."
	"The challenging part of the previous challenge was to correct all the errors simultaneously in each part."
Blank/Non-meaningful statements	"very nice challenge"

8.5.2 Findings from focus group interview

The key themes emerging from the focus group interviews are that the reflection and evaluation activities:

- **Enabled students to revise, reflect and summarize what they did -** Students said that having the reflection immediately after the challenge helped them "recollect what we they did" and "reflect back on what they learnt".
- **Enabled students to connect concepts -** While solving the challenges, students may miss

certain key concepts which were required to be learnt. The evaluation questions tested the knowledge gained in the previous challenge. The feedback from these evaluation questions helped them identify gaps in their knowledge. (*E.g. " it helped us to identify, map where all the shortcomings were and what was our knowledge, what was important, what was not important."*). Answering the reflection questions also triggered the realisation of a connection between concepts. (*E.g. "When I was writing, I realised there is one point, there is the other point, and this is the connection between them"*)

- **Reduced cognitive load -** Students said that the reflection questions helped them calm their mind after the strain of doing the challenges, and thus prepared them for the next challenge (*E.g. "It relaxes my mind. I was putting some strain here in the challenge... It helps me to calm down."*)
- **Enabled sustained engagement -** Students also said that a score associated with each reflection and evaluation question kept them engaged and alert, and made them focus on the questions since they did not want to lose points. (*E.g. "they were giving points for the right answer, and for every wrong answer there was explanation. It was motivating. Ya, we should get those points."*)

8.6 Local learning theory: VeriSIM enables students to develop an effective mental model of the design

In previous chapters, we developed the idea of how identifying and modelling scenarios in the design is essential for building an effective mental model of the design, which in turn enables students to effectively evaluate design diagrams against the given requirements. In this chapter, we described how students identified and modelled scenarios using features in VeriSIM. Based on the findings from the interaction logs and focus group interview, we propose the following local learning theory of how students are identifying and modelling scenarios in the design. Figure 8.6 and Table 8.4 summarizes the key aspects of the local learning theory, and how activities in VeriSIM are helping students perform effective software design evaluation.

Students develop knowledge of the problem domain and design diagrams based on the activities in 'Problem Understanding Stage'. In these activities, students analyse the given requirements and design diagrams, and answer questions which test their knowledge of the requirements and design diagrams.

Students' understanding of elements of the mental model, such as the dynamic behaviours (control flow and data flow), main goals and scenarios are developed when they solve challenges in the 'Design Tracing Stage'. In earlier challenges (Challenge 1 and 2), students analyse the scenario, the given model (state diagram), and the given design diagrams. They map parts of the scenario to the corresponding state. They identify errors in the model and use the class diagram and sequence diagrams to rectify the errors. They then move on to the next part of the scenario and its corresponding state. In later challenges (Challenge 3 and 4), students identify sub-parts of the scenario, and identify variables and entities which are relevant for this part. They follow a similar process of construction and refinement of the model. In all challenges, students execute the model and evaluate its correctness. As students progressively construct the state diagram in each challenge, they are able to map the scenario to the state diagram and systematically trace the control flow and data flow of the given scenario. The affordance to manipulate and execute the model further enhances their understanding of the control and data flow in each scenario, and also helps in developing the relationship between different design diagrams.

Students' ability to simulate alternate scenarios are developed by the scenario branching strategy. As they construct the scenario tree, they visualize different scenarios for each requirement. The strategy also helps them structure and decompose the given problem, thereby developing their design diagram and problem domain knowledge as well.

Students are able to describe the purpose of the design tracing and scenario branching strategy, and also articulate the relation between the two strategies. Consider the following interview excerpt -
"Design tracing is breaking up the different scenarios in the CMAP into different scenarios. So we could.. in scenario branching, we are putting all the scenarios in one map, and in design tracing, we are taking each single scenario and working on that scenario, and how it is getting executed, and we are seeing the node how it is working." From this excerpt, we can see that students are able to relate the scenario branching and the design tracing strategy. They are able to understand the zoomed out view which scenario branching provides, and the zoomed in view of each of these scenarios which they are able to do using design tracing.

Students are also able to reflect on how they will use the strategies learnt when they encounter a new design problem. Consider the following excerpt from an interview -
"In my case, I will use first make the use case of the system, and the user who will be the actor. Then I will create all the classes related to the entities present. And after that I will create the scenario what the user will be doing... So the listing out of scenario will be done by me. And after that, looking at the links of the scenario, I will be making the sequence part, and sequence part will be implemented by the state diagrams. So that the proper execution of the program can be done." From this excerpt, we see that students are able to appropriate how they will augment the strategies learnt in their previous understanding of the software design process.

Hence, the VeriSIM pedagogy helps students develop effective mental models of the design, by identifying and modelling various scenarios in the design. Students are able to develop a broad understanding of the requirements and the design, by identifying relevant scenarios based on the requirements. They are also able to simulate the dynamic behaviour of each of these scenarios using the design tracing strategy. The broad as well as deep exploration of various scenarios in the design is enabling students evaluate the design against the given requirements.

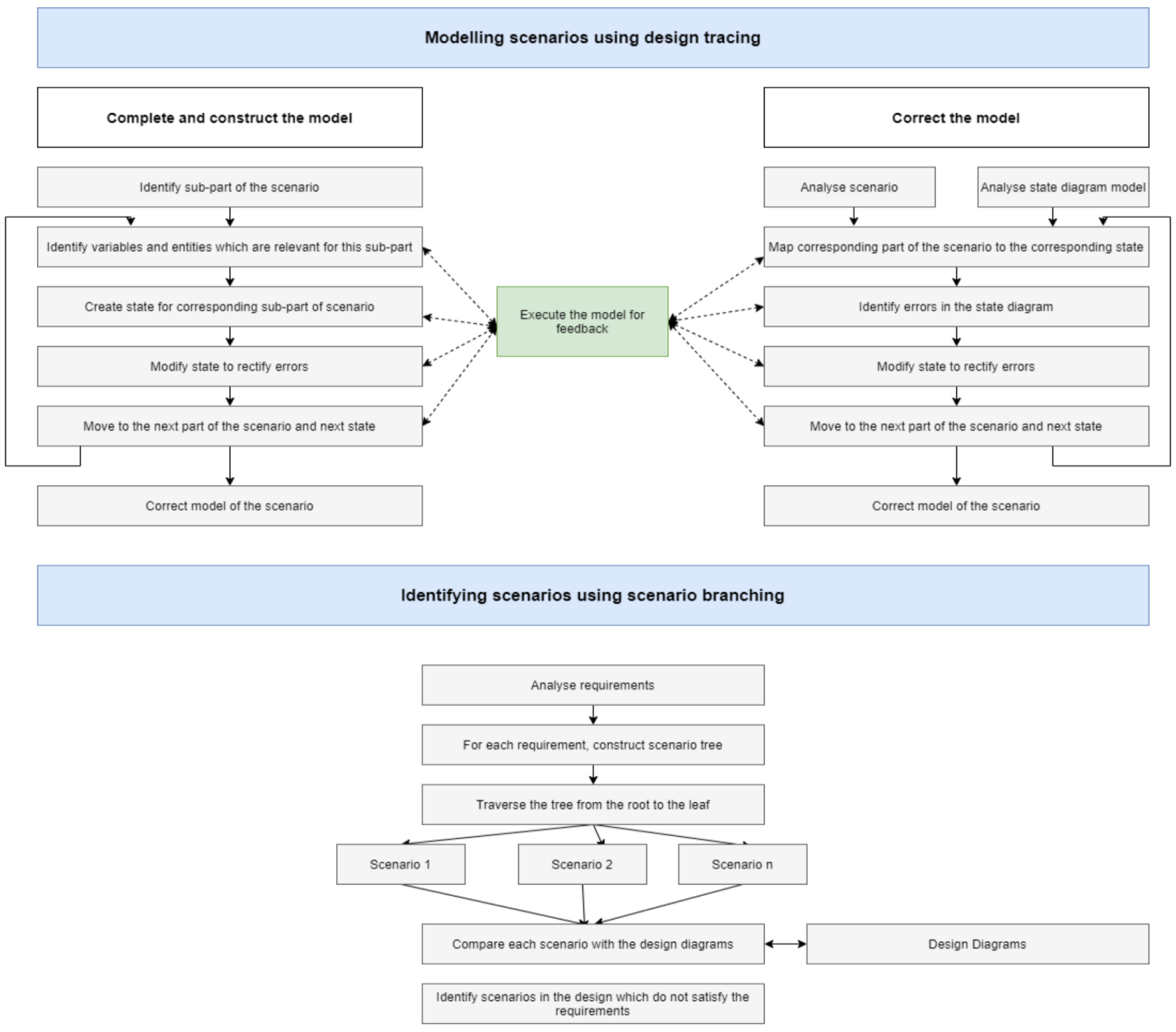

Modelling scenarios using design tracing
Complete and construct the model
Identify sub-part of the scenario
Identify variables and entities which are relevant for this sub-part
Create state for corresponding sub-part of scenario
Modify state to rectify errors
Move to the next part of the scenario and next state
Correct model of the scenario
Execute the model for feedback
Correct the model
Analyse scenario
Analyse state diagram model
Map corresponding part of the scenario to the corresponding state
Identify errors in the state diagram
Modify state to rectify errors
Move to the next part of the scenario and next state
Correct model of the scenario
Identifying scenarios using scenario branching
Analyse requirements
For each requirement, construct scenario tree
Traverse the tree from the root to the leaf
Scenario 1
Scenario 2
Scenario n
Compare each scenario with the design diagrams
Design Diagrams
Identify scenarios in the design which do not satisfy the requirements

Figure 8.6: Key aspects of the local learning theory based on the VeriSIM pedagogy

Table 8.4: Summary of how activities in VeriSIM help in performing software design evaluation

Activities in VeriSIM	What students do	How it helps them
Problem Understanding Stage	• Analyse software design diagrams and requirements • Attempt reflection and evaluation activities	Students develop knowledge of the problem domain and design diagrams
Design Tracing Stage (Challenge 2)	• Analyse scenario and state diagram model • Map corresponding part of scenario to corresponding state • Identify errors in the model • Modify state to rectify errors	Students are able to build relationships between main goals, control flow, data flow, and diagram surface elements
Design Tracing Stage (Challenge 3 and 4)	• Identify sub-parts of the scenario • Identify variables and entities which are relevant to the sub-parts • Construct and revise the model	Students are able to identify relevant parts of the model i.e. the data variables, events and states, and their appropriate values
Reflection Stage	• Reflect on the design tracing strategy and how it will be useful for them in the future	Students are able to abstract the strategy and think of how they can apply it in different contexts
Scenario branching strategy	• Construct a scenario tree for each requirement • Traverse the tree to generate scenarios • Compare each scenario with the design diagrams	Students are able to generate scenarios and visualize whether these scenarios satisfy the given requirements.

Chapter 9

Discussion

9.1 Overview of the Research

In this thesis, we reported two design-based research (DBR) cycles of problem analysis, solution design and development, evaluation and reflection. In the first DBR Cycle, we analysed literature in computing education research and empirical software engineering in order to identify cognitive processes and strategies experts use in the context of software design. The key insight is that experts create rich mental models of the design, and use various reasoning techniques to simulate scenarios in the design. On the other hand, an analysis of the literature shows that difficulties which novices face seem to stem from an incorrect and incomplete mental model of the design. These difficulties have been explored in the context of creating designs, but sufficient emphasis has not been given on how novices approach evaluating design diagrams against the given requirements. We followed this line of investigation in Study 1a and Study 1b, where we characterized student responses as well as their process of evaluating design diagrams. These studies revealed that students' mental models did not accommodate the control and data flow in designs, and mainly focussed on superficial aspects of the design diagrams. They did not simulate alternate scenarios while evaluating the design. They also proposed new functionalities and requirements outside the model boundary of the design and the requirements, instead of evaluating the design diagrams against the given requirements.

These findings from the problem analysis phase of DBR Cycle 1 shows that effective evaluation of a given design depends on students' ability to create a rich mental model and to simulate the control as well as data flow of the design. In order to help students do this, we proposed VeriSIM - a model-based learning pedagogy which enabled learners to identify and model various scenarios in the design. We operationalised the pedagogy into the VeriSIM learning environment, which incorporated features of model construction, revision, visualization, and evaluation. We hypothesized that as students created models of various scenarios in the design, they would be able to evaluate design diagrams better. We examined this hypothesis in Study 2, where we investigated the effectiveness of VeriSIM in students' ability to trace scenarios and evaluate the design diagrams against the given requirements. Our findings show that by scaffolding students to trace given scenarios in the design, there is a significant improvement in their ability to trace scenarios, but not in their ability to identify scenarios which do not satisfy the given requirements.

In the second DBR Cycle, our reflections from the previous cycle showed that students require explicit help to generate and identify scenarios which do not satisfy requirements. This led us to introduce a second module in VeriSIM - the scenario branching strategy, in which students use a mapping tool to generate alternate scenarios in the design. In Study 3, students went through both modules of VeriSIM. The findings show that there is an improvement in their ability to identify alternate scenarios in the design which do not satisfy the given requirements. Students also perceived that the mapping tool helped them in structuring a given problem, as well as think of alternate scenarios in the design.

In this cycle, we also investigated how features in VeriSIM are contributing to student learning. We analysed the interaction logs and student perceptions in order to identify the usefulness of various features in VeriSIM. Based on these findings, we proposed a local learning theory of how students were identifying and modelling scenarios in the design, which showed that the activities in VeriSIM helped students effectively evaluate the given design against the requirements.

9.2 Answering the Research Questions

The studies conducted in the DBR Cycles enabled us to answer the research questions proposed in the thesis. We present these answers below:

RQ 1: How do students evaluate a design against the given requirements?

We answered RQ 1 by conducted two studies (Study 1a and 1b) with students. Studies with students showed that they were unable to create rich mental models of the design, and instead focussed on surface-level parts of the design. They were able to identify the main goals of the design diagrams, but were unable to explain the control flow and data flow based on the execution of different scenarios. We also found that they added new functionalities into the design, instead of adhering to the task of evaluating the design diagrams against the given requirements. Based on the findings from the novice studies, we claim that to effectively perform software design evaluation, **students need explicit training in creating accurate mental models of the design. This involves them identifying relevant scenarios, and modelling the control flow and data flow of these scenarios.** These findings informed the design of the VeriSIM pedagogy, and its operationalisation into the VeriSIM learning environment.

RQ 2 and 3: What are effects of the VeriSIM pedagogy in students' ability to evaluate a design against the given requirements?

In Study 2 and 3 students interacted with VeriSIM learning environment. Analysis of students' pre-test and post-test responses, as well their responses in the focus group interviews and feedback forms show a change in students' processes while evaluating software designs, as well as in their attitudes towards modelling and software design.

In Study 2, we observed that there is a significant difference between how students traced a scenario before and after interaction with VeriSIM. Students' focus shifted from a superficial understanding of scenarios, to an understanding which involved identification of relevant data variables, events and their change of state on execution of a given scenario. We believe this change is due to activities in VeriSIM which helped students model different scenarios in the design. Students also perceived that the VeriSIM pedagogy helped them understand the inter-relationship between different design diagrams. Based on these findings, we claim that interaction with **VeriSIM helps learners develop a rich mental model of the given design and requirements**.

In Study 3, we observed that there is a change in how students evaluate a given design against the requirements. Students' understanding of evaluation evolved from merely adding new functionalities and requirements, to a process which involved identifying alternate scenarios in the design which violate the given requirements. We believe that the activities in VeriSIM such as the scenario branching strategy facilitated this change. Hence, we claim that **VeriSIM enables learners to effectively evaluate design diagrams against the given requirements**.

We also observed that although students had undergone a software engineering course prior to interactions with VeriSIM, the activities in VeriSIM prompted a change in their attitude towards modelling and design. VeriSIM helped them understand the advantages of designing a class diagram before actually starting with the project. Some students also said that in previous years, they used to directly start coding without creating a design, which resulted in several errors in their code. They now realised that creating designs at the start would help them create correct programs, save time, as well as facilitate discussion among team members.

The above advantages of design are obvious, and the key reason why it is taught. These advantages were not explicitly mentioned, either in the pedagogy or by the researchers conducting the studies. Hence, we believe interactions with VeriSIM helped students realise these advantages, and **brought about a shift in their attitude towards modelling and software design.**

RQ 4: How are features in the VeriSIM learning environment contributing towards student learning?

We answered RQ 4 by analysing logs of students interacting with the VeriSIM learning environment, and student responses in the focus-group interviews. We found that the activities in the design tracing stage enabled students to progressively trace scenarios in the design. These findings give evidence for the appropriateness of the model progression activities in enabling learners to construct models of various scenarios in the design. Second, the model execution and visualisation features in the Design Tracing Stage helped learners simulate the control and data flow of various scenarios in the design, by helping students map a particular state to the corresponding part of the scenario. These features scaffolded learners to model various scenarios in the design. Third, the scenario branching strategy enabled students to structure the design problem, and break down the problem into scenarios, thereby enabling them to identify scenarios which were missing in the design diagrams. We also saw instances of students being able to abstract the purpose of the design tracing and scenario branching strategies, as well as reflect on how they will apply these strategies in a new design problem.

9.3 Generalizability

The VeriSIM pedagogy enables students to create a rich mental model of various scenarios in software design. We believe that the pedagogy can be extended to teaching-learning of programming as well. There are existing strategies such as program tracing which help students develop an accurate mental model of program execution (Cunningham et al., 2017; Xie et al., 2018). The model order progression activities can be used to enable students to effectively trace a given program. We discuss this in more detail in our future work (Section 10.2.1).

We have restricted our scope to evaluation of class and sequence diagrams. The class and sequence diagrams are the most commonly used structural and behavioural diagrams (Dobing and Parsons, 2006). Since the VeriSIM pedagogy helps students identify the control and data flow elements from the class and sequence diagrams, we conjecture that the underlying principle of identifying and modelling scenarios can be extended to other design diagrams such as object diagrams, activity diagrams, collaboration diagrams etc.

We also conjecture that identifying and modelling scenarios in the design can help students in creating designs as well. As students create a design based on the given requirements, they can identify and model various scenarios in their own design. We also saw evidence of students repurposing the strategies learnt in VeriSIM into the design creation process (Section

8.6). Hence, we believe that the VeriSIM pedagogy can be used to help students in the design creation process as well.

