MODEL TRACING OF CODING STYLES OF PROGRAMMERS:
A FORMATIVE APPROACH

by

Shilpi Rao
B.E. (Comp. Eng.), DJ Sanghvi College of Engineering, Mumbai University, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Interactive Arts and Technology

© Shilpi Rao 2007
SIMON FRASER UNIVERSITY
2007

All rights reserved. This work may not be reproduced in whole or in part, by photocopy
or other means, without the permission of the author.
APPROVAL

Name: Shilpi Rao
Degree: Master of Science
Title of thesis: Model Tracing of Coding Styles of Programmers: A Formative Approach

Examining Committee: Dr. Lyn Bartram
Chair

Dr. Marek Hatala, Senior Supervisor,
Associate Professor, Interactive Arts and Technology
Simon Fraser University, Canada

Dr. Vive Kumar, Supervisor,
Senior Lecturer, Information Sciences and Technology
Massey University, New Zealand

Dr. Halil Erhan, External Examiner,
Assistant Professor, Interactive Arts and Technology
Simon Fraser University, Canada

Date Approved: Aug 10, 2007
Declaration of Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the “Institutional Repository” link of the SFU Library website <www.lib.sfu.ca> at: <http://ir.lib.sfu.ca/handle/1892/112>) and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author’s written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada

Revised: Fall 2007
STATEMENT OF ETHICS APPROVAL

The author, whose name appears on the title page of this work, has obtained, for the research described in this work, either:

(a) Human research ethics approval from the Simon Fraser University Office of Research Ethics,

or

(b) Advance approval of the animal care protocol from the University Animal Care Committee of Simon Fraser University;

or has conducted the research

(c) as a co-investigator, in a research project approved in advance,

or

(d) as a member of a course approved in advance for minimal risk human research, by the Office of Research Ethics.

A copy of the approval letter has been filed at the Theses Office of the University Library at the time of submission of this thesis or project.

The original application for approval and letter of approval are filed with the relevant offices. Inquiries may be directed to those authorities.

Bennett Library
Simon Fraser University
Burnaby, BC, Canada
Abstract

Programming Style refers to the ability to follow code conventions, to engineer code in a disciplined manner, to systematically debug code, to optimize code delivery through appropriate settings in the IDE (Integrated Development Environment), to regulate completion rates and quality of programming tasks, and finally to efficiently collaborate with other programmers and resources. This research investigates whether programming styles of individual programmers can be computationally recognized; if styles can be recognized by the machine, can they then be regulated so that programmers can reflect on their own programming styles; finally, can a mixed-initiative computational mechanism assist programmers to identify good programming styles and repair bad programming habits. My research focuses on a real-time architecture called MICE (Mixed-Initiative Coding Environment) that I have developed to help programmers to reflect on their coding style and correct their style.

Keywords:
Ontology, Rules, Human-Computer Interaction, Mixed-Initiative Interactions, Self-Regulated Learning, Model-Tracing, Intelligent Tutoring System, Software Engineering, Help-System, Java programming, Programming styles, personalised and customised help
Dedicated to my mommy, Savita Rao :) and daddy
“Life can give you 100 reasons to Cry, but you can give Life 1000 reasons to Smile!”

— Anonymous
Acknowledgments

It wouldn’t have been possible to complete this thesis by myself. There are many people who have helped and encouraged me with my thesis, knowingly or unknowingly. Here I would like to acknowledge all the people who provided their support, help, and encouragement.

Good supervisors are a hard and lucky find. And, I have two of the best supervisors I could ask for. Vive and Marek have both been highly encouraging supervisors. Dr. Marek Hatala, my senior supervisor, had meetings with me regularly, provided essential feedback, and kept track of my progress. Marek funded my research, as a result of which I could concentrate solely on my research and finish it early. Marek also funded a number of my conference travels.

Dr. Vive Kumar, my supervisor, helped me get introduced to this research area. Even though he moved to New Zealand, his regular emails and online chatting helped me to be on track with my work. He helped me to look at the research from various angles. While at SFU, Vive provided me scholarships and also travel funds to attend a number of conferences.

I am thankful to LearningKit (SSHRC) and LORNET (NSERC) for providing funding for my research.

I also want to thank Dr. Halil Erhan for agreeing to become my external examiner.

I owe a great deal to Dragan Gasevic, Ty Mcy Eap aka Timmy, and Nima Kaviani for being there in the research lab to help me with any doubts or general discussion about my thesis. They provided a great working environment and are an inspiration to work with. And how can I forget, the coffee breaks and Timmy’s food supply.

Chris Groeneboer, has been so helpful not only with my thesis but also helping me cool down. She went through my entire thesis and helped me come up with my experimental design. She has so much faith in my thesis and is so excited about it. She can’t wait to start using it :) .
I would also like to thank Gordon Pritchard, Justin Thomas and Gordon Manson for helping me deal with computer, printer or any other lab-related problems. Their help has always been fast and immediate. Joyce Black had been so supportive to answer every question I had about the masters program and helping me with my thesis deadlines. It is because of their help; the work is on time and smooth.

I would like to thank all my friends, who helped me not only with the thesis but especially take those important breaks that helped me relax and not stress out too much - Steven Barnes helped me with my statistical analysis - without his help I would have spent days on it, Jurika Shakya - to get me introduced to Latex, Davis Marques to help me with all the LaTeX doubts, Hasaan Malik - the funny guy who cheered me up all the time and tolerated me even when I was totally frustrated, Patrick Phang for all his professional help, and Kirsten - the movie maker who introduced me to the movie making world and all the challenges she faced to do so (that helped me appreciate my thesis even more ;). I would also like to express gratitude to my lab mates Ai Nakatani, Malahat Hosseini, Davis Marques, Kirsten, Vidya, Steve, Eddie Hou, Susan Olubunmi, and Jack Sam who were always there to hang out and listen to my talks, be it work related or anything on this earth. With friends like them in Canada, I didn’t miss my family much.

Ashok Shah and Jurika Shakya, my ex-roommates, have been such an inspiration for me. They not only taught me about work and studies but also other essential stuff to survive in Canada - house work :P

My mom, Savita Rao, has always encouraged me to keep faith in myself and motivated me to select the right path. My mom and family have always been in touch with me even though they are miles apart and have motivated me throughout the process. Their love and faith is highly inspirational. Same goes for my friends from Mumbai. Darshan Mody my best buddy has always entertained me with his witty jokes and his take on the world. He along with Ruchita Vora, Nirav Shah, Kunal Jethwani, and Ankit Choudhari have always been there for me, always ready to help and ready to talk about anything and everything. They are the best supportive bunch I could ever ask for.

A fit body is a fit mind. I would like to thank my gym friends Jenn Portillo, Cari Plotnikoff, and Susan who encouraged me to come to the gym regularly and helped me keep fit throughout my study.
Contents

Approval ii
Abstract iii
Dedication iv
Quotation v
Acknowledgments vi
Contents viii
List of Tables xii
List of Figures xiii
List of Programs xv

1 Introduction 1
   1.1 Computer Programming and MICE 2
   1.2 MICE tools 6
   1.3 Programming Style 7
      1.3.1 Code Conventions 8
      1.3.2 Planning 8
      1.3.3 Code Engineering 8
      1.3.4 Optimal IDE Settings for Coding 11
      1.3.5 Collaboration While Coding 11
## 2 Literature Review

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Introduction</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Knowledge Engineering</td>
<td>13</td>
</tr>
<tr>
<td>2.2.1 Ontology</td>
<td>14</td>
</tr>
<tr>
<td>2.2.2 Rules</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Human-Computer Interactions</td>
<td>17</td>
</tr>
<tr>
<td>2.3.1 Human-in-the-loop</td>
<td>18</td>
</tr>
<tr>
<td>2.3.2 Mixed-Initiative Interactions</td>
<td>19</td>
</tr>
<tr>
<td>2.3.3 Feedback</td>
<td>20</td>
</tr>
<tr>
<td>2.4 Self-Regulated Learning</td>
<td>21</td>
</tr>
<tr>
<td>2.4.1 SRL Characteristics and Components</td>
<td>22</td>
</tr>
<tr>
<td>2.4.2 Self-Regulated Learning model</td>
<td>23</td>
</tr>
<tr>
<td>2.5 Intelligent Tutoring Systems</td>
<td>25</td>
</tr>
<tr>
<td>2.5.1 Challenges and Opportunities</td>
<td>26</td>
</tr>
<tr>
<td>2.5.2 Model-Tracing and Constraint-Based Models</td>
<td>28</td>
</tr>
<tr>
<td>2.6 Stages of Programming</td>
<td>31</td>
</tr>
<tr>
<td>2.7 Summary</td>
<td>33</td>
</tr>
</tbody>
</table>

## 3 MICE Architecture

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 The Functional Architecture of MICE</td>
<td>34</td>
</tr>
<tr>
<td>3.2 The Technical Architecture of MICE</td>
<td>36</td>
</tr>
<tr>
<td>3.3 MICE Ontology</td>
<td>37</td>
</tr>
<tr>
<td>3.3.1 Interaction Ontology</td>
<td>37</td>
</tr>
<tr>
<td>3.4 Feedback</td>
<td>39</td>
</tr>
</tbody>
</table>