The model-based learning paradigm has been the underlying theoretical basis for activities and features in VeriSIM. We have provided arguments and evidence for the appropriateness of model-based learning for teaching-learning of software design evaluation. We speculate that the key ideas of the VeriSIM pedagogy can be used to evaluate designs in other areas as well, such as electronic circuit design, mechanical design, product design etc. Even in such design contexts, a set of requirements and specifications are provided along with the design. In order to effectively evaluate the design, designers require adequate domain knowledge (e.g. laws concerning electrical flow, laws governing mechanical systems) as well as knowledge of various components of the design (e.g. how a transistor works). They can then identify and simulate scenarios in the design which do not satisfy the requirements. For example, in the case of an electronic circuit design, designers can think of scenarios with different signal ranges and supply voltage ranges for which a circuit faces issues. They build mental models of these scenarios by identifying relevant parts of the circuit. They may then use mathematical models or simulators/emulators to predict the output of the scenario model. Based on the output, designers speculate whether the given scenario satisfies the specifications for these ranges. Hence, as designers identify different scenarios and model them, they can examine whether specific scenarios satisfy the given requirements.

We conjecture that various aspects of the VeriSIM pedagogy can foster the above mentioned practices in other design contexts as well. Learners can be scaffolded to develop relevant knowledge about the domain and components of the model. Instructors can use model progression activities (analogous to that in VeriSIM) to help students develop an accurate mental model of the given design. Students can also be trained to identify scenarios which do not satisfy the design, by using external representations like the scenario branching tree. Hence, we hypothesize that the essence of the VeriSIM pedagogy i.e. enabling students to identify and model scenarios in the design can transfer to teaching-learning of design evaluation in other contexts as well.

9.4 Limitations

Limitations related to learner characteristics

The subjects of studies in this thesis were undergraduate engineering students enrolled in bachelor programmes in computer science or information technology, who were from urban colleges, who were proficient in using computers, and whose medium of instruction was English. All of them were familiar with basic UML diagrams, and the software design process. This is also confirmed by students' positive self-perception of their UML design knowledge in a registration form prior to studies. However, we believe that students have varying prior knowledge and experiences in working with software designs. We have not investigated the effect these variations in prior knowledge have on the results of our studies. Students' prior experience in working with software designs, would likely have affected their interactions with VeriSIM as well as how they performed software design evaluation. The effect of varying prior knowledge can be examined in future studies.

In addition to prior knowledge, various personal, social, emotional and cognitive characteristics also affect learning experiences. We also believe students' motivation in performing software design tasks, as well as computing in general could have also affected the results of our studies. In Studies 1a, 2 and 3, we used convenience sampling based on the availability of students, and hence could not account for their motivation and interest in software design. Variations in these affective factors could have also affected the findings. Further studies can be conducted to tease apart the role of such affective factors in software design evaluation.

Limitations due to scoping the construct and skills involved in 'evaluation'

In this thesis, we have looked at developing students' design diagram evaluation skills. We have restricted our focus to semantic deficiencies, and emphasised on helping students identify and model relevant scenarios. In our studies, we have taken the perspective that "effective" evaluation was done by students who identified relevant scenarios not adhering to the requirements.

This scoping of evaluation comes with its limitations.

First, a holistic perspective to evaluation involves (in addition to semantics), other aspects such as syntactic and pragmatic deficiencies, as well as non-functional requirements such as modularity, extensibility etc. These perspectives of evaluation go beyond identifying relevant scenarios and mapping requirements to design diagrams. The effect of these aspects have not been identified in this thesis.

Second, effective evaluation (much like any other software engineering practice) depends on individual as well as inter-personal skills. Software engineering expertise is characterised by internal personality attributes, and attributes regarding engagement with others, in addition to developing technical expertise (Li, 2016). The software design process involves collaboration and deliberation with multiple stakeholders, and other team members. Training students to perform evaluations in such ecologically valid settings would involve helping students collaborate with each other, deliberate on the decisions made, and communicate these decisions effectively. In this thesis, we have focussed on helping students develop rich and accurate mental models of the design. How these models of the design should be communicated it to a larger team is also essential in taking the design forward to the next stage. Developing these skills would also contribute towards effective evaluation, but has not been investigated in this thesis.

Limitations related to characteristics of software design evaluation problems

Students were provided with different software designs across studies in the thesis, such as the design of an ATM system, a library management system, an automated door locking system, and a streaming website. These designs were taken from problem domains familiar to learners. Our findings are thus limited to examining students' evaluation processes in familiar problem domains. These designs were also fairly simplistic, and did not capture the inherent complexity of the actual real-world system. However, we believe that the key goal of the VeriSIM pedagogy, of enabling students to build effective mental models of the design by simulating scenarios, will aid them as they evaluate complex designs as well. This can also be investigated in future studies, where the VeriSIM learning environment is modified to include fairly complex designs as well.

Limitations of the UML formalism

Although Unified Modelling Language (UML) is a general-purpose modelling language, it has certain limitations. Many software engineers do not use UML due to its lack of context, and overheads of understanding notation (Petre, 2013). Others retrofit it as a means to document the development of a software system, or as a creative ‘thought-tool’ at the beginning of the software design life-cycle (Petre, 2013).

Despite these limitations, Petre notes that UML plays an important role in software engineering education (Petre, 2014). UML provides students a medium for “model-based thinking”, and equips them with a collection of representations and reasoning tools to engage with software design problems. The focus of this thesis has also been to aid students in engaging with the semantics of the design, by identifying and modelling various scenarios in the design. We believe that this helps students focus on the correctness and completeness of the design, as opposed to strict syntactic issues, which may not be relevant for them in the future.

9.5 Implications

This thesis has implications for computer science instructors as well as computing education researchers.

9.5.1 Implications for Computer Science Instructors

In this thesis, we brought out the gap that evaluation of software design diagrams is not given sufficient emphasis in the curriculum. Our initial studies to understand how learners approach the evaluation task show that they are not able to construct a rich and detailed model of the design. These findings on student difficulties can inform software design instructors in their teaching. Based on the findings of studies conducted in this thesis, we provide the following guidelines for the teaching of software design diagram evaluation.

1. **Help students understand the problem domain and requirements -** Problem understanding is a vital part of the design process. Instructors should ensure that students sufficiently understand the domain and the requirements, before jumping in to verifying the design diagrams. In the VeriSIM pedagogy, we have shown that the scenario branching strategy can be used to help students understand the problem space. Teachers can use this strategy to help students simulate scenarios in the design which map to the requirements and also check if there are scenarios which are not adhering to the given requirements.

2. **Equip students to identify specific scenarios and model them -** Once students identify relevant scenarios, instructors should help them model these scenarios. In the process of modelling these scenarios, students are forced to closely analyse the design diagrams and simulate the control and data flow across diagrams. This ensures that they develop a rich mental model of the design, which is essential for them to evaluate the design against the given requirements.

3. **Provide activities that help students progressively model scenarios in the design -** The VeriSIM pedagogy gives evidence that the model progression of activities helps students in modelling scenarios. Instructors can provide activities of observing, correcting and completing a model, before students model scenarios.

4. **Provide opportunities for reflection and metacognition -** Throughout the evaluation process, instructors should present students with opportunities to reflect on their thinking, why they performed certain actions, and what they must do to solve the evaluation task.

5. **Use the evaluation tasks as a natural extension to creation tasks -** Providing students with evaluation tasks before creation has several benefits. First, it provides students examples of how designs have been created from the given requirements, and what are defects which should be avoided. Second, they can use the strategies they learnt while evaluating other designs, in order to create as well as evaluate their own designs. Students can simulate and model scenarios in their own created designs to ensure that it adheres to the given requirements.

Findings from this thesis also provides implications for teaching-learning of software design. First, teaching software design should provide students with tools and strategies to create models of the design, rather than focussing on teaching the notations and syntax of UML diagrams. This is because the skill of effective modelling is more likely to be used in industry, rather than a specific modelling language.

Second, the mental model for design diagrams outlined in this thesis can also serve as an aid for instructors to design appropriate curricula and learning pathways for teaching-learning of software design. Instructors can initially scaffold students to identify the diagram surface elements and main goals of design diagrams, followed by a deeper exploration of the control flow and the data flow. This can help students build an integrated understanding of different design diagrams and construct effective mental models of the entire design.

9.5.2 Implications for Computing Education Researchers

This thesis extends research in computing education in three fronts - extending the theory of program comprehension, characterization of students' mental models in computing, and use of the model-based learning paradigm in computing education. First, this work extends the existing theory of program comprehension. The Block model has been used as an educational framework for teaching-learning of program comprehension. However, the model has predominantly investigated comprehension of code artifacts. We expand this existing notion to design diagrams as artifacts. We built upon existing work by exploring students' comprehension and evaluation processes, and the difficulties they face in the context of design diagrams. Computing education researchers can conduct further studies to validate as well as extend on our adapted mental model for design diagrams.

Second, computing education researchers can extend the findings of this thesis by exploring variations in student models as they evaluate different design diagrams and problems of varying complexities. How students' mental models vary when they are presented with other structural and behavioural diagrams (like activity diagram, object diagrams etc.) can be explored. Studies can also be conducted with problems from unfamiliar domains to examine whether the behaviour of adding new functionalities persist. Findings from such studies can further contribute to the theory of how students evaluate design diagrams against the given requirements.

Finally, the thesis gives evidence for the model-based learning paradigm as an appropriate pedagogy for software design. This opens the space for researchers to investigate this paradigm in other aspects of software design, as well as programming. Variations of this paradigm can be applied to create pedagogies and designs which investigate learning and its effectiveness in other contexts.

Chapter 10

Conclusion

10.1 Contributions of the Thesis

This thesis provides contributions to pedagogy in software engineering education, and extends research in computing education research as well as design of learning environments.

10.1.1 Contributions towards pedagogies for software design

The key contribution of this thesis is the VeriSIM pedagogy which helps students effectively evaluate software design diagrams using the design tracing and scenario branching strategy. These strategies can be directly used by instructors to train students in evaluating and creating software designs for the given requirements. We have also provided guidelines which instructors can use for teaching of software design evaluation (see Section 9.5.1).

The theoretical basis of the VeriSIM pedagogy is the model-based learning paradigm. Although model-based learning has been extensively studied and applied in science education research, it has not been explored in software design, or even computing education in general. This thesis provides an instantiation of the model-based learning paradigm in the context of software design. We believe this opens up the field for research in applying model-based learning paradigms in computing education - in contexts like programming and creating designs.

10.1.2 Contribution towards computing education research theory

This thesis also contributes to computing education in two fronts. First, it contributes towards a characterization of novices processes in software design evaluation. Based on our analysis of literature from expert practices in software design and strategies they use to evaluate designs, we found that experts create rich mental models of the problem domain and the design. They perform mental simulations on these models by simulating the control flow and data flow of various scenarios in the design. On the other hand, based on findings from novice studies, we found that novices focus on surface-level parts of the design, and show limited instances of data flow simulation. Novices also add new functionalities into the design rather than evaluating the design against the given requirements. Computing education researchers can apply and extend this characterization of novice processes for software design evaluation in varying contexts such as for different design diagrams and unfamiliar problem domains.

Second, the thesis contributes towards the existing theory of program comprehension by adapting the Block Model and extending the artifacts used to design diagrams (see Section 9.5.2). Computing education researchers can conduct studies to build on the mental model elements for design diagrams proposed in this thesis, and thus further extend existing theory on mental models as well as program and design comprehension.

10.1.3 Contribution towards learning environment design

In this thesis, we designed and developed the VeriSIM learning environment in order develop students' design diagram evaluation skills. VeriSIM is available online for anyone to use at the following link - **https://verisim.tech**. The learning environment can be directly used by instructors as well as students to be trained in evaluating design diagrams against the requirements.

The VeriSIM learning environment incorporates various design features such as the ability to construct, modify, and visualize the execution of a given scenario. This extends the current work in program visualization literature, which has primarily looked at visualizations of program execution. Learning environment designers can adapt and extend these features for other software design diagrams and contexts, as well as teaching-learning of programming.

10.2 Future Work

This thesis opens avenues for development as well as research in several areas.

10.2.1 Extending the model-based learning paradigm to program comprehension

This thesis provides evidence for the suitability of the model-based learning paradigm for design diagram evaluation. A natural extension of this paradigm is towards the teaching-learning of program comprehension. There have been studies which show that tracing is an effective strategy to help students comprehend a given program (Cunningham et al., 2017; Xie et al., 2018). However, how tracing can be taught is not given sufficient emphasis. The model-based learning paradigm and model order progression can serve as teaching strategies for tracing. For example, an instructor can scaffold students to first explore a trace, then correct a trace, followed by completing an incorrect trace of a program. Such gradual progressions can help students develop expertise in comprehending a given program.

Effects of the benefits of model progression for program tracing can be investigated using a quasi-experimental research design, which compares students tracing programs with and without model progression. Findings from such studies can investigate the usefulness of model-based learning and the model progression activities in programming as well.

10.2.2 Developing an instructor interface for the VeriSIM learning environment

The current version of VeriSIM Module 1 involves challenges in which users trace scenarios for a single design problem - an automated door locking system. However, in order to be used by instructors in a classroom, we believe that several types of problems need to be provided to students. Hence, future work with regards to VeriSIM's development is providing an interface where instructors can add problems from different contexts and different types of design diagrams of varying complexities. A series of such problems can be used in a lab or classroom setting for sustained instruction in teaching-learning of design evaluation.

10.2.3 Using eye-tracking for a deeper understanding of how students evaluate a design

In Study 1b, we used video data as the primary data source to understand the strategies which students used to make sense of the design. We could only infer which diagram they were focusing on, but not on which part or aspect of the design diagram. Physiological sensors such as eye-tracking can provide an even more finer level of detail towards how students comprehend a given design. For example - the tracking device can provide details about certain parts or areas in the design diagrams students focus on the most. It can also provide accurate patterns of how students are switching between different diagrams, and what areas in these design diagrams they are focusing on. Such findings can further contribute towards strategies which students use to make sense of the design and also uncover difficulties and misconceptions encountered by students.

10.2.4 Conducting qualitative studies to gain deeper insights

The effectiveness of the VeriSIM pedagogy has been ascertained mainly by (1) Comparison of answers in pre-test and post-test (2) Student perceptions after using VeriSIM and (3) Analysis of student interaction logs. Findings from these data sources and analysis have shown that the VeriSIM pedagogy has caused changes in students' evaluation processes as well as their attitude towards design. In addition to these analysis methods, qualitative studies involving video capture, think-aloud and in-depth interviews can also be conducted as students interact with VeriSIM. These studies can lead to deeper insights and provide a deeper understanding of students' learning pathways during their interactions with VeriSIM.

10.2.5 Investigating the effect of evaluation on creation of designs

In introductory programming literature, skills like comprehension and debugging are essential skills which contribute towards effective programming. In fact, there is a recent emphasis towards introducing students to comprehend programs before they actually start creating programs (Xie et al., 2018; Schulte et al., 2010; Nelson et al., 2017). We conjecture that the same holds true for software designs as well. We believe that as students comprehend and evaluate designs, they will be able to create better designs. This conjecture can be tested conducting a quasi-experimental study comparing students who create a design versus students who first evaluate designs and then create a design. Findings from such studies can provide evidence for the usefulness of evaluation tasks in enabling students to create better designs.

10.3 Final Reflections

The main goal of this thesis has been to provide learners with scaffolds to help them better reason with a given design. I believe that training students to build and reason with software designs should be an important goal of computer science programmes, particularly because building and reasoning with software systems are essential activities software engineers perform on their job. I believe that this thesis has contributed towards enhancing students' competence in these activities.

Looking back, the idea of exploring this field of study had been planted even before I started my PhD. For my Masters thesis, I developed a tool which automatically ascertained if a set of sequence diagrams satisfy a given property. I found that programming a software to reason with software designs was difficult, but still tractable, since the tool behaves exactly the way I program it to behave. However, explorations in this thesis made me realise that teaching students to do it is much harder! What really goes on inside a person's mind as they interact with software systems? This thesis does shed some light to this question. Humans come with their own set of life experiences, prior knowledge and biases, and all of these have a part to play as one reasons with software artifacts. Findings from literature and novice studies made me realise the complex interactions which occur in an experts' mind as they reason with a software design, and the difficulties which novices face. Designing pedagogies to address these difficulties has also shown me that technological tools and affordances can certainly help students become better at reasoning with software systems.

My PhD journey has helped me make the transition from reasoning with software, to helping learners effectively reason with them. My experiences of conducting studies with students, designing learning environments, and evaluating its effectiveness has helped me become more aware of the challenges of doing educational research. However, it has also provided great satisfaction to see learners using the tools and scaffolds which I designed, to make small, yet significant improvements in their learning. I believe the PhD experience has been a great starting point to a research journey in computing education research which I hope to pursue.

Appendix A

Sample Student Consent Form

Consent to Participate in Educational Research

[Title: Evaluating a Software System Design]

You have been asked to participate in a research study conducted by Prajish Prasad from the Inter-Disciplinary Program in Educational Technology at the Indian Institute of Technology Bombay (IITB). The purpose of the study is to get feedback on a pedagogy to help students evaluate a software system design. You were selected as a possible participant in this study because of your educational background as a computer science under-graduate.

- **PARTICIPATION AND WITHDRAWAL**

Your participation in this study is completely voluntary and you are free to choose whether to be in it or not. If you choose to be in this study, you may subsequently withdraw from it at any time without penalty or consequences of any kind. The investigator may withdraw you from this research if circumstances arise which warrant doing so.

You will not be compensated for the participation. You should read the information below, and ask questions about anything you do not understand, before deciding whether or not to participate.

- **PURPOSE OF THE STUDY**

The purpose of the study is to get feedback on a pedagogy to help students evaluate a software system design

- **PROCEDURES**

If you volunteer to participate in this study, we would ask you to do the following things:

1. Perform a task which consists of verifying properties of a software system design
2. Participate in interview

We expect that the study will take about 2 hours. There will be a screen recording of your interactions on the computer and an audio recording during the interview.