## 4 Research Design

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Framework of the Method</td>
<td>43</td>
</tr>
<tr>
<td>4.2 Data Collection and Management</td>
<td>44</td>
</tr>
<tr>
<td>4.2.1 Sample</td>
<td>44</td>
</tr>
<tr>
<td>4.2.2 Data Management</td>
<td>45</td>
</tr>
<tr>
<td>4.3 Measurement of Variables</td>
<td>45</td>
</tr>
<tr>
<td>4.3.1 Research Instruments</td>
<td>45</td>
</tr>
<tr>
<td>4.4 Data Analysis</td>
<td>46</td>
</tr>
<tr>
<td>4.5 Assumptions, Strengths, Weaknesses</td>
<td>48</td>
</tr>
</tbody>
</table>
5 Results and Analysis
5.1 Research Question 1 ................................................. 49
  5.1.1 Number Of Bytes (NOB) versus Compile Time Graph ........ 50
  5.1.2 Un/Succesful Compiles Graph ................................ 55
  5.1.3 Error Pattern Graph ........................................... 58
  5.1.4 Error Type Encountered Graph ................................ 60
  5.1.5 Error Solving versus Compile Time Graph ....................... 60
  5.1.6 Files Referenced Graph ........................................ 61
  5.1.7 BlueJ Tools and Software Engineering/SRL techniques ......... 63
5.2 Research Question 2 .............................................. 64
5.3 Research Question 3 ............................................... 69
5.4 Research Question 4 ............................................... 71
5.5 Research Question 5 ............................................... 72
5.6 Research Question 6 ............................................... 74
5.7 Programmer Participants Evaluation of MICE ......................... 79
  5.7.1 Comparative Graphs ............................................. 84
  5.7.2 Error Line Number versus Compile Time Graph ................. 88

6 Future Work .......................................................... 92

7 Conclusion ............................................................ 96

A Complexity Analysis .................................................. 98

B Error Pattern Graphs .................................................. 100

C Error Type Encountered Graphs ....................................... 107

D Question Set ........................................................... 110
  D.1 Sample Novice Question Set ...................................... 110
    D.1.1 Question Set 1 ................................................. 110
    D.1.2 Question Set 2 ................................................. 110
  D.2 Sample Intermediate Question Set ................................ 111
    D.2.1 Question Set 1 ................................................. 111
    D.2.2 Question Set 2 ................................................. 112
List of Tables

5.1 Tool/Strategy Usage and Code Analysis - Participant 1 .......................... 63
5.2 Tool/Strategy Usage and Code Analysis - Participant 7 .......................... 65
List of Figures

3.1 Functional Architecture of MICE ........................................... 35
3.2 A Scenario Interactions being Tracked and Graph Generation .......... 37

5.1 NOB Versus Compile Time - Participant 4 ............................... 52
5.2 NOB Versus Compile Time - Participant 3 ............................... 53
5.3 NOB Versus Compile Time - Participant 17 ............................. 54
5.4 NOB Versus Compile Time - Participant 14 ............................. 54
5.5 Unsuccessful and Successful Compiles - Participant 2 ................. 56
5.6 Unsuccessful and Successful Compiles - Participant 17 ............... 57
5.7 Unsuccessful and Successful Compiles - Participant 7 ................. 58
5.8 Unsuccessful and Successful Compiles - Participant 21 ............... 59
5.9 Error Solving Versus Compile Time - Participant 6 .................. 61
5.10 Error Solving Versus Compile Time - Participant 17 ................. 62
5.11 Files Referenced - Participant 1 ........................................... 62
5.12 Forethought Phase of All Participants .................................... 66
5.13 Performance Phase of All Participants ................................... 67
5.14 Reflection Phase of All Participants ..................................... 68
5.15 Forethought Phase .......................................................... 70
5.16 Performance Phase ......................................................... 70
5.17 Reflection Phase .......................................................... 71
5.18 Code Analysis of All Participants ......................................... 73
5.19 Instructors Evaluation of MICE ............................................. 76
5.20 Q.1 How confident do you feel about using the system after this experiment? 80
5.21 Q.2 How much effort did it require to learn? ............................ 81
5.22 Q.3 How helpful was the system in providing feedback to identify your good
and bad programming habits? ........................................ 82
5.23 Q.4 How helpful was the comparison between your programming style; and
an experienced and a non-experienced programmer? ............ 83
5.24 Comparing Code Construction - Participant 3 with In/Experienced Program-
mers ................................................................. 85
5.25 Comparing Code Construction - Participant 17 with In/Experienced Pro-
grammers .............................................................. 85
5.26 Comparing Code Construction - Participant 21 with In/Experienced Pro-
grammers .............................................................. 86
5.27 Comparing Error Solving - Participant 21 with In/Experienced Programmers 87
5.28 Q.5 How helpful was the system to localize your bugs? .............. 88
5.29 Error Line Number Versus Compile Time - Participant 7 ............ 89
5.30 Q.6 How helpful was the system in helping you identify concepts in Java that
you might have difficulties with? ........................................ 90
5.31 Q.7 How much did you like the BlueJ IDE for your programming needs? ... 91

6.1 MICE Functional Architecture ........................................ 94

B.1 Error Pattern - Participant 17 - Program 1 .......................... 101
B.2 Error Pattern - Participant 17 - Program 2 .......................... 102
B.3 Error Pattern - Participant 18 - Program 1 .......................... 103
B.4 Error Pattern - Participant 18 - Program 2 .......................... 104
B.5 Error Pattern - Participant 14 - Program 1 .......................... 105
B.6 Error Pattern - Participant 14 - Program 2 .......................... 106

C.1 Error Pattern - Participant 21 - Program 1 .......................... 108
C.2 Error Pattern - Participant 21 - Program 2 .......................... 109
List of Programs
Chapter 1

Introduction

The instructional design of many introductory programming courses in computer science do not include introduction to "good" programming styles. In most cases, these courses offer introduction to syntax constructs of the programming language, means to translate problem statements to program design, how to handle a compiler, and notably only summative feedback to students on their code. Skill development in programming is an evolutionary process. Skilled programmers exhibit expertise, also referred as programming style, in language constructs, design issues, efficient debugging, standards requirements, best practices, coding resources, and ability to successfully collaborate with colleagues. Many of these essential skills are not taught in classrooms. Even in a handful of programming courses where attempts are made to introduce these essential programming skills to students, there has been a severe lack of methodology, scope, and instruction time for these introductions. In addressing this, this thesis proposes a pedagogical mechanism to guide students through a formative process that paves way for a formal, theory-centric, standardized introduction to the aforementioned essential skills of programming. Further, this formative process also introduces novel opportunities for instructors to base their assessment methods on skill development thus motivating students to focus also on the processes involved in programming skill development rather than just the end result of skill acquisition.

Restricting feedback only to summative notions not only plays a major role in determining the quality of students' designing and coding capabilities, but also deprives instructors an opportunity to guide students towards an optimal pathway to teach programming. Having deprived of formative guidance, students are more likely to develop improper or undesirable styles of coding which may make future refinement and amalgamation of coding in an
industry-oriented environment very difficult.

Process feedback is information about how students engage in a task [96]. Poor coding styles may be remedied by capturing the process of how novice students learn to program and by providing system-initiated, process-oriented, formative feedback about their programming behaviour [83].

Madeo and Bird [65] advocate that to be effective, weaker students or novice programmers should be taught programming in a very tightly structured environment where instructions are given about the next expected process rather than let the student handle the programming problem-solving process by themselves. While this may be true in most cases, it would be desirable to allow instructors to determine the degree to which students are confined to the structure of the environment, which is only possible in an environment that is conducive not only to summative feedback but also to formative feedback.

A strong programming style will lead to high quality, efficient, well tested, less error prone, easy to extend code, capability to debug easily and to seek help from apt sources at appropriate times. This research aims to design and develop a system called MICE (Mixed-Initiative Coding Environment) to proactively and reactively assist programmers regulate their coding styles when they code in Java using an Integrated Development Environment (IDE). We aim to show the possibility of capturing formative processes of students' coding styles at real time. Further, we aim to show how such a formative approach to coding helps nurture desirable programming styles.

1.1 Computer Programming and MICE

Computer programming potentially involves skills in understanding (application context and possibility), planning (design), imaging (imagination and visualization), attitude (acceptance of work involved and confidence in completing projects), logic (conceptualization, language use, and knowledge), creativity (artistry) and work (persistence, exploration, purpose and commitment) [101]. While applying these skills to develop code, programmers acquire individual styles.