- **POTENTIAL BENEFITS**

  - The students get familiar with authentic software design problems.
  - Learn how to correctly and effectively verify properties of a software system design.

- **CONFIDENTIALITY**

Any information that is obtained in connection with this study and that can be identified with you will remain confidential and will be disclosed only with your permission or as required by law.

We will not use your name in publications; however we may need to use your academic performance details if you give us permission.

- **IDENTIFICATION OF INVESTIGATORS**

If you have any questions or concerns about the research, please feel free to contact Prajish Prasad (prajish.prasad@gmail.com) or Prof. Sridhar Iyer, CDEEP IITB (sri@iitb.ac.in) with any questions or concerns.

SIGNATURE OF PARTICIPANT

I understand the procedures described above. My questions have been answered to my satisfaction, and I agree to participate in this study.

Name of Subject	
E-mail address	Contact No.
Signature of Subject	Date

Appendix B

Materials provided to students for Study 1b

You are a software developer in the software development team of “SafeHome Solutions”. SafeHome solutions is a startup which wants to create automated door locking systems for homes. After discussions with various stakeholders, an initial set of requirements have been captured. Based on these requirements, the design team has come up with an initial design of the product. They have now passed on the design to your team. Your team lead, Nisha wants to do a check on the design before passing it to her team for development. She approaches you to do this task.

She provides you with a set of key requirements for the automated door locking system. The requirements collected are -

1. If the passcode hasn't been set yet, the user can register and enter a required passcode.
2. When the user chooses the lock option, and enters the correct passcode, the door should lock. If the passcode is incorrect, the door remains unlocked.
3. When the user chooses the unlock option, and enters the correct passcode, the door should unlock. If the passcode is incorrect, the door remains locked.
4. The door should lock/unlock only if it is closed.

She also provides you with the design, which is a set of UML diagrams.(Provided on the system)

Task Objective: For each requirement, your task is to provide a logical explanation for how the design satisfies/does not satisfy the requirement. You are free to use any notation/diagrams to support your explanation.

B.1 UML Diagrams provided to Students

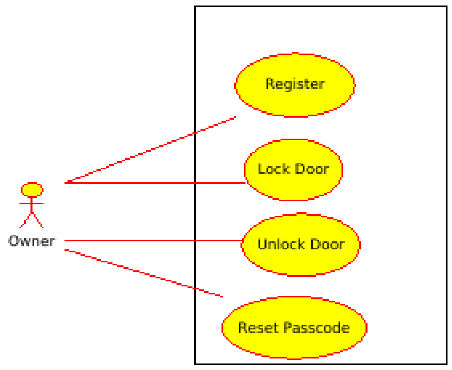


Figure B.1: Use case diagram of the door locking system

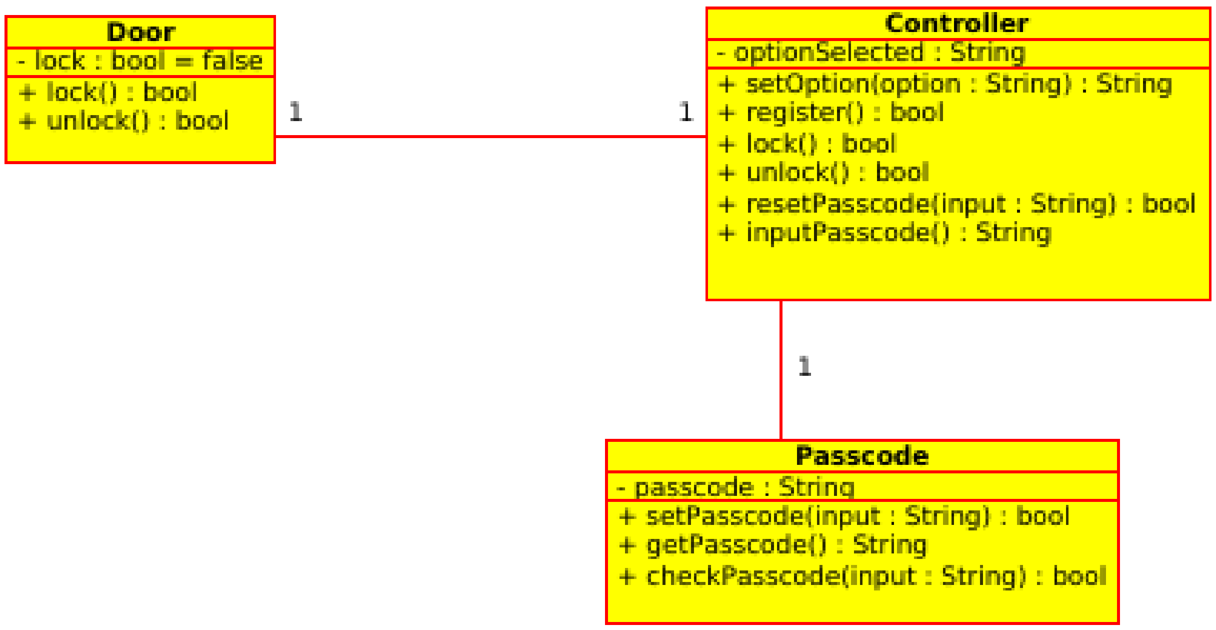

Door
- lock : bool = false
+ lock() : bool
+ unlock() : bool
1
1
Controller
- optionSelected : String
+ setOption(option : String) : String
+ register() : bool
+ lock() : bool
+ unlock() : bool
+ resetPasscode(input : String) : bool
+ inputPasscode() : String
1
Passcode
- passcode : String
+ setPasscode(input : String) : bool
+ getPasscode() : String
+ checkPasscode(input : String) : bool

Figure B.2: Class diagram of the door locking system

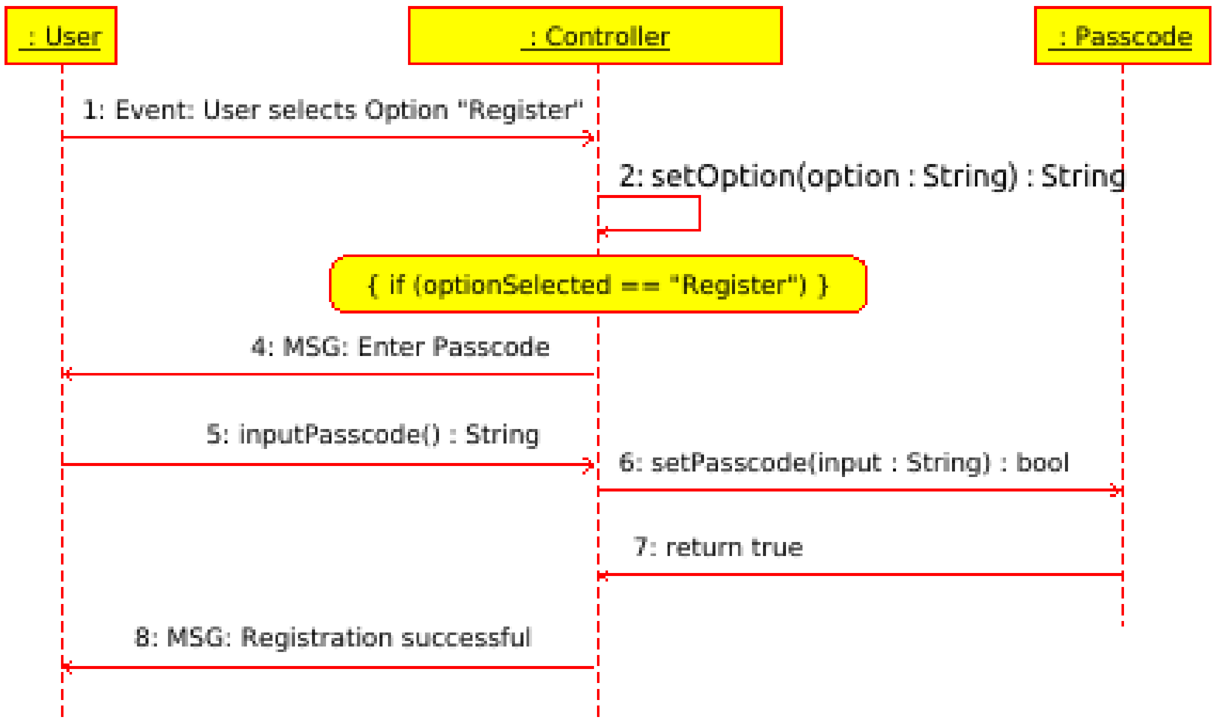

: User
: Controller
: Passcode
1: Event: User selects Option "Register"
2: setOption(option : String) : String
{ if (optionSelected == "Register") }
4: MSG: Enter Passcode
5: inputPasscode() : String
6: setPasscode(input : String) : bool
7: return true
8: MSG: Registration successful

Figure B.3: Sequence diagram for the register use case

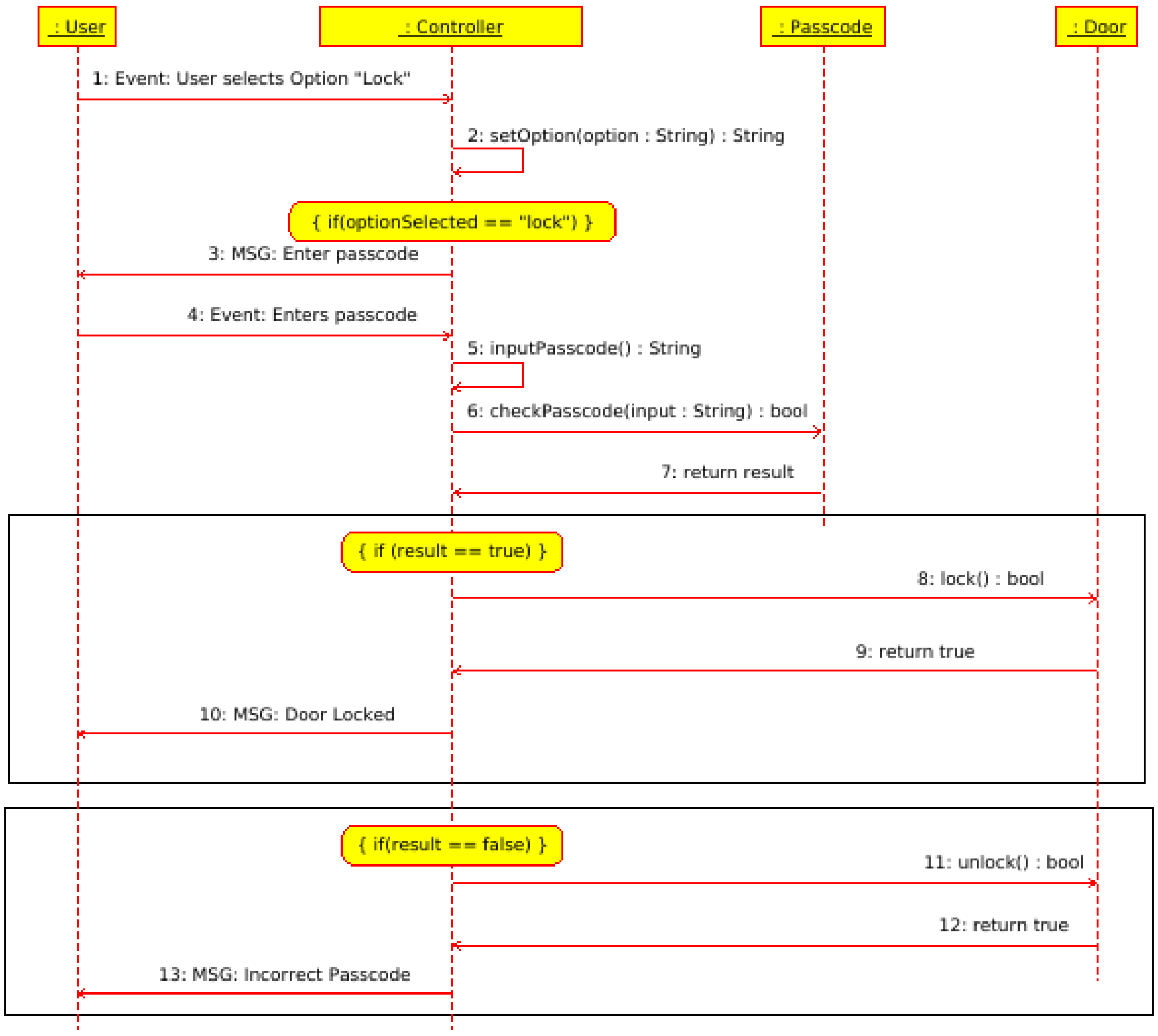

: User
: Controller
: Passcode
: Door
1: Event: User selects Option "Lock"
2: setOption(option : String) : String
{ if(optionSelected == "lock") }
3: MSG: Enter passcode
4: Event: Enters passcode
5: inputPasscode() : String
6: checkPasscode(input : String) : bool
7: return result
{ if (result == true) }
8: lock() : bool
9: return true
10: MSG: Door Locked
{ if(result == false) }
11: unlock() : bool
12: return true
13: MSG: Incorrect Passcode

Figure B.4: Sequence diagram for the lock door use case

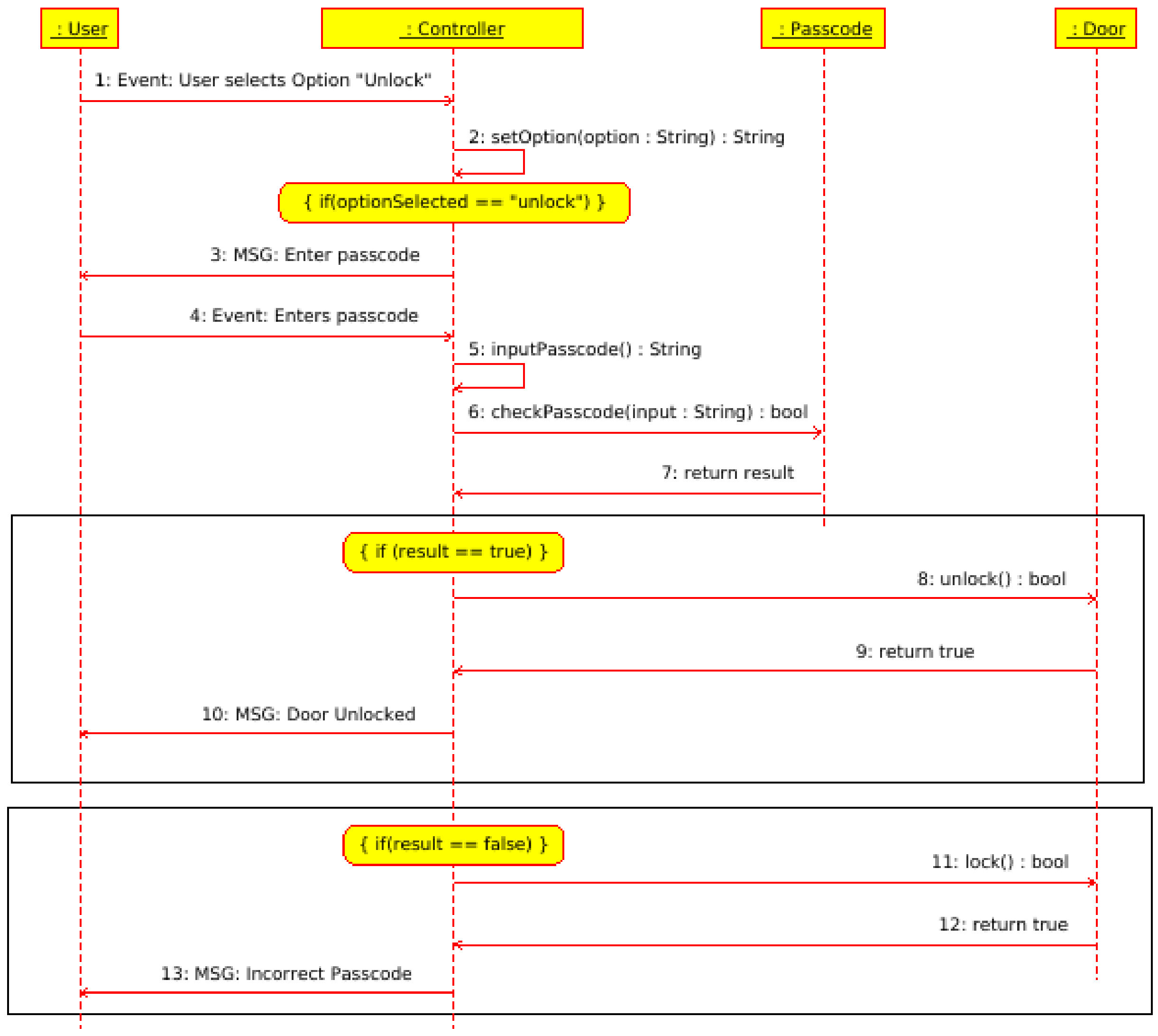
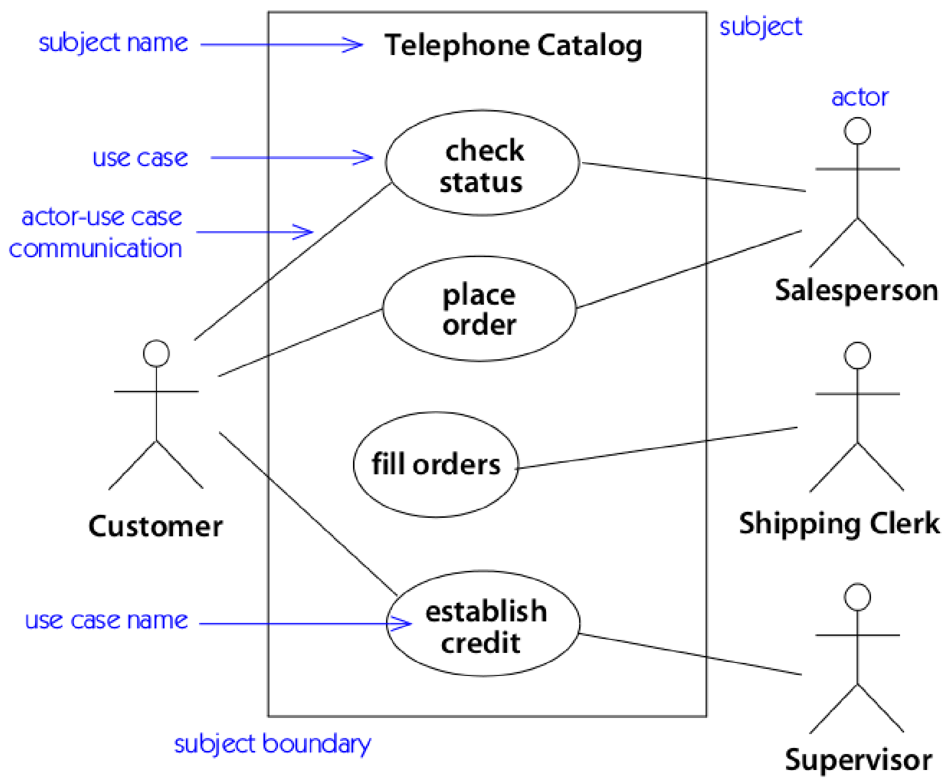

: User
: Controller
: Passcode
: Door
1: Event: User selects Option "Unlock"
2: setOption(option : String) : String
{ if(optionSelected == "unlock") }
3: MSG: Enter passcode
4: Event: Enters passcode
5: inputPasscode() : String
6: checkPasscode(input : String) : bool
7: return result
{ if (result == true) }
8: unlock() : bool
9: return true
10: MSG: Door Unlocked
{ if(result == false) }
11: lock() : bool
12: return true
13: MSG: Incorrect Passcode

Figure B.5: Sequence diagram for the unlock door use case

Overview of a Use Case Diagram

A **use case diagram** at its simplest is a representation of a user's interaction with the system that shows the relationship between the user and the different use cases in which the user is involved. A use case diagram can identify the different types of users of a system and the different use cases and will often be accompanied by other types of diagrams as well.