We define Programming Style as processes that a programmer adopts, over a period of time, to achieve specific programming goals. The scope of this research defines Programming Style as the ability of programmers:

- to plan their goals,
• to plan the sequence of tasks to achieve their goal,
• to follow code conventions,
• to engineer code in a disciplined manner [60, 26],
• to systematically debug code [108],
• to optimize code development and delivery through appropriate settings in IDE (Integrated Development Environment),
• to regulate completion rates and quality of programming tasks, and
• to efficiently collaborate with other programmers and resources.

These components mold the blueprint of what we consider as the Programming Style of an individual programmer. The current MICE version is embedded within the BlueJ IDE. Programmers can use BlueJ to develop Java code and use various tools provided by BlueJ to perform their programming tasks, while MICE runs in the background capturing all interactions that happen between the programmer and BlueJ. Essentially, MICE targets three objectives [85]:

• First, MICE aims to capture programming style of programmers, at real time, by observing the interactions of programmers with the IDE when they develop task-specific code.

• Second, MICE aims to promote the principles derived from the theory of Self-Regulated Learning (SRL) in its feedback mechanism. SRL views learning as an activity that students perform proactively, rather than as a covert event that happens to them in reaction to teaching [104]. MICE embeds an ontological representation of Zimmerman’s SRL model. According to Zimmerman’s SRL model [110], the forethought phase involves goal setting and strategic planning. Software engineering techniques corresponding to the first phase are - drawing flowcharts and writing algorithms, making test cases (black box testing), creating UML diagrams (class diagrams, use cases, sequence diagram, etc.) for strategic planning, and referring to Internet or books. The second phase of Zimmerman’s SRL model is associated with performance of the task, which involves execution of the activities planned in the first stage, and self-monitoring. With respect to coding, this stage involves code development, debugging
the code, and running test cases. And the last stage of Zimmerman’s model is associated with self-reflection, which entails comparison of one’s performance with one’s previous performance or others’ performance or an absolute standard. In coding, this corresponds with analysis of code complexity, analysis of readability of the program, removal of unused code, and reflection on coding style. Thus, one can establish a clear and direct mapping between coding interactions and phases of Zimmerman’s SRL model.

- Third, MICE aims to develop software mechanisms to enable Mixed-Initiative strategies so that conversants, in our case, programmers and the MICE system, can take initiate and ask/provide feedback at appropriate times.

Experiments have been conducted to show that programmers do not possess a fixed coding style. Programmers use different coding styles or use the same style for different problems. Samin et al. [89] conducted an experiment to observe the influence of SRL-specific feedback, and noted significant variance in frequency of compiles, changes in lines of code between compiles, frequency of errors and warnings, and correctness of the final code when students received SRL-specific feedback. However, in the experiment by Samin et al., the software system forced the participants to use SRL components quite explicitly. Also, no immediate feedback was given to the participants in Samin’s experiment about their code development style. On the contrary, MICE tracks interactions in the background without students being aware of them (i.e., students neither perform any additional tasks nor are forced to use explicit SRL-specific software components), and the tools that promote SRL (for example, tools available in BlueJ IDE) are available passively and it is left to the programmer to engage in SRL-specific activities.

Using MICE, students can view their development style and compare their styles with that of others. MICE offers feedback that is both dynamic and immediate. Also, it is important to guide programmers at a suitable time and bother them only when required [69]. For example, if a programmer regularly encounters errors of the same type and he takes too long a time to solve them, then it is safe to assume that it is a conceptual error and no longer a ‘typo’ error. Thus, at this juncture, feedback to student can include suggestions to read about the associated concept and example programs related to the concept. In offering such contextual assistance, MICE can initiate one or more of the following feedback methods at opportune moments:
CHAPTER 1. INTRODUCTION

1. MICE can provide the programmer with timely clues and hints. Such hints could be to consult a helper, to use a resource to learn a concept, to change their coding style, to compare their style with others, to perform SRL-specific activities, to check the code complexity, and to target specific input/output requirements. Along with these hints, appropriate information or tool are provided so that programmers can transparently (i.e., from within MICE) engage in these suggested activities.

2. Not many software development environments offer theory-oriented feedback. MICE can offer the programmer SRL-specific feedback. It does that by allowing programmers to graphically view their programming styles and compare their styles with that of others. Also, having a handle on programming styles of individual students as well as that of groups of students can help instructors to regulate and modify their own teaching style.

In order to investigate the research objectives of capturing programming style of a programmer, promoting SRL in programming, and giving mixed-initiative feedback, we evaluated the utility of MICE with respect to its end users - the programmers and the instructors. We developed the following research questions that target specific components of these objectives:

1. Do student programmers have “identifiable” programming styles?

2. Do student programmers “exhibit” self-regulatory abilities and are they inclined to use self-regulatory tools embedded in the IDE, even when these tools are only available passively?

3. Do intermediate student programmers exhibit more self-regulatory abilities than novice student programmers?

4. Are novice programmer’s faster learners of SRL principles than the intermediate programmers? Can SRL be taught progressively?

5. Are programmers open enough to change their programming style and does this change lead to better programs?

6. Does student programmer’s data provide appropriate context for the teachers to identify their weak programming styles and give feedback to student to improve upon their
programming styles identified in this manner? Does this data offer help to instructors
to provide more customized and personalized help to the student? Will instructors
take appropriate measures to encourage usage of SRL in their courses? Do instructors
trust the data collected by the system?

Detailed analyses of each of these hypotheses are addressed in Section 5 of the thesis
based on a carefully designed experiment described in Section 4.

1.2 MICE tools

MICE maintains a programming context for each programmer which can be viewed and used
by the system, peers, and instructors. This context is also used by programmers themselves
to self-regulate their coding style. The context consists of code development style/s used
by the programmer - how the code is developed, how the code is debugged, how the code
quality is maintained, and how self-regulatory engagements are enacted. The context also
includes programmer preferences and their current tasks.

In MICE, this contextual information is stored in an ontology and updated dynamically
as and when an event occurs. This event-specific Interaction Ontology records interactions
of programmers observed within the MICE system and the IDE. Time-stamped interactions
of each programmer can be extracted from the ontology.

At any point of time, the information in the ontology can be viewed by the programmer,
the system, or helper. This information helps the programmer to review and reflect on
programming styles and modify them, if so desired. These changes in programming styles
can also be observed over a period of time. Instructors use this context to not only provide
feedback to the student but also to see how their teaching methodology affects students’ pro-
gramming styles. An undesirable programming style observed among a significant number
of programmers in a class invariably points to the deficiency of instruction.

There are acute implications that one should be aware of prior to employing context-
oriented tools in learning environments. An experiment conducted by Kumar [58] reveals
that understanding the context is time consuming and is a complex cognitive task. Review-
ing the context of a programmer prior to providing help is regarded as an additional work.
However, once the context is understood, peers and instructors can provide quality help spe-
cific to the needs of the programmer. The experiment also showed that programmers were
satisfied and preferred such context-specific advice over typical context-free advice. Thus,
once the initial barrier of context being treated as an additional work is negotiated, it will prove to be very useful. With this in mind, MICE has been designed to collect formative information specific to programming and abstract the same in an ontology for the perusal of student programmers as well as the instructors.

Most programmers employ an IDE (Integrated Development Environment) embedded with a compiler. They start to write code in the IDE and from time to time submit their code to the compiler for verification. When a programmer submits his/her partially or fully developed code to a compiler, the code that goes through the verification process is identified as a compile-time code segment (CT-SEG). The compiler returns a list of errors and warnings present in the code and a details about those errors and warnings - such as type of error and warning, the line number where the errors/warnings occurred, and so on. Once the program has been compiled successfully, it can be executed/run to view the output generated by the code. MICE tracks these interactions with respect to each compilation, collects code changes programmers make in between each pair of compiles, and updates these information in the ontology.

1.3 Programming Style

In this section we identify our assumptions concerning programming styles. Also, we highlight how each of these style components relates to our research.

The first assumption is that a programmer does not have to adhere to a single recognized programming style in a given period of time. Programmers can use one or more styles and switch styles as often as they like. This change can be due to numerous factors such as complexity, type of the programming task, time availability, help available, coding expertise, and so on. Instead of attempting to stereotype the observed styles of a programmer to a particular type, the scope of MICE enables the identification of a variety of styles exhibited by a programmer over a period of time.

The second assumption is that there is no one good programming style. The best-suited programming style/s may vary from programmer-to-programmer.

Based on these assumptions, we extend the notion of programming styles in two main aspects: first, programming style is a problem-solving process and hence can possibly change from context to context over a longer period of time; second, programming style includes a number of other factors outlined below, and a programmer can engage in one or more of
these factors that determines the individual programming style [85].

1.3.1 Code Conventions

Code Conventions is defined as the ability of a programmer to adhere to specific conventions prescribed for a particular language. For example, Java Code Conventions include usage of tabs, indents, blank lines, spaces, alignments, braces, wrapping, naming, file organization, documentation, language construct statements, and imports¹.