The purpose of the use case diagrams is simply to provide the high level view of the system and convey the requirements in layman's terms for the stakeholders. Additional diagrams and documentation can be used to provide a complete functional and technical view of the system.



Overview of a Class Diagram

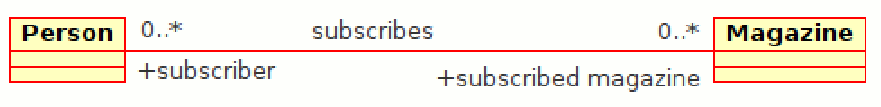
In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

The class diagram is the main building block of object-oriented modelling. It is used for general conceptual modelling of the systematic of the application, and for detailed modelling translating the models into programming code.

In the diagram, classes are represented with boxes that contain three compartments:

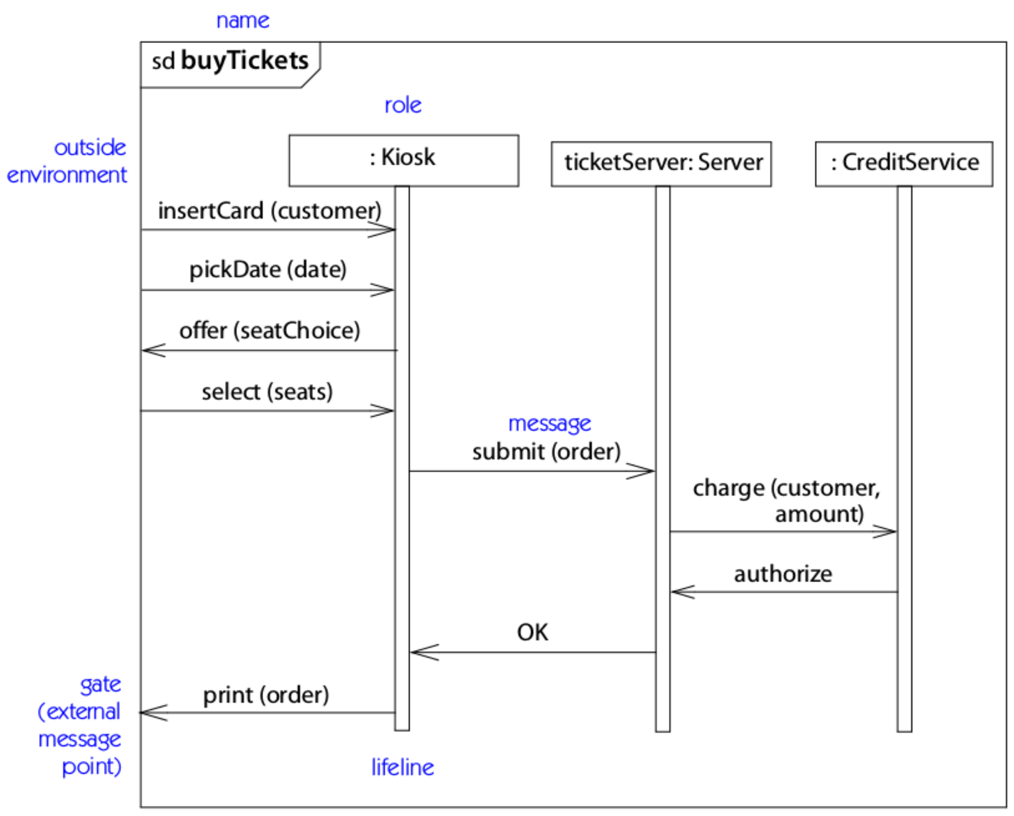
- The top compartment contains the name of the class. It is printed in bold and centered, and the first letter is capitalized.
- The middle compartment contains the attributes of the class. They are left-aligned and the first letter is lowercase.
- The bottom compartment contains the operations the class can execute. They are also left-aligned and the first letter is lowercase.

In the design of a system, a number of classes are identified and grouped together in a class diagram that helps to determine the static relations between them. With detailed modelling, the classes of the conceptual design are often split into a number of subclasses.



Overview of a Sequence Diagram

A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. A sequence diagram shows, as parallel vertical lines (*lifelines*), different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur. This allows the specification of simple runtime scenarios in a graphical manner.



Appendix C

Participant Registration Form used in Study 2 and 3

Understanding Software Design Session - SIES - Registration Form

Our research team has designed a technology-enhanced learning environment, for helping students get a better understanding of software design diagrams(UML). You will be introduced to a technique called design tracing. VeriSIM contains various challenges and reflection questions which will help you understand and apply the design tracing technique on a given design.

WHY SHOULD YOU ATTEND THIS SESSION
When you graduate and enter the software industry, you will be spending your first several months working on an existing projects and developing additional features based on new requirements. These activities require you to explicitly understand the software design.

At the end of the session, you will be able to:
1. Understand the purpose of class diagram and sequence diagram in a given design
2. Understand and apply the design tracing technique to trace a given scenario
3. Visualize how data and events can be used to trace a given scenario

* Required

1. Email address *

____

Basic Information

Please provide following information to register for the workshop

2. Name *

Your name as you want it to appear in your certificate

____

3. Gender *

Mark only one oval.

- ⬭ Female
- ⬭ Male
- ⬭ Prefer not to say
- ⬭ Other: ____________________

4. Age *

____

5. College *

Name of your college as you want it to appear in your certificate

____

6. Degree *

Mark only one oval.

- ⬭ B.E
- ⬭ B. Tech
- ⬭ B. Sc
- ⬭ Other: ____________________

7. Year *

Mark only one oval.

- 1st
- 2nd
- 3rd
- 4th
- Other: ____________________

8. Course *

Mark only one oval.

- Computer Engineering
- Information Technology
- Other: ____________________

9. Overall percentage scored in last semester *

You can convert CGPA to percentage by multiplying it by 100

Mark only one oval.

- 90 - 100%
- 80 - 90%
- 70 - 80%
- 60 - 70%
- 50 - 60%
- 40 - 50%
- Less than 40%
- Other: ____________________

Rate your confidence regarding your knowledge of the following

1 indicates "Not confident at all", 5 indicates "Absolutely confident". If a specific term or task is totally unfamiliar to you, please mark 1

10. Understand the object-oriented paradigm in design *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	⬭	⬭	⬭	⬭	⬭	Absolutely confident

11. Read and understand a given class diagram *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	⬭	⬭	⬭	⬭	⬭	Absolutely confident

12. Read and understand a given sequence diagram *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	⬭	⬭	⬭	⬭	⬭	Absolutely confident

13. Understand the purpose of using class diagrams in software design *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

14. Understand the purpose of using sequence diagrams in software design *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

15. Identify defects in a set of UML diagrams created by someone else *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

16. Identify defects in a set of UML diagrams that I have created myself *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

17. Create software design diagrams(Eg: UML) for a given set of requirements *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	⬭	⬭	⬭	⬭	⬭	Absolutely confident

Consent Form

STUDY TITLE: Study of student use of a learning environment for Software Design

You have been asked to participate in a research study conducted by Prajish Prasad from the Inter-Disciplinary Program in Educational Technology at the Indian Institute of Technology Bombay (IITB). The purpose of the study is to get feedback on a learning environment which helps students understand and comprehend software design diagrams. You were selected as a possible participant in this study because of your educational background.

PARTICIPATION AND WITHDRAWAL
Your participation in this study is completely voluntary and you are free to choose whether to be in it or not. If you choose to be in this study, you may subsequently withdraw from it at any time without penalty or consequences of any kind. The investigator may withdraw you from this research if circumstances arise which warrant doing so.
You should read the information below, and ask questions about anything you do not understand, before deciding whether or not to participate.

- In this study you will be asked to go through activities in the learning environment

- Your solutions will be used for research purposes only by the investigators of this study.

- Participating in this research study is voluntary. You have the right not to answer any question, and to stop your participation in the study at any time. We expect that the study will take upto 5 hours.

- You will not be compensated for the participation. You will be provided a participation certificate from Inter-Disciplinary Program in Educational Technology, IIT Bombay

- We will not use your name in publications; however we may need to use your academic qualification details if you give us permission.

- We would like to capture user logs of your interaction with the learning environment. so that we can use it for reference while proceeding with this study. If you grant permission for this interaction log capture, you have the right to revoke recording permission and/or end your participation at any time.

- We would like to record the audio of your interview so that we can use it for reference while proceeding with this study. If you grant permission for this interview to be recorded, you have the right to revoke recording permission and/or end your participation at any time. If we use your voice anywhere it will not be identified by name.

I understand the procedures described above. My questions have been answered to my satisfaction, and I agree to participate in this study. I have been given a copy of this form.

Please contact Prajish Prasad (prajish.prasad@gmail.com) or Prof. Sridhar Iyer, IDP ET IITB (sri@iitb.ac.in) with any questions or concerns.

18. I give permission for the following information to be included in publications resulting from this study (Please check all that apply):

Check all that apply.

- [ ] My academic qualification details
- [ ] Direct quotes from my audio recordings
- [ ] My interaction logs

19. Your Name *

____

20. Any other questions or concerns?

____

____

____

____

____

Appendix D

Pre-test and Post-test Materials for Study 2

D.1 Pre-test

Name:	**Date:**
Year and department:	**College:**

Instructions:
1. Use the design diagrams of the **ATM system** in page 3,4 to answer the following questions
2. You can use additional sheets if space is not sufficient. Attach them along with these sheets

The design of an ATM system has been provided in pages 3,4. The requirements of the system are -

1. A user with a valid account can register his/her ATM 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 and inputs the correct PIN, the following options are shown
    a. Withdraw - The user can withdraw money from his/her account. If the balance is less than Rs.1000, withdrawal is denied
    b. Change PIN - The user can change his/her PIN by entering the previous PIN correctly

Q1. Consider the execution of the following incomplete scenario - “The user has a balance of Rs. 5000 in his account, enters the correct PIN and withdraws Rs.500”. What happens at the end of this scenario? Explain the sequence of steps and changes that occur in the system from the beginning to the end on execution of this scenario.

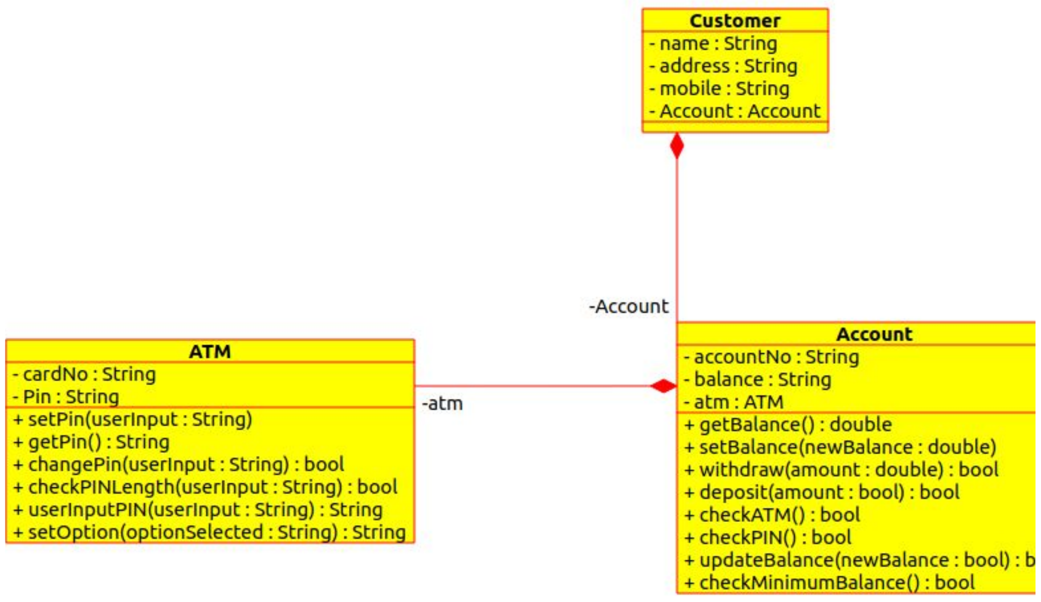
Q2. Consider the execution of the following incomplete scenario - “ A new user selects the option register and sets a PIN ”. What happens at the end of this scenario? Explain the sequence of steps and changes that occur in the system from the beginning to the end on execution of this scenario.

Q3. Identify defects(if any) in the following design diagrams based on the requirements. For each defect, provide a logical explanation of why you think it is a defect.

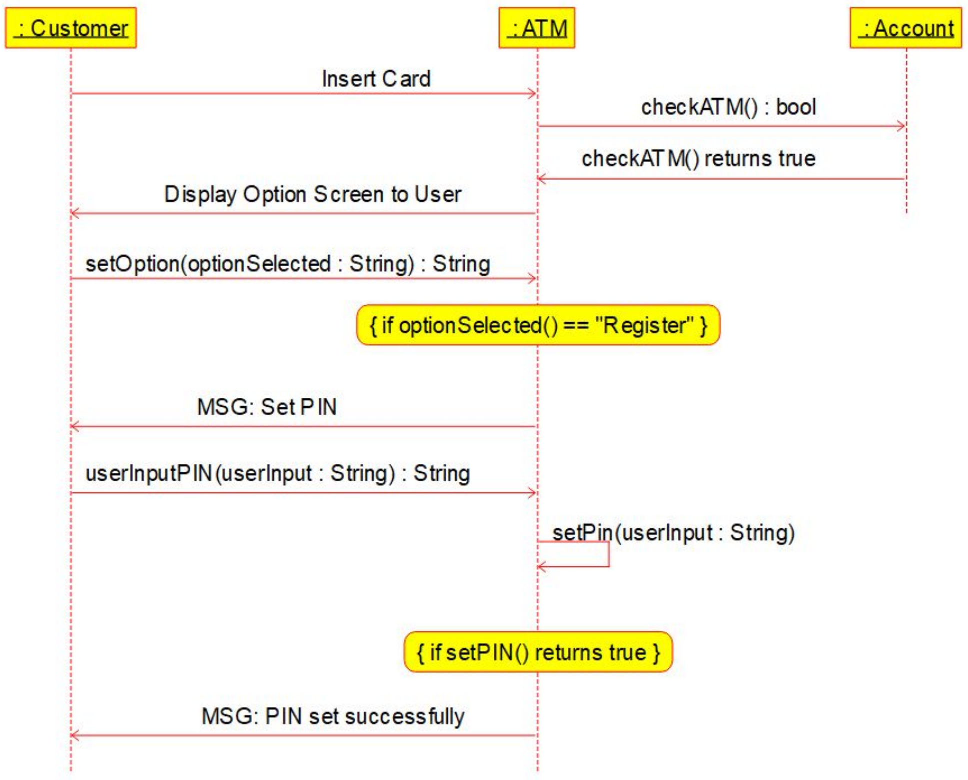
Q4. What according to you is the purpose of class diagrams in a software design?

Q5. What according to you is the purpose of sequence diagrams in a software design?

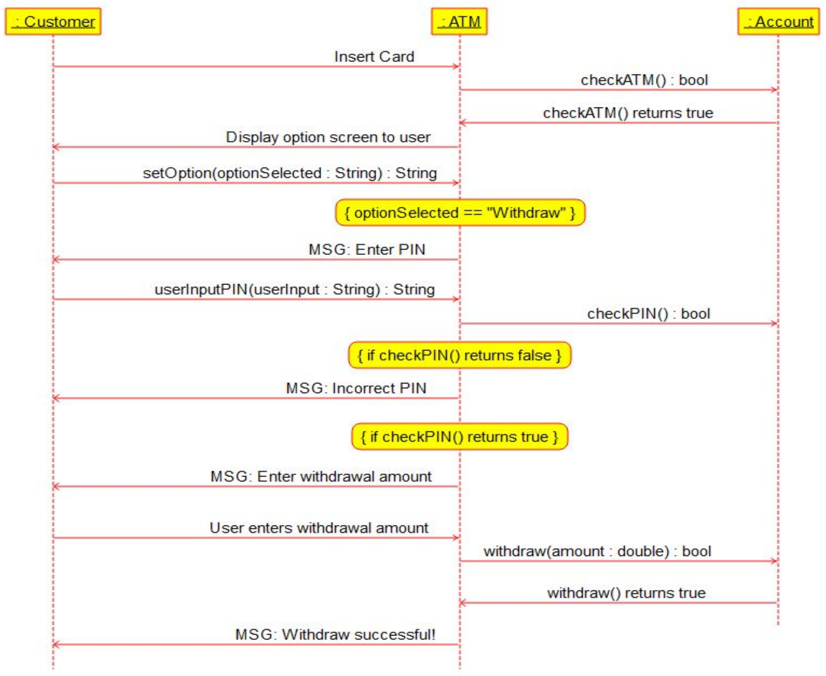
Class Diagram


Customer
- name : String
- address : String
- mobile : String
- Account : Account
-Account
ATM
- cardNo : String
- Pin : String
+ setPin(userInput : String)
+ getPin() : String
+ changePin(userInput : String) : bool
+ checkPINLength(userInput : String) : bool
+ userInputPIN(userInput : String) : String
+ setOption(optionSelected : String) : String
-atm
Account
- accountNo : String
- balance : String
- atm : ATM
+ getBalance() : double
+ setBalance(newBalance : double)
+ withdraw(amount : double) : bool
+ deposit(amount : bool) : bool
+ checkATM() : bool
+ checkPIN() : bool
+ updateBalance(newBalance : bool) : b
+ checkMinimumBalance() : bool

Register PIN Sequence Diagram

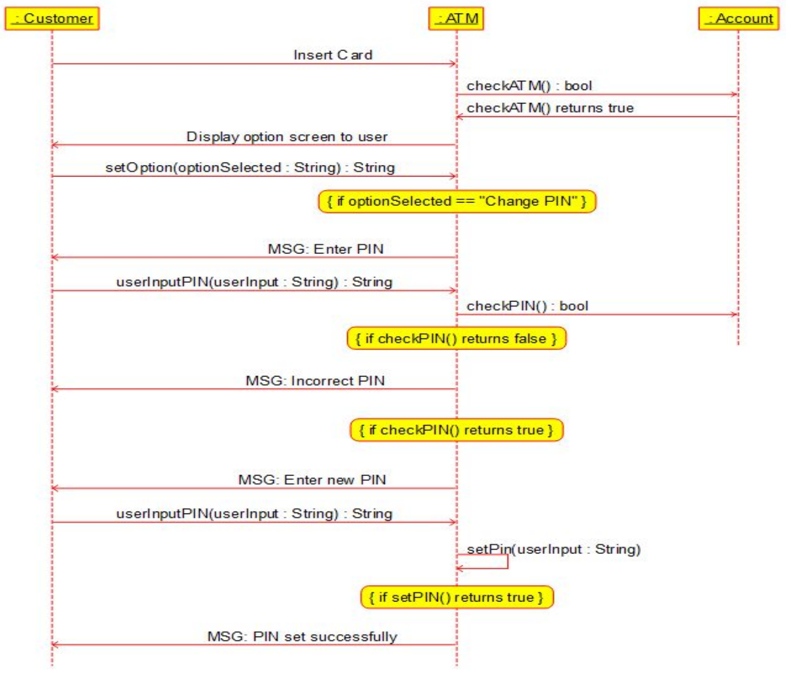

: Customer
: ATM
: Account
Insert Card
checkATM() : bool
checkATM() returns true
Display Option Screen to User
setOption(optionSelected : String) : String
{ if optionSelected() == "Register" }
MSG: Set PIN
userInputPIN(userInput : String) : String
setPin(userInput : String)
{ if setPIN() returns true }
MSG: PIN set successfully

Withdraw Sequence Diagram



: Customer
: ATM
: Account
Insert Card
checkATM() : bool
checkATM() returns true
Display option screen to user
setOption(optionSelected : String) : String
{ optionSelected == "Withdraw" }
MSG: Enter PIN
userInputPIN(userInput : String) : String
checkPIN() : bool
{ if checkPIN() returns false }
MSG: Incorrect PIN
{ if checkPIN() returns true }
MSG: Enter withdrawal amount
User enters withdrawal amount
withdraw(amount : double) : bool
withdraw() returns true
MSG: Withdraw successful!

Change PIN Sequence Diagram



: Customer
: ATM
: Account
Insert Card
checkATM() : bool
checkATM() returns true
Display option screen to user
setOption(optionSelected : String) : String
{ if optionSelected == "Change PIN" }
MSG: Enter PIN
userInputPIN(userInput : String) : String
checkPIN() : bool
{ if checkPIN() returns false }
MSG: Incorrect PIN
{ if checkPIN() returns true }
MSG: Enter new PIN
userInputPIN(userInput : String) : String
setPin(userInput : String)
{ if setPIN() returns true }
MSG: PIN set successfully

D.2 Post-test

Name: **Date:**

Year and department: **College:**

Instructions:
1. Use the design diagrams of the **online library system** in page 3,4 to answer the following questions
2. You can use additional sheets if space is not sufficient. Attach them along with these sheets

The design of an online library has been provided in another sheet to you. The requirements are as follows -

1. If the user has not registered, then they should register in order to obtain access to the library
2. Registered users can issue books if it is available.
3. Registered users can return issued books.
4. Registered users can issue upto 5 books.
5. A book can be issued to a single user only

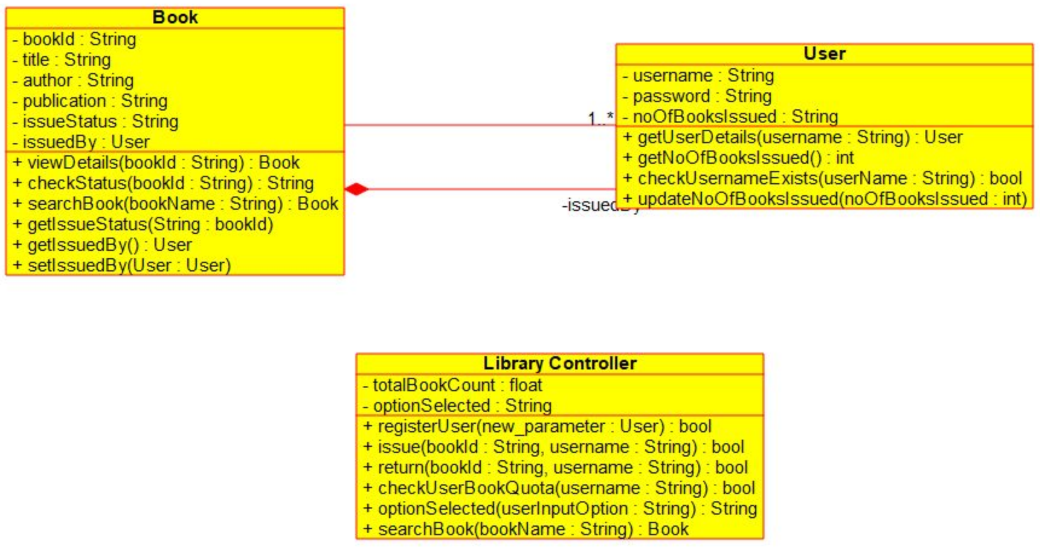
Q1. Using the design tracing technique, trace the following scenario -
“When a new user selects the register option, and enters the username and password, the user registers successfully.”

Q2. Using the design tracing technique, trace the following scenario -
“When the user selects the return option and enters the bookId, the user successfully returns the book”

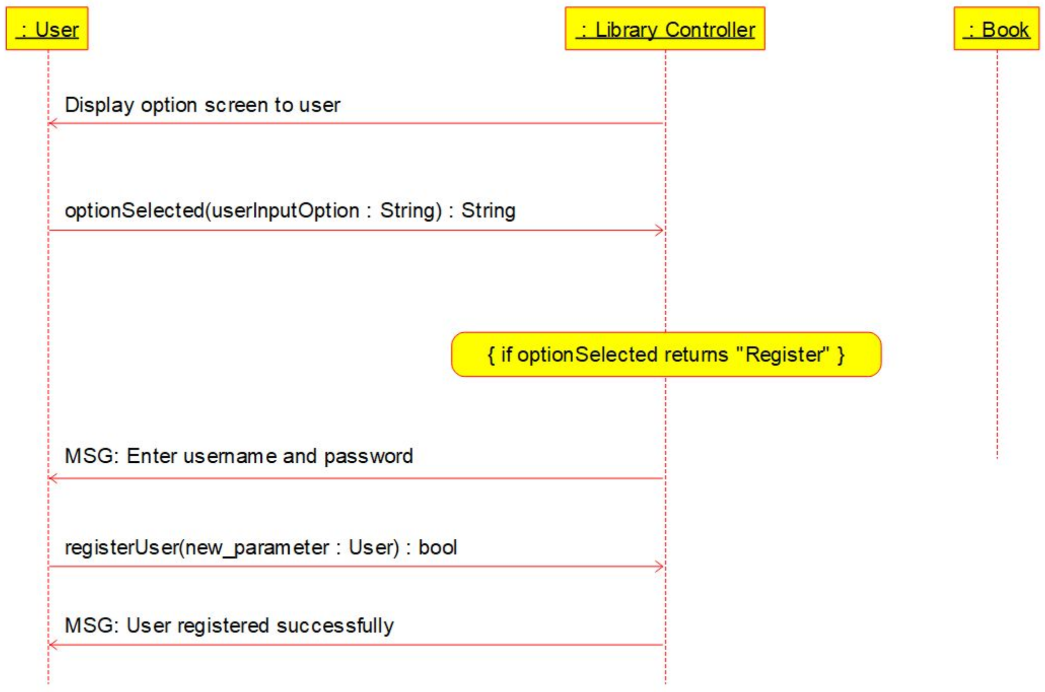
Q3. Identify defects(if any) in the following design diagrams based on the requirements. For each defect, provide a logical explanation of why you think it is a defect. You can use the design tracing technique which you learnt to identify defects.

Q4. After interacting with VeriSIM, do you think there is a change in your understanding **specifically of** the purpose of class diagrams and sequence diagrams in software design? If so, how?

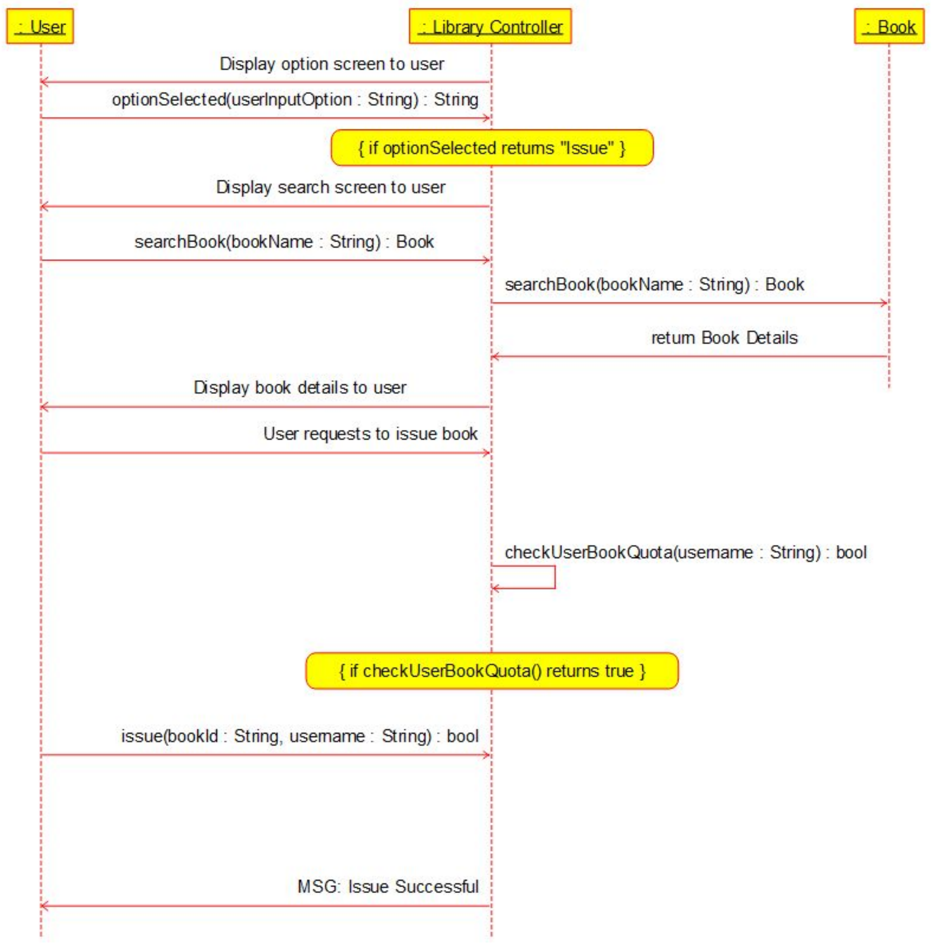
Class Diagram


Book
- bookId : String
- title : String
- author : String
- publication : String
- issueStatus : String
- issuedBy : User
+ viewDetails(bookId : String) : Book
+ checkStatus(bookId : String) : String
+ searchBook(bookName : String) : Book
+ getIssueStatus(String : bookId)
+ getIssuedBy() : User
+ setIssuedBy(User : User)
1..*
-issuedBy
User
- username : String
- password : String
- noOfBooksIssued : String
+ getUserDetails(username : String) : User
+ getNoOfBooksIssued() : int
+ checkUsernameExists(userName : String) : bool
+ updateNoOfBooksIssued(noOfBooksIssued : int)
Library Controller
- totalBookCount : float
- optionSelected : String
+ registerUser(new_parameter : User) : bool
+ issue(bookId : String, username : String) : bool
+ return(bookId : String, username : String) : bool
+ checkUserBookQuota(username : String) : bool
+ optionSelected(userInputOption : String) : String
+ searchBook(bookName : String) : Book

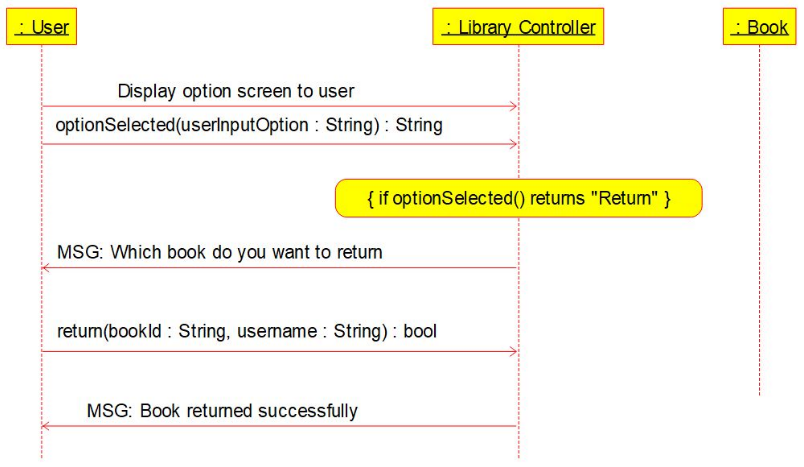
Register Sequence Diagram


: User
: Library Controller
: Book
Display option screen to user
optionSelected(userInputOption : String) : String
{ if optionSelected returns "Register" }
MSG: Enter username and password
registerUser(new_parameter : User) : bool
MSG: User registered successfully

Issue Sequence Diagram


: User
: Library Controller
: Book
Display option screen to user
optionSelected(userInputOption : String) : String
{ if optionSelected returns "Issue" }
Display search screen to user
searchBook(bookName : String) : Book
searchBook(bookName : String) : Book
return Book Details
Display book details to user
User requests to issue book
checkUserBookQuota(username : String) : bool
{ if checkUserBookQuota() returns true }
issue(bookId : String, username : String) : bool
MSG: Issue Successful

Return Sequence Diagram


: User
: Library Controller
: Book
Display option screen to user
optionSelected(userInputOption : String) : String
{ if optionSelected() returns "Return" }
MSG: Which book do you want to return
return(bookId : String, username : String) : bool
MSG: Book returned successfully

D.3 Feedback form after interacting with VeriSIM

VeriSIM Feedback Form

Dear Learner,
Thank You for your sustained participation during this workshop. We hope you had something concrete to take-away at the end of this workshop.

This survey is a mechanism through which you can communicate your feedback to the instructor about various aspects of the learning environment.

* Required

1. Email address *

____

2. Your Name *

Your personal details will be kept confidential

____

Feedback about usability of VeriSIM

In the upcoming sections we will be asking your feedback about VeriSIM. Do provide us your honest feedback and do not hesitate to point out shortcomings.

3. 2.1. I think that I would like to use VeriSIM frequently *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

4. 2.2. I found VeriSIM unnecessarily complex *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

5. 2.3. I thought VeriSIM was easy to use *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

6. 2.4. I think that I would need the support of a technical person to be able to use VeriSIM *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

7. 2.5. I found the various functions in VeriSIM were well integrated *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

8. 2.6. I thought there was too much inconsistency in VeriSIM *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	⬭	⬭	⬭	⬭	⬭	Strongly Agree

9. 2.7. I would imagine that most people would learn to use VeriSIM very quickly *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	⬭	⬭	⬭	⬭	⬭	Strongly Agree

10. 2.8. I found VeriSIM very cumbersome to use *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	⬭	⬭	⬭	⬭	⬭	Strongly Agree

11. 2.9. I felt very confident using VeriSIM *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	⬭	⬭	⬭	⬭	⬭	Strongly Agree

12. 2.10. I needed to learn a lot of things before I could get going with VeriSIM *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

Now that you have interacted with VeriSIM, rate your confidence regarding your knowledge of the following

1 indicates "Not confident at all", 5 indicates "Absolutely confident". If a specific term or task is totally unfamiliar to you, please mark 1

13. Identify relevant data variables from the the class diagram to trace a scenario *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

14. Identify relevant events from sequence diagrams to trace a scenario *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

15. Visualize the change of values of variables while tracing a scenario *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

16. Understand the purpose of using class diagrams in software design *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

17. Understand the purpose of using sequence diagrams in software design *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

18. Identify defects in a set of software design diagrams created by someone else *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

19. Identify defects in a set of software design diagrams that I have created myself *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

20. Create software design diagrams(Eg: UML) for a given set of requirements *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

21. Debug (correct all the errors) a long and complex program that I had written, and make it work *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

22. Comprehend (Understand) a long, complex multifile program in my favorite programming language *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

23. Mentally trace through the execution of a long, complex, multifile program given to me. *

Mark only one oval.

	1	2	3	4	5	
Not confident at all	◯	◯	◯	◯	◯	Absolutely confident

One last section! This information will help us improve the workshop for future participants. Thank you so much for providing this feedback.