1.3.2 Planning

Planning is an important task in programming. Programmers decide what their goals are, break the goals into numerous achievable tasks, and decide on methods to implement those tasks. Planning takes place throughout the coding process. Programmers write algorithms, draw flowchart, develop sequence diagrams, and/or develop state chart diagram to determine the sequence of coding actions that accomplish the target tasks. They also use books, cross-reference other programs, and use the Internet as part of their planning activity. BlueJ IDE provides tools for making sequence diagrams and to construct UML diagram to help programmers plan their tasks. The BlueJ IDE uses Class Diagrams in UML notation to display their project. Using such graphical representation for developing Classes lead to better design practices [29]. Novice programmers prefer this method of project presentation than the tree structures used by most editors [29]. Also, the Display UML tool can be used to view the properties (variables, methods, constructors within the class; and their return types and information about the input arguments) of each class in the UML syntax [57]. BlueJ includes a Sequence Diagram Editor to draw the sequence diagrams. If a programmer is observed not to use these tools as part of their planning activities and if the observed coding style is not to the expectations of the instructor, then MICE, guided by the instructor, can ask the programmer to use these tools.

1.3.3 Code Engineering

Code Engineering is defined as the ability of a programmer to construct code in a disciplined manner, preferably using sound software engineering principles. Normally, code engineering, among others, involves designing code, typing-in or pasting-in language constructs,

CHAPTER 1. INTRODUCTION

compiling, version control, code refactoring, and using templates/patterns.

**Code Construction**

Kumar [60] and Doherty et al. [26] discuss a simple tool that recognizes code construction styles of programmers based on compile-time code segments (CT-SEG). Compile-time code segments are code (partial or complete) submitted to the compiler by programmers for verification of correctness. That is, every time a programmer submits code for compilation, a version of the code (CT-SEG) is saved, thus enabling the tool to trace the ability of the programmer to incrementally construct code.

A study conducted by Kumar shows that the number of CT-SEG and the pattern of CT-SEG vary across programmers. That is, programmers compiled their code at varying time intervals irrespective of the difficulty of the task of their coding expertise. In the same study, Kumar [60] also observed the lines of code (LOC) between compiles. It showed different code construction styles of student programmers. Also, programmers’ behavior can be tracked as a function of change in language constructs in specific debugging contexts. We believe that such syntax-level tracking can further be advanced to semantic-level tracking.

**Code Quality**

Code is expected to be of good quality. That is, it should be less complex and should have undergone rigorous testing. By less complexity we mean, easy to read, easy to understand, easy to extend, easy to maintain, and easy to test the code. Further, code can be made more readable and less confusing by removing unreachable methods and redundant fields from the code [98]. Lines of Code (LOC) method and Function Points Analysis (FPA) are commonly accepted complexity determination methods. MICE uses BlueJ’s Class Evaluator tool to evaluate the classes and determines their complexity using Chidamber and Kemerer metrics [15] (metrics are discussed in detail in Appendix A). MICE uses this tool to make programmers aware of the complexity of their code.

Tom Copeland’s PMD tool in BlueJ [57] helps to find unused variables, code, and empty blocks. It presents these results to the programmer, so that the unused portions can be removed (or commented away) to make the program less complex, less error-prone, and easy to read.

Testing plays an important role in determining the quality of code. Majority of errors
can be detected during testing if the test cases explore a significant number of test paths. Individual classes and methods can be tested in BlueJ using adhoc testing, which is a type of testing to locate errors without preplanning or documentation. JUnit is a testing framework that can be incorporated with BlueJ to perform unit and regression testing, where tests can be recorded and replayed [76, 56]. Units in object-oriented programming refer to individual testing of methods and classes [56]. This test functionality helps to increase the confidence of programmers, by ensuring that changes made to the code have not broken existing functionalities. BlueJ also contains a built-in, interactive, graphical, unit testing component that allows testing of object-oriented code without writing a single new line of code [56]. This feature is especially advantageous for the novice programmers because they can test their code without having to code any lines for writing test cases [87].

**Code Debugging**

Debugging is an art and is associated closely with code-engineering. However, because of the complexity involved in tracing programmers’ debugging tactics and strategies, we treat code debugging outside the scope of code engineering. Typically, programmers employ a range of automated debugging techniques [108] that are listed below.

- Delta debugging - automatically narrows down the difference between a passing and a failing run
- Program slides - separates the part of a program or program run relevant to the bug
- Observing state - uses a debugger to observe the values of variables
- Watching state - uses a debugger to watch small parts of the program state to determine if they change during execution
- Assertions - uses comparison of observed values with the intended values when observing a program state

The automated debugging techniques do not capture debugging patterns over a period of time. In MICE, we are interested in observing how well programmers are able to debug code in between compilations and also over longer sessions of programming. That is, the number of type errors and warnings produced by the compiler can be stored whenever a programmer submits code for compilation. MICE can track a programmer’s errors and
CHAPTER 1. INTRODUCTION

warnings across multiple compiles and record whether the programmer tries to solve errors and warnings as soon as they appear, or debugs only a select few errors and warnings, or continues coding without correcting them.