24. How many stages of VeriSIM did you complete? *

Mark only one oval.

- ◯ I completed all the stages - Problem Understanding, Design Tracing and Reflection stages
- ◯ I completed the Problem Understanding Stage and some challenges in the Design Tracing stage
- ◯ I completed the Problem Understanding Stage but I could not complete any of the challenges in the Design Tracing stage
- ◯ I could not complete the Problem Understanding stage
- ◯ Other: ____________

25. What are the main things you learned from the workshop? *

26. What features of VeriSIM did you find most useful ? *

27. What features of VeriSIM did you find challenging/ frustrating ? *

28. Any suggestions to improve when we design the next online workshop for you. *

Appendix E

Scenario Branching Worksheet

Name: **Date:**

Year and department: **College:**

Instructions:
We will be using the ATM system example in this session. The requirements of the system are -

1. A user with a valid account can register his/her ATM 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 and inputs the correct PIN, the following options are shown
   a. Withdraw - The user can withdraw money from his/her account. If the balance is less than Rs.1000, withdrawal is denied
   b. Change PIN - The user can change his/her PIN by entering the previous PIN correctly

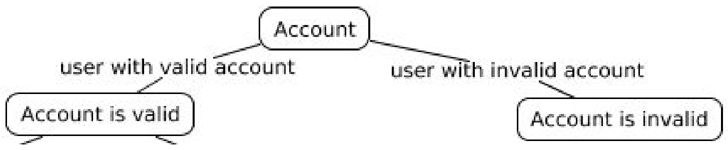
Example: Requirement 1:

A user with a valid account can register his/her ATM 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.

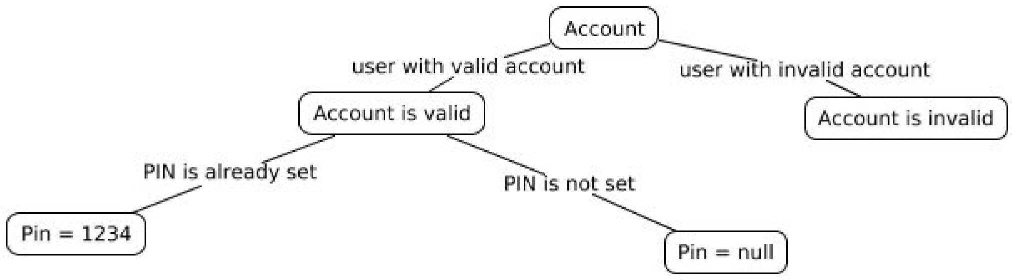
Step 1: Identifying sub-goals in the requirement
Identify the sub-goals in the requirement. For example, in requirement 1, the sub-goals are:

a. User with valid account
b. Sets a PIN if a PIN hasn't been set yet
c. PIN should be of length 4 and should contain only numbers

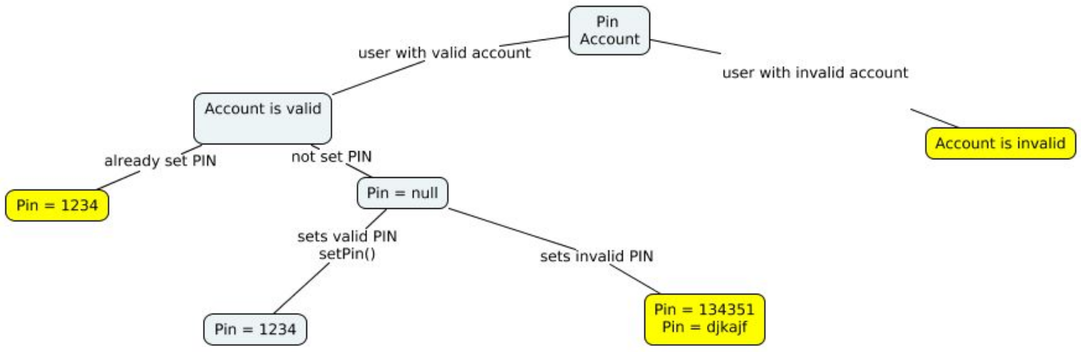
Step 2: For each sub-goal, identify relevant variables. Identify different possibilities of these variables. Use the concept map to come up with alternate scenarios
Consider the example for requirement 1
a. User with valid account - The account object can be either set or not set. Hence, two scenarios are possible, either the user has a valid account or an invalid account. The linking phrases indicate the different scenarios. The values inside the node indicate objects and data members from the class diagram



b. Sets a PIN - Based on the value of Pin, two scenarios are possible, either the PIN is already set or it is not set



c. Finally, the entered Pin can be valid or invalid - setPin() is called to set the correct Pin.



In the above scenario tree, start from the root node and traverse all the way down. Each path corresponds to a scenario.
Scenario 1: User with a valid account has already set a Pin
Scenario 2: User with a valid account has not set a Pin and sets a valid Pin
Scenario 3: User with a valid account has not set a Pin and sets an invalid Pin
Scenario 4: User has an invalid account

Which of the following scenarios are not described in the design diagrams? - Scenario 1, Scenario 3 and Scenario 4
Hence, these are defects which need to be rectified in the diagrams

Activity: Requirement 2a and 2b:

Come up with the scenario tree for requirement 2a and 2b. Use the steps similar to the ones done for requirement 1.
You can use the following form to record your observations - **https://bit.ly/2sG1QxT**

Appendix F

Pre-test and Post-test Materials for Study 3

F.1 Pre-test

Name: **Date:**

Year and department: **College:**

Instructions:
1. Use the design diagrams of the **ATM system** in page 3,4 to answer the following questions
2. You can use additional sheets if space is not sufficient. Attach them along with these sheets

The design of an ATM system has been provided in pages 3,4. The requirements of the system are -

1. A user with a valid account can register his/her ATM 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 and inputs the correct PIN, the following options are shown
    a. Withdraw - The user can withdraw money from his/her account. If the balance is less than Rs.1000, withdrawal is denied
    b. Change PIN - The user can change his/her PIN by entering the previous PIN correctly

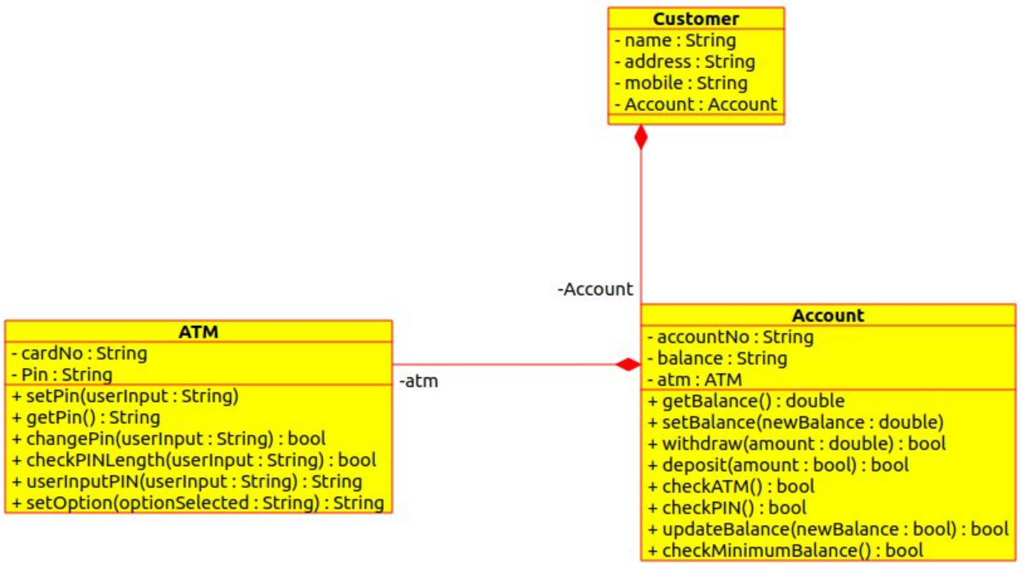
Q1. Consider the execution of the following incomplete scenario - “The user has a balance of Rs. 5000 in his account, enters the correct PIN and withdraws Rs.500”. What happens at the end of this scenario? Explain the sequence of steps and changes that occur in all the design diagrams from the beginning to the end on execution of this scenario.

Q2. Consider the execution of the following incomplete scenario - “ A new user selects the option register and sets a PIN ”. What happens at the end of this scenario? Explain the sequence of steps and changes that occur in all the design diagrams from the beginning to the end on execution of this scenario.

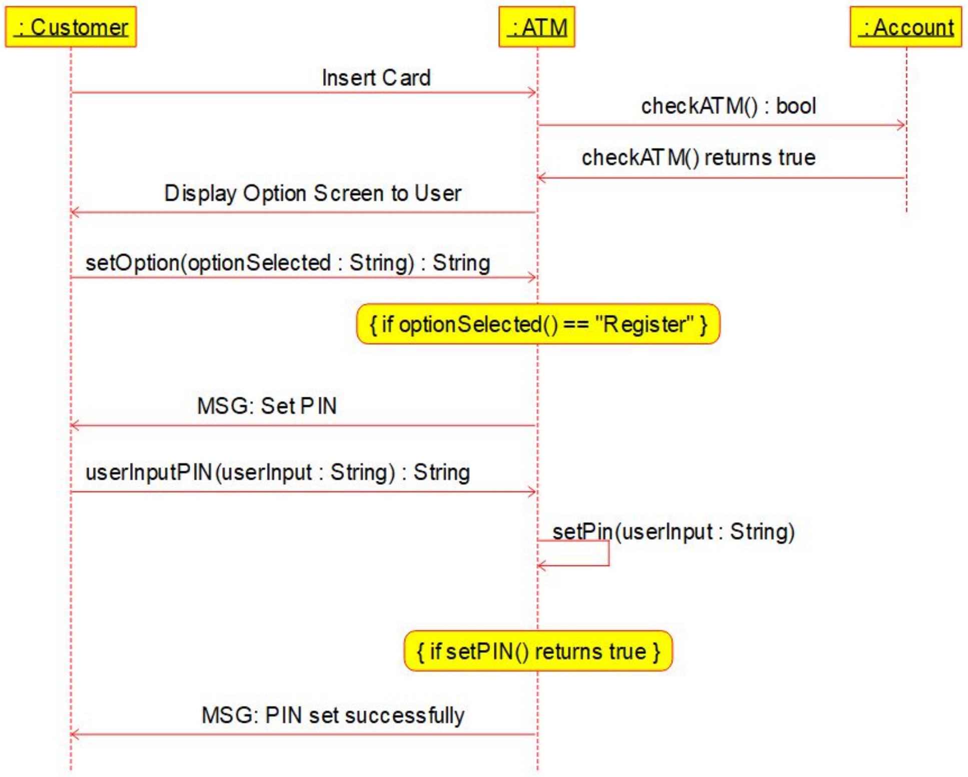
Q3. For each requirement, list all possible scenarios based on the design. Examples of scenarios are given in Q1 and Q2.

Q4. Identify defects(if any) in the following design diagrams based on the requirements. For each defect, provide a logical explanation of why you think it is a defect.

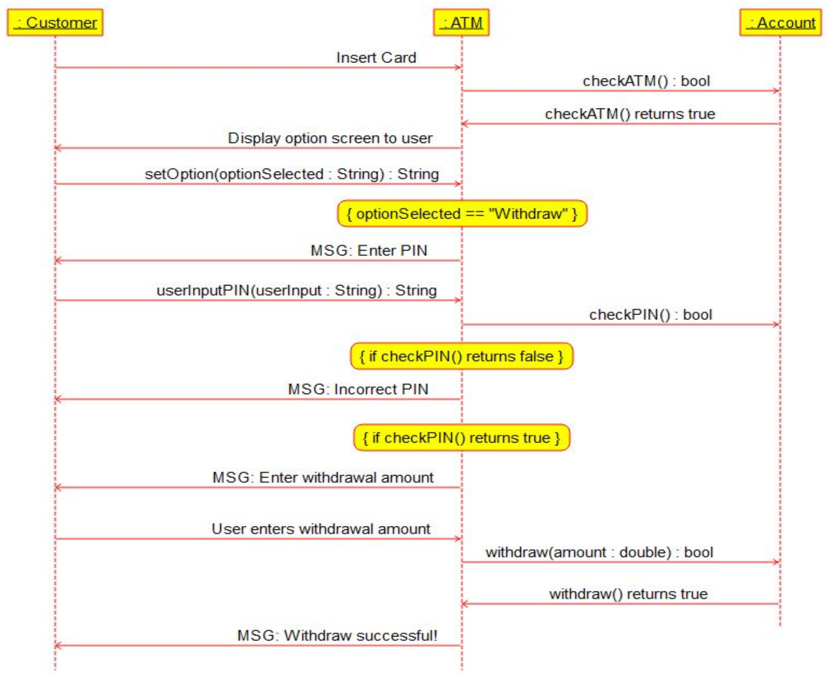
Class Diagram


Customer
- name : String
- address : String
- mobile : String
- Account : Account
-Account
ATM
- cardNo : String
- Pin : String
+ setPin(userInput : String)
+ getPin() : String
+ changePin(userInput : String) : bool
+ checkPINLength(userInput : String) : bool
+ userInputPIN(userInput : String) : String
+ setOption(optionSelected : String) : String
-atm
Account
- accountNo : String
- balance : String
- atm : ATM
+ getBalance() : double
+ setBalance(newBalance : double)
+ withdraw(amount : double) : bool
+ deposit(amount : bool) : bool
+ checkATM() : bool
+ checkPIN() : bool
+ updateBalance(newBalance : bool) : bool
+ checkMinimumBalance() : bool

Register PIN Sequence Diagram

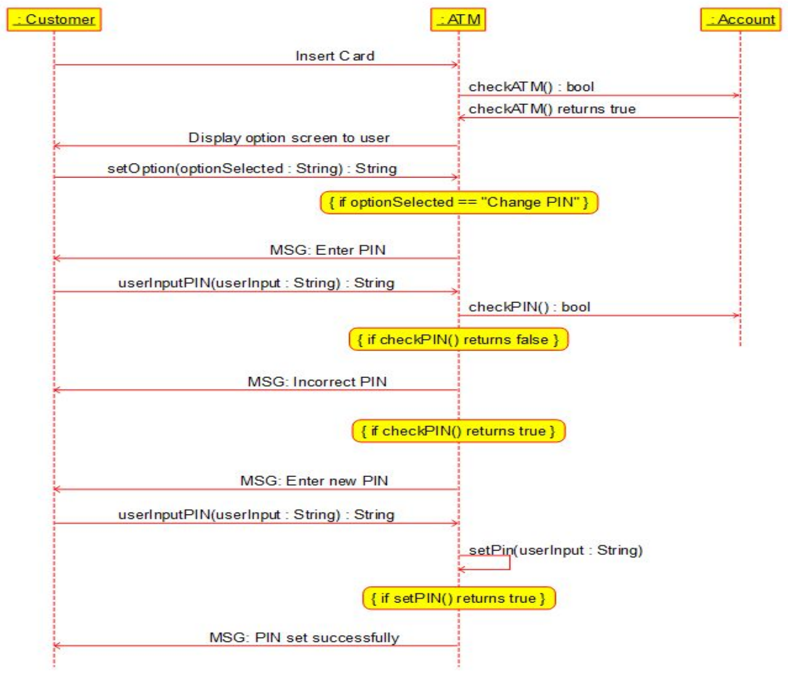

: Customer
: ATM
: Account
Insert Card
checkATM() : bool
checkATM() returns true
Display Option Screen to User
setOption(optionSelected : String) : String
{ if optionSelected() == "Register" }
MSG: Set PIN
userInputPIN(userInput : String) : String
setPin(userInput : String)
{ if setPIN() returns true }
MSG: PIN set successfully

Withdraw Sequence Diagram



: Customer
: ATM
: Account
Insert Card
checkATM() : bool
checkATM() returns true
Display option screen to user
setOption(optionSelected : String) : String
{ optionSelected == "Withdraw" }
MSG: Enter PIN
userInputPIN(userInput : String) : String
checkPIN() : bool
{ if checkPIN() returns false }
MSG: Incorrect PIN
{ if checkPIN() returns true }
MSG: Enter withdrawal amount
User enters withdrawal amount
withdraw(amount : double) : bool
withdraw() returns true
MSG: Withdraw successful!

Change PIN Sequence Diagram



: Customer
: ATM
: Account
Insert Card
checkATM() : bool
checkATM() returns true
Display option screen to user
setOption(optionSelected : String) : String
{ if optionSelected == "Change PIN" }
MSG: Enter PIN
userInputPIN(userInput : String) : String
checkPIN() : bool
{ if checkPIN() returns false }
MSG: Incorrect PIN
{ if checkPIN() returns true }
MSG: Enter new PIN
userInputPIN(userInput : String) : String
setPin(userInput : String)
{ if setPIN() returns true }
MSG: PIN set successfully

F.2 Post-test

Name: **Date:**

Year and department: **College:**

Instructions:
1. Use the design diagrams on page 2,3 to answer the following questions

Indiaflix is a streaming service that offers a wide variety of regional movies and TV shows – on thousands of internet-connected devices. You are given the responsibility to create a user interface for Indiaflix. The requirements are as follows -

1. There are two plans - Basic and Premium. If a person is not registered yet, they can choose any of the plans during signup.
2. The basic account is free. In the basic account, users have access only to a limited number of movies and TV shows. Each content is assigned a tag - basic or premium. Users of the basic account can only watch upto 5 hours of content every week. Basic users can login only in a single screen at a time.
3. For the premium account, users have to pay an annual subscription of Rs. 999/-. Premium account holders have access to all content in the platform, and can watch unlimited hours of content every week. Premium users can login in to upto 4 screens at a time.

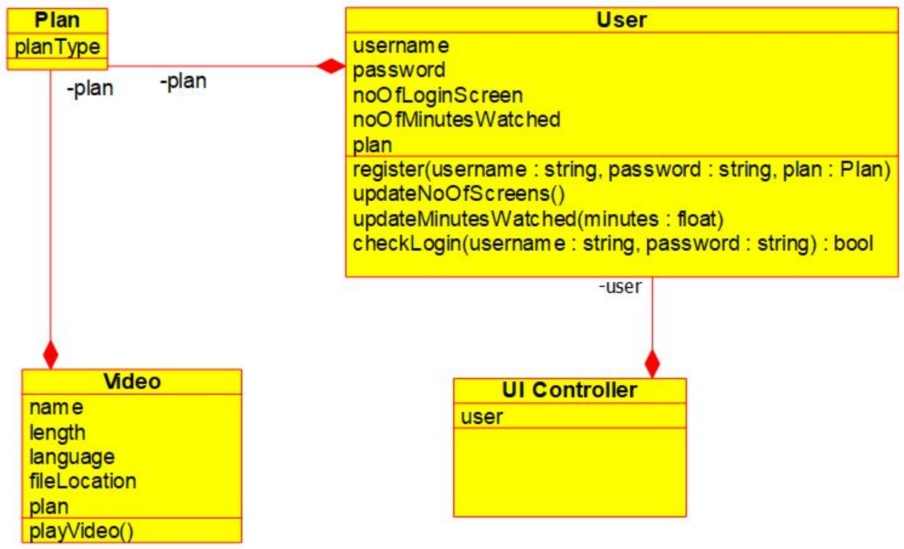
Q1. Consider the execution of the following scenario - "When a new user selects the register option, chooses the basic plan and enters the username and password, the user registers successfully."
What happens at the end of this scenario? Explain the sequence of steps and changes that occur in all the design diagrams from the beginning to the end on execution of this scenario.

Q2. Consider the execution of the following scenario - "When a premium user logs in to IndiaFlix, and selects a content to watch, the system allows them to watch the content"
What happens at the end of this scenario? Explain the sequence of steps and changes that occur in all the design diagrams from the beginning to the end on execution of this scenario.

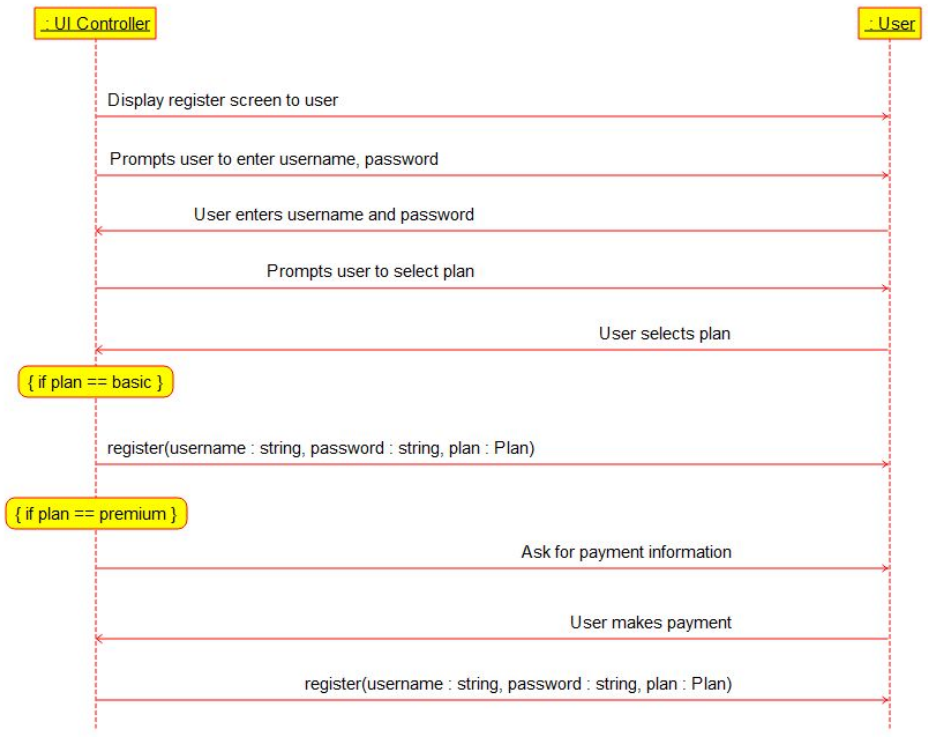
Q3. For each requirement, list all possible scenarios based on the design. Examples of scenarios are given in Q1 and Q2.

Q4. Identify defects(if any) in the following design diagrams based on the requirements. For each defect, provide a logical explanation of why you think it is a defect.

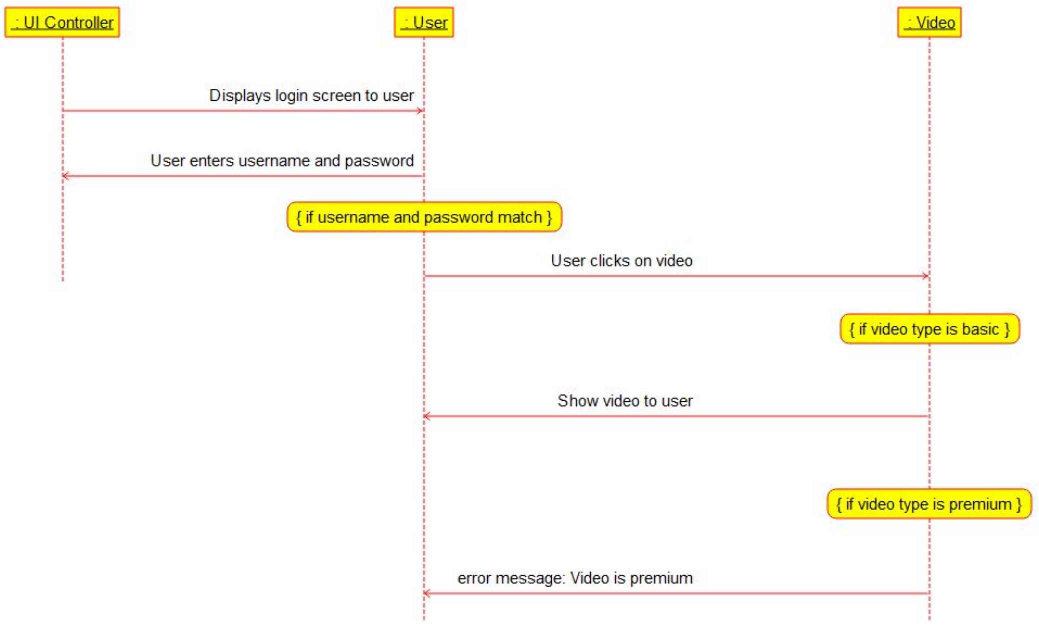
Class Diagram



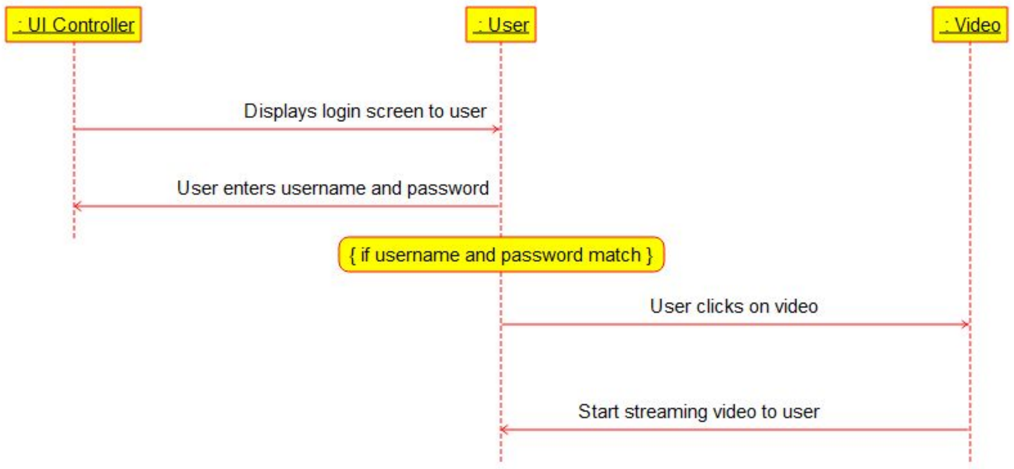
Register Sequence Diagram


: UI Controller
: User
Display register screen to user
Prompts user to enter username, password
User enters username and password
Prompts user to select plan
User selects plan
{ if plan == basic }
register(username : string, password : string, plan : Plan)
{ if plan == premium }
Ask for payment information
User makes payment
register(username : string, password : string, plan : Plan)

Watch Basic Content Sequence Diagram


: UI Controller
: User
: Video
Displays login screen to user
User enters username and password
{ if username and password match }
User clicks on video
{ if video type is basic }
Show video to user
{ if video type is premium }
error message: Video is premium

Watch Premium Content Sequence Diagram


: UI Controller
: User
: Video
Displays login screen to user
User enters username and password
{ if username and password match }
User clicks on video
Start streaming video to user

References

ACM, 2014. Software engineering 2014 curriculum guidelines for undergraduate degree programs in software engineeringa volume of the computing curricula series. URL `https://www.acm.org/binaries/content/assets/education/se2014.pdf`

Adams, W. K., Reid, S., LeMaster, R., McKagan, S. B., Perkins, K. K., Dubson, M., Wieman, C. E., 2008. A study of educational simulations part i-engagement and learning. Journal of Interactive Learning Research 19 (3), 397–419.

Adelson, B., Soloway, E., 1985. The role of domain expenence in software design. IEEE Transactions on Software Engineering (11), 1351–1360.

Adelson, B., Soloway, E., 1986. A model of software design. International Journal of Intelligent Systems 1 (3), 195–213.

Ahmed, S., Wallace, K. M., Blessing, L. T., 2003. Understanding the differences between how novice and experienced designers approach design tasks. Research in engineering design 14 (1), 1–11.

Arisholm, E., Briand, L. C., Hove, S. E., Labiche, Y., 2006. The impact of uml documentation on software maintenance: An experimental evaluation. IEEE Transactions on Software Engineering 32 (6), 365–381.

Azevedo, R., Cromley, J. G., Moos, D. C., Greene, J. A., Winters, F. I., 2011. Adaptive content and process scaffolding: A key to facilitating students' self-regulated learning with hypermedia. Psychological Test and Assessment Modeling 53 (1), 106.

Barab, S., Squire, K., 2004. Design-based research: Putting a stake in the ground. The journal of the learning sciences 13 (1), 1–14.

Baxter, L. A., 1991. Content analysis. Studying interpersonal interaction 239, 254.

Begel, A., Simon, B., 2008a. Novice software developers, all over again. In: Proceedings of the Fourth international Workshop on Computing Education Research. ACM, pp. 3–14.

Begel, A., Simon, B., 2008b. Struggles of new college graduates in their first software development job. In: ACM SIGCSE Bulletin. Vol. 40. ACM, pp. 226–230.

Bolloju, N., Leung, F. S., 2006. Assisting novice analysts in developing quality conceptual models with uml. Communications of the ACM 49 (7), 108–112.

Braun, V., Clarke, V., 2012. Thematic analysis.

Brechner, E., 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, pp. 134–136.

Buckley, B. C., Gobert, J. D., Horwitz, P., O'Dwyer, L. M., 2010. Looking inside the black box: assessing model-based learning and inquiry in biologica™. International Journal of Learning Technology 5 (2), 166–190.

Budgen, D., Burn, A. J., Brereton, O. P., Kitchenham, B. A., Pretorius, R., 2011. Empirical evidence about the uml: a systematic literature review. Software: Practice and Experience 41 (4), 363–392.

Burgueño, L., Vallecillo, A., Gogolla, M., 2018. Teaching uml and ocl models and their validation to software engineering students: an experience report. Computer Science Education 28 (1), 23–41.

Burkhardt, J.-M., Détienne, F., Wiedenbeck, S., 2002. Object-oriented program comprehension: Effect of expertise, task and phase. Empirical Software Engineering 7 (2), 115–156.

Chaudron, M. R., Heijstek, W., Nugroho, A., 2012. How effective is uml modeling? Software & Systems Modeling 11 (4), 571–580.

Chren, S., Buhnova, B., Macak, M., Daubner, L., Rossi, B., 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, pp. 100–109.

Ciccozzi, F., Famelis, M., Kappel, G., Lambers, L., Mosser, S., Paige, R. F., Pierantonio, A., Rensink, A., Salay, R., Taentzer, G., et al., 2018. How do we teach modelling and model-driven engineering? a survey. In: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. pp. 122–129.

Clement, J., 2000. Model based learning as a key research area for science education. International Journal of Science Education 22 (9), 1041–1053.

Cobb, P., Confrey, J., DiSessa, A., Lehrer, R., Schauble, L., 2003. Design experiments in educational research. Educational researcher 32 (1), 9–13.

Cohane, R., November 2017. Financial cost of software bugs.
URL `https://medium.com/@ryancohane/financial-cost-of-software-bugs-51k`

Cohen, J., 1960. A coefficient of agreement for nominal scales. Educational and psychological measurement 20 (1), 37–46.

Cohen, L., Manion, L., Morrison, K., 2017. Research methods in education. routledge.

Cunningham, K., Blanchard, S., Ericson, B., Guzdial, M., 2017. Using tracing and sketching to solve programming problems: Replicating and extending an analysis of what students draw. In: Proceedings of the 2017 ACM Conference on International Computing Education Research. ACM, pp. 164–172.

Curtis, B., Krasner, H., Iscoe, N., 1988. A field study of the software design process for large systems. Communications of the ACM 31 (11), 1268–1287.

Dagenais, B., Ossher, H., Bellamy, R. K., Robillard, M. P., De Vries, J. P., 2010. Moving into a new software project landscape. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. ACM, pp. 275–284.

Davies, J., Nersessian, N. J., Goel, A. K., 2005. Visual models in analogical problem solving. Foundations of Science 10 (1), 133–152.

Derry, S. J., Pea, R. D., Barron, B., Engle, R. A., Erickson, F., Goldman, R., Hall, R., Koschmann, T., Lemke, J. L., Sherin, M. G., et al., 2010. Conducting video research in the learning sciences: Guidance on selection, analysis, technology, and ethics. The Journal of the Learning Sciences 19 (1), 3–53.

Détienne, F., 2001. Software Design–Cognitive Aspect. Springer Science & Business Media.

Dobing, B., Parsons, J., 2006. How uml is used. Communications of the ACM 49 (5), 109–113.

Dzidek, W. J., Arisholm, E., Briand, L. C., 2008. A realistic empirical evaluation of the costs and benefits of uml in software maintenance. IEEE Transactions on software engineering 34 (3), 407–432.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Zander, C., 2006. Can graduating students design software systems? ACM SIGCSE Bulletin 38 (1), 403–407.

Fernández-Sáez, A. M., Genero, M., Caivano, D., Chaudron, M. R., 2016. Does the level of detail of uml diagrams affect the maintainability of source code?: a family of experiments. Empirical Software Engineering 21 (1), 212–259.

Fitzgerald, S., Simon, B., Thomas, L., 2005. Strategies that students use to trace code: an analysis based in grounded theory. In: Proceedings of the first international workshop on Computing education research. ACM, pp. 69–80.

Fretz, E. B., Wu, H.-K., Zhang, B., Davis, E. A., Krajcik, J. S., Soloway, E., 2002. An investigation of software scaffolds supporting modeling practices. Research in Science Education 32 (4), 567–589.

Ge, X., Land, S., 2004. A conceptual framework for scaffolding ill-structured problem-solving processes using question prompts and peer interactions. Educational Research Technology and Development 52 (2), 1042–1629.

Genero, M., Fernández-Saez, A. M., Nelson, H. J., Poels, G., Piattini, M., 2011. Research review: a systematic literature review on the quality of uml models. Journal of Database Management (JDM) 22 (3), 46–70.

Gentner, D., Gentner, D. R., 1983. Flowing waters or teeming crowds: Mental models of electricity. Mental models 99, 129.

Glezer, C., Last, M., Nachmany, E., Shoval, P., 2005. Quality and comprehension of uml interaction diagrams-an experimental comparison. Information and Software Technology 47 (10), 675–692.

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

Haskins, B., Jonette, S., Dick, B., Moroney, G., Lovell, R., Dabney, J., 2004. Error cost escalation through the project life cycle. In: Proceedings of the 14th Annual INCOSE International Symposium.

Hestenes, D., 1987. Toward a modeling theory of physics instruction. American journal of physics 55 (5), 440–454.

Hestenes, D., 1992. Modeling games in the newtonian world. American Journal of Physics 60 (8), 732–748.

Hestenes, D., 2010. Modeling theory for math and science education. In: Modeling students' mathematical modeling competencies. Springer, pp. 13–41.

Hungerford, B. C., Hevner, A. R., Collins, R. W., 2004. Reviewing software diagrams: A cognitive study. IEEE Transactions on Software Engineering 30 (2), 82–96.

Johnson-Laird, P. N., 1983. Mental models: Towards a cognitive science of language, inference, and consciousness. No. 6. Harvard University Press.

Kim, J., Hahn, J., Hahn, H., 2000. How do we understand a system with (so) many diagrams? cognitive integration processes in diagrammatic reasoning. Information Systems Research 11 (3), 284–303.

Kopainsky, B., Alessi, S. M., 2015. Effects of structural transparency in system dynamics simulators on performance and understanding. Systems 3 (4), 152–176.

Kopainsky, B., Alessi, S. M., Pedercini, M., Davidsen, P. I., 2015. Effect of prior exploration as an instructional strategy for system dynamics. Simulation & Gaming 46 (3-4), 293–321.

Lange, C. F., Chaudron, M. R., 2006. Effects of defects in uml models: an experimental investigation. In: Proceedings of the 28th international conference on Software engineering. pp. 401–411.

Lange, C. F., Chaudron, M. R., Muskens, J., 2006. In practice: Uml software architecture and design description. IEEE software 23 (2), 40–46.

LaToza, T. D., Myers, B. A., 2010. Developers ask reachability questions. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1. pp. 185–194.

Letovsky, S., 1987. Cognitive processes in program comprehension. Journal of Systems and software 7 (4), 325–339.

Li, P. L., 2016. What makes a great software engineer. Ph.D. thesis.

Lin, X., Lehman, J. D., 1999. Supporting learning of variable control in a computer-based biology environment: Effects of prompting college students to reflect on their own thinking. Journal of Research in Science Teaching: The Official Journal of the National Association for Research in Science Teaching 36 (7), 837–858.

Lindland, O. I., Sindre, G., Solvberg, A., 1994. Understanding quality in conceptual modeling. IEEE software 11 (2), 42–49.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., et al., 2004. A multi-national study of reading and tracing skills in novice programmers. In: ACM SIGCSE Bulletin. Vol. 36. ACM, pp. 119–150.

Liu, Z., Schonwetter, D. J., 2004. Teaching creativity in engineering. International Journal of Engineering Education 20 (5), 801–808.

Loftus, C., Thomas, L., Zander, C., 2011. Can graduating students design: revisited. In: Proceedings of the 42nd ACM technical symposium on Computer science education. ACM, pp. 105–110.

Louca, L. T., Zacharia, Z. C., 2015. Examining learning through modeling in k-6 science education. Journal of Science Education and Technology 24 (2-3), 192–215.

Mc Neill, T., Gero, J. S., Warren, J., 1998. Understanding conceptual electronic design using protocol analysis. Research in Engineering Design 10 (3), 129–140.

Milrad, M., Spector, J. M., Davidsen, P. I., et al., 2003. Model facilitated learning. Learning and teaching with technology: Principles and practices, 13–27.

Montazemi, A. R., Conrath, D. W., 1986. The use of cognitive mapping for information requirements analysis. MIS quarterly, 45–56.

Moreno, R., 2004. Decreasing cognitive load for novice students: Effects of explanatory versus corrective feedback in discovery-based multimedia. Instructional science 32 (1-2), 99–113.

Moreno, R., Flowerday, T., 2006. Students' choice of animated pedagogical agents in science learning: A test of the similarity-attraction hypothesis on gender and ethnicity. Contemporary educational psychology 31 (2), 186–207.

Moreno, R., Mayer, R. E., Spires, H. A., Lester, J. C., 2001. The case for social agency in computer-based teaching: Do students learn more deeply when they interact with animated pedagogical agents? Cognition and instruction 19 (2), 177–213.

Mulder, Y. G., Bollen, L., de Jong, T., Lazonder, A. W., 2016. Scaffolding learning by modelling: The effects of partially worked-out models. Journal of research in science teaching 53 (3), 502–523.

Mulder, Y. G., Lazonder, A. W., de Jong, T., 2011. Comparing two types of model progression in an inquiry learning environment with modelling facilities. Learning and Instruction 21 (5), 614–624.

Nelson, G. L., Xie, B., Ko, A. J., 2017. Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in cs1. In: Proceedings of the 2017 ACM Conference on International Computing Education Research. pp. 2–11.

Nelson, M., Piattini, M., 2012. A systematic literature review on the quality of uml models. Innovations in Database Design, Web Applications, and Information Systems Management, 310–334.

Newman, M., 2002. Software errors cost us economy 59.5 billion annually. NIST Assesses Technical Needs of Industry to Improve Software-Testing.

Norman, D. A., 2014. Some observations on mental models. In: Mental models. Psychology Press, pp. 15–22.

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

Otero, M. C., Dolado, J. J., 2002. An initial experimental assessment of the dynamic modelling in uml. Empirical Software Engineering 7 (1), 27–47.

Otero, M. C., Dolado, J. J., 2004. Evaluation of the comprehension of the dynamic modeling in uml. Information and Software Technology 46 (1), 35–53.

Papaevripidou, M., Zacharia, Z. C., 2015. Examining how students' knowledge of the subject domain affects their process of modeling in a computer programming environment. Journal of Computers in Education 2 (3), 251–282.

Pennington, N., 1987. Comprehension strategies in programming. In: Empirical studies of programmers: second workshop. Ablex Publishing Corp., pp. 100–113.

Petre, M., 2009. Insights from expert software design practice. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. pp. 233–242.

Petre, M., 2013. Uml in practice. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 722–731.

Petre, M., 2014. "no shit" or "oh, shit!": responses to observations on the use of uml in professional practice. Software & Systems Modeling 13 (4), 1225–1235.

Plomp, T., 2013. Educational design research: An introduction. Educational design research, 11–50.

Quintana, C., Reiser, B. J., Davis, E. A., Krajcik, J., Fretz, E., Duncan, R. G., Kyza, E., Edelson, D., Soloway, E., 2004. A scaffolding design framework for software to support science inquiry. The journal of the learning sciences 13 (3), 337–386.

Rapp, D. N., 2005. Mental models: Theoretical issues for visualizations in science education. In: Visualization in science education. Springer, pp. 43–60.

Roehm, T., Tiarks, R., Koschke, R., Maalej, W., 2012. How do professional developers comprehend software? In: Proceedings of the 34th International Conference on Software Engineering. IEEE Press, pp. 255–265.

Rumbaugh, J., Jacobson, I., Booch, G., 1999. The unified modeling language. Reference manual.

Rumbaugh, J., Jacobson, I., Booch, G., 2004. Unified modeling language reference manual, the. Pearson Higher Education.

Sahami, M., Danyluk, A., Fincher, S., Fisher, K., Grossman, D., Hawthorne, E., Katz, R., LeBlanc, R., Reed, D., Roach, S., et al., 2013. Computer science curricula 2013: Curriculum guidelines for undergraduate degree programs in computer science. Association for Computing Machinery (ACM)-IEEE Computer Society.

Sandoval, W. A., Reiser, B. J., 2004. Explanation-driven inquiry: Integrating conceptual and epistemic scaffolds for scientific inquiry. Science education 88 (3), 345–372.

Scanniello, G., Gravino, C., Tortora, G., 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. pp. 1–6.

Schön, D. A., 1987. Teaching artistry through reflection-in-action: Educating the reflective practitioner.

Schulte, C., 2008. Block model: an educational model of program comprehension as a tool for a scholarly approach to teaching. In: Proceedings of the Fourth international Workshop on Computing Education Research. pp. 149–160.

Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., Paterson, J. H., 2010. An introduction to program comprehension for computer science educators. In: Proceedings of the 2010 ITiCSE working group reports. ACM, pp. 65–86.

Schumacher, R. M., Czerwinski, M. P., 1992. Mental models and the acquisition of expert knowledge. In: The psychology of expertise. Springer, pp. 61–79.

Schwarz, C. V., Reiser, B. J., Davis, E. A., Kenyon, L., Achér, A., Fortus, D., Shwartz, Y., Hug, B., Krajcik, J., 2009. Developing a learning progression for scientific modeling: Making scientific modeling accessible and meaningful for learners. Journal of research in science teaching 46 (6), 632–654.

Schwarz, C. V., White, B. Y., 2005. Metamodeling knowledge: Developing students' understanding of scientific modeling. Cognition and instruction 23 (2), 165–205.

Seel, N. M., 2017. Model-based learning: a synthesis of theory and research. Educational Technology Research and Development 65 (4), 931–966.

Seel, N. M., Dinter, F. R., 1995. Instruction and mental model progression: Learner-dependent effects of teaching strategies on knowledge acquisition and analogical transfer. Educational Research and Evaluation 1 (1), 4–35.

Sharp, D. L., Bransford, J. D., Goldman, S. R., Risko, V. J., Kinzer, C. K., Vye, N. J., 1995. Dynamic visual support for story comprehension and mental model building by young, at-risk children. Educational Technology Research and Development 43 (4), 25–42.

Sheetz, S., Tegarden, D., 1998. Distributed capture of system requirements as individual and group cognitive maps. AMCIS 1998 Proceedings, 334.

Siau, K., Tan, X., 2005a. Improving the quality of conceptual modeling using cognitive mapping techniques. Data & Knowledge Engineering 55 (3), 343–365.

Siau, K., Tan, X., 2005b. Technical communication in information systems development: The use of cognitive mapping. IEEE transactions on professional communication 48 (3), 269–284.

Siau, K., Tan, X., 2006. Using cognitive mapping techniques to supplement uml and up in information requirements determination. Journal of Computer Information Systems 46 (5), 59–66.

Siau, K., Tan, X., 2008. Use of cognitive mapping techniques in information systems development. Journal of Computer Information Systems 48 (4), 49–57.

Sien, V. Y., 2011. An investigation of difficulties experienced by students developing unified modelling language (uml) class and sequence diagrams. Computer Science Education 21 (4), 317–342.

Sokolowski, J., 2009. Banks, cm principles of modeling and simulation, hoboken.

Soloway, E., Ehrlich, K., 1984. Empirical studies of programming knowledge. IEEE Transactions on software engineering (5), 595–609.

Sonnentag, S., 1998. Expertise in professional software design: A process study. Journal of applied psychology 83 (5), 703.

Sorva, J., 2013. Notional machines and introductory programming education. ACM Transactions on Computing Education (TOCE) 13 (2), 8.

Sorva, J., Lönnberg, J., Malmi, L., 2013. Students' ways of experiencing visual program simulation. Computer Science Education 23 (3), 207–238.

Stikkolorum, D. R., Chaudron, M. R., 2016. A workshop for integrating uml modelling and agile development in the classroom. In: Proceedings of the Computer Science Education Research Conference 2016. ACM, pp. 4–11.

Swan, J., Barker, T., Britton, C., Kutar, M., 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, pp. 10–pp.

Tang, A., Aleti, A., Burge, J., van Vliet, H., 2010. What makes software design effective? Design Studies 31 (6), 614–640.

Thomasson, B., Ratcliffe, M., Thomas, L., 2006. Identifying novice difficulties in object oriented design. ACM SIGCSE Bulletin 38 (3), 28–32.

Topi, H., Valacich, J. S., Wright, R. T., Kaiser, K. M., Nunamaker Jr, J., Sipior, J. C., De Vreede, G., 2010. Curriculum guidelines for undergraduate degree programs in information systems. ACM/AIS task force.

Torchiano, M., 2004. Empirical assessment of uml static object diagrams. In: Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004. IEEE, pp. 226–230.

Travassos, G., Shull, F., Fredericks, M., Basili, V. R., 1999. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In: ACM Sigplan Notices. Vol. 34. ACM, pp. 47–56.

Unhelkar, B., 2005. Verification and validation for quality of UML 2.0 models. Vol. 2. Wiley Online Library.

Visser, W., Hoc, J.-M., 1990. Expert software design strategies. In: Psychology of programming. Elsevier, pp. 235–249.

Von Mayrhauser, A., Vans, A. M., 1996. Identification of dynamic comprehension processes during large scale maintenance. IEEE Transactions on Software Engineering 22 (6), 424–437.

Westphal, B., 2019. Teaching software modelling in an undergraduate introduction to software engineering. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). IEEE, pp. 690–699.

White, B. Y., Frederiksen, J. R., 1990. Causal model progressions as a foundation for intelligent learning environments. Artificial intelligence 42 (1), 99–157.

Whittle, J., Bull, C., Lee, J., Kotonya, G., 2014. Teaching in a software design studio: Implications for modeling education.

Wiedenbeck, S., 1986. Beacons in computer program comprehension. International Journal of Man-Machine Studies 25 (6), 697–709.

Wijnen, F. M., Mulder, Y. G., Alessi, S. M., Bollen, L., 2015. The potential of learning from erroneous models: comparing three types of model instruction. System dynamics review 31 (4), 250–270.

Winn, W., 1994. Contributions of perceptual and cognitive processes to the comprehension of graphics. In: Advances in psychology. Vol. 108. Elsevier, pp. 3–27.

Wohlin, C., Aurum, A., 2004. An evaluation of checklist-based reading for entity-relationship diagrams. In: Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No. 03EX717). IEEE, pp. 286–296.

Xie, B., Nelson, G. L., Ko, A. J., 2018. An explicit strategy to scaffold novice program tracing. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education. ACM, pp. 344–349.

List of Publications

Conference Papers

1. **Prasad, P.**, & Iyer, S. (2020, August). How do Graduating Students Evaluate Software Design Diagrams?. *In Proceedings of the 2020 ACM Conference on International Computing Education Research* (pp. 282-290).

2. **Prasad, P.**, & Iyer, S. (2020, June). VeriSIM: a learning environment for comprehending class and sequence diagrams using design tracing. *In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training* (pp. 23-33).

3. Narayana, S., **Prasad, P.**, Lakshmi, T. G., & Murthy, S. (2016, December). Geometry via Gestures: Learning 3D geometry using gestures. *In 2016 IEEE Eighth International Conference on Technology for Education (T4E)* (pp. 26-33). IEEE.

4. Lakshmi, T. G., Narayana, S., **Prasad, P.**, Murthy, S., & Chandrasekharan, S. (2016). Geometry-via-Gestures: Design of a gesture based application to teach 3D Geometry. *In Proceedings of the 24th international conference on computers in education* (pp. 180-189). Mumbai, India: Asia-Pacific Society for Computers in Education.

5. Deep, A., **Prasad, P.**, Narayana, S., Chang, M., & Murthy, S. (2016, July). Game Based Learning of Blood Clotting Concepts. *In 2016 IEEE 16th International Conference on Advanced Learning Technologies (ICALT)* (pp. 526-530). IEEE.

6. Alse, K., Ganesh, L., **Prasad, P.**, Chang, M., & Iyer, S. (2016, July). Assessing Students' Conceptual Knowledge of Computer Networks in Open Wonderland. In *2016 IEEE 16th International Conference on Advanced Learning Technologies (ICALT)* (pp. 513-517). IEEE.

Posters

1. **Prasad, P.**, & Iyer, S. (2020, November). Inferring Students' Tracing Behaviors from Interaction Logs of a Learning Environment for Software Design Comprehension. *In Koli Calling'20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (pp. 1-2).

2. Reddy, D., Alse, K., Lakshmi, T.G., **Prasad, P.**, & Iyer, S. (2021, March). Learning Environments for Fostering Disciplinary Practices in CS Undergraduates. *In SIGCSE 2021*: To appear.

3. **Prasad, P.** (2018, August). Developing Students' Cognitive Processes Required for Software Design Verification. *In Proceedings of the 2018 ACM Conference on International Computing Education Research* (pp.284-285). ACM.

Acknowledgements

It's been said that it takes a village to bring up a child. Looking back at my PhD journey, I've come to realise that the same holds true for bringing this thesis to its completion as well. There has been a 'village' of people who have supported me during my PhD, and I take this opportunity to thank them.

First, I would like to thank my guide Prof. Sridhar Iyer for his support and guidance throughout my PhD journey. He has been more confident of my ideas and abilities than me. The confidence I have gained over the course of my PhD, I owe it to him. I have always come out of every thesis meeting or discussion with much more clarity and enthusiasm towards my research. His keen eye for detail, as well as his focus on the big picture has helped me become a better researcher.

I would like to thank my RPC members Prof. Sahana Murthy and Prof. M. Sasikumar for providing their valuable comments and feedback to my work. Their critical insights have been instrumental in shaping various parts and arguments in the thesis. I would also like to thank Prof. Ramkumar Rajendran, Prof. Chandan Dasgupta, Prof. Ritayan Mitra and Prof. Sanjay Chandrasekharan, for their conversations and guidance, and for their courses which initiated me into EdTech research.

I would like to thank my batchies Lakshmi, Soumya, and my immediate seniors Anurag and Kavya for their friendship. I would particularly like to thank Lakshmi for her inputs and advice right from the start of my PhD. She's been the "enthu" of our batch, and her eagerness enabled me to push forward in my own research as well. She has been my sounding board for initial ideas, research designs, and analysis, and I have always benefited from discussing things with her. I'm thankful to Anurag (who being my immediate senior) has always given me practical advice on what I should do next, and how I can get there.

I am thankful to the seniors and alumni of my department - Aditi, Shitanshu, Rwito, Kapil, Vikram, JK, Gargi, Yogi and Deepti. The interactions and stimulating discussions I've had with them in and outside the lab have greatly shaped my perspective towards EdTech research. I'm also thankful to my ET labmates - Ashu, Pankaj, Rumana, Navneet, Shiv, Narasimha, Lucian, Herold, Ulfa, Priya and Veenita, for making the lab a fun and engaging place to work.

Many people have contributed to the development and outreach of VeriSIM. I'm especially grateful to Bhupender and Kinnari for their enthusiasm and eagerness to develop various aspects of the learning environment. I thank my former colleagues in Fr. CRCE Bandra, and Dr. Deepti Reddy from SIES, for providing the opportunity to conduct my research studies at their institute. I would also like to thank Ajit for his technical support and Sunita for accompanying me in my research studies. The ET office staff have always been ready to help, and I'm indebted to Pallavi, Prakash, Seema, and Vidya for assisting in all administrative tasks, so that I could focus on my research.

I owe an immense amount of gratitude to my family, for their unconditional love and support. I thank Appa and Amma for their love, sacrifices, and prayers, without which I would not be who I am today. I thank my sister Pritty, and my brother-in-law Pramod for all their love and concern. My uncles, aunties and cousins, especially Achapacha and Achamachi, thank you for periodically calling and encouraging me. To my wife's parents - Pa and Ma, thank you for your constant support and prayers. To my wife's extended family in Mumbai - Chachu, Shinymama, Sefi, Blesson, Senith, Kunjchachu, and Bindumama - thanking you all for the good times we shared together. The time spent with all of you provided a welcome relief to me from the stresses and strains of research.

I am greatly blessed to know that there are people who have been praying for me. I thank my pastors, Rev. Gigi Thomas and Rev. Thomas Mathew, for praying for me, and for periodically enquiring about my research progress. Ajo, Christy, Titus - thank you for your brotherly love, concern and prayers. I am also indebted to my christian friends on campus for the enriching experiences and fellowship - Dr. Joseph John, Dr. Sameer Jadhav, Dr. Tom Mathew, Tim, Prakash, Keerthi, Nikhil, Maria, Srinivas, Lokesh, Vishwanath, Ajay, Fabian, Elsa, Sibi, and Suma.

My daughter Sera entered my life midway into my PhD, and life hasn't been the same since then. She has made the journey much sweeter and much more fun. Her curiosity, imagination and liveliness has inspired me to look at research and the world around me with renewed enthusiasm. To my wife Sheba, who has stood by me through the ups and downs of this arduous journey - thank you for the innumerable sacrifices you've made. Your love, prayers and support kept me going, and pushed me to give my best towards my research. Above all, I would like to thank my Lord and Saviour Jesus Christ, for guiding me, sustaining me, and taking me through this journey. There have been uncertain and trying times, but knowing that I am loved, and that His presence is all I need to take me through has given me courage and peace to press on. And for that I will be eternally thankful.