Kumar refers to a number of patterns of debugging [60]. Most participants in the study tried to eliminate errors as soon as they appeared and they completely neglected the warnings produced by the compiler. When these errors and warnings were compared with the lines of code (LOC) across various compiles, a pattern that indicated a marked change in LOC was observed. This change in LOC can vary from changing a few lines of code to changing or eliminating a major portion of the code, depending on the programmer’s debugging style.

Jeliot 3 is a graphical debugging tool embedded in BlueJ (used by MICE) that offers debugging services. The usage of this tool is tracked by MICE and the usage of Jeliot tool is a component of the debugging styles of programmers.

1.3.4 Optimal IDE Settings for Coding

The Integrated Development Environment (IDE) plays an important role in programmer productivity. An IDE is an environment that integrates multiple software engineering toolkits and presents the same to the programmer in a single interface. For example, the BlueJ IDE integrates and customizes toolkits associated with project management (e.g., importing and exporting projects) and code management (e.g., bundling program components (creating jar files), editing fonts and colors, using libraries, plugins and resources, compiler and debugger selection and setting). Programmer should be guided to use an optimal IDE setting to suit their individual programming styles, preferably based on pre-defined models of IDE settings used by experts.

1.3.5 Collaboration While Coding

Effectively collaborating with colleagues is crucial not only in extreme programming and other agile software development methodologies, but also in normal coding environment. For example, a programmer may casually shout across the room for clarification on a particular type of bug or share code with a chat friend for an informal code review. A number of tools support collaboration in terms of chat, discussion boards, and so on. Morris et al [69] discuss ways in which support software can passively as well as actively promote collaboration.
Similarly, iHelp provides an efficient model of code collaboration [10]. These external tools can be integrated with MICE to capture collaborative activities of a programmer.

In summary, we have enumerated a number of coding tasks that programmers' engage in and the suite of tools that help them accomplish those tasks. Programmers may use all or some of those tools, based on specific coding practices. These practices form the blueprint of a programmer's coding style. Most of the coding style interactions based on the aforementioned components are tracked by MICE dynamically and are updated in the programmers' ontology models. Based on these models, formative feedback can be offered to programmers concerning their programming style to nurture their self-regulatory abilities.
Chapter 2

Literature Review

2.1 Introduction

This literature review is directed towards techniques centered around theory-based, formative, help system for programmers - to help them to identify their coding styles, to modify and improve their coding styles, and to receive timely help from a software system concerning their coding styles. The review covers topics in Knowledge Engineering - how subject and process knowledge are stored and processed; Human-Computer Interaction - including contemporary help systems, contexts, and feedback mechanisms; Pedagogy - reviewing theories of Self-Regulated Learning and Mixed-Initiative Interaction; Technology - reviewing techniques and challenges in building computational models; and Software Engineering - helping students program better; and finally how this research derives pieces of knowledge from these areas and integrates them all together.

2.2 Knowledge Engineering

Knowledge engineering involves acquisition, representation, validation and re-use of knowledge [4]. Knowledge engineering is an important component for any system that deals with knowledge, for example, expert systems and machine learning systems.

An expert system consists of a knowledge base and an inference engine as its main components. A knowledge base is usually dynamic, that continuously updates ‘facts’ known to the system and ‘executes’ conditions under which changes to the list of facts can be enacted [4]. In a rule-based expert system, the computer program uses rules - to represent
CHAPTER 2. LITERATURE REVIEW

conditions, to reach conclusions from a set of premises. However, rule-based expert systems are not procedural, but declarative programs. Unlike procedural programs, declarative programs use a runtime engine to make decisions about the scheduling and control-flow. Modern rule-based systems use a combination of procedural and declarative languages, to get the advantages of both the languages (ability to use knowledge in ways system designers didn't for see - declarative; and faster usage/define exact procedure to follow to solve a problem - procedural). The first step in the development of any rule-based system is to begin collecting the knowledge from which the rules will be derived. Knowledge can come from human experts, procedures manual, or other written documents [34]. Knowledge representation plays the following 5 roles [24]:

1. it is a substitute for the real data,
2. how should we think about the world/audience,
3. how/what the audience tends to reason intelligently/infer from the knowledge being represented,
4. medium of efficient computation, and
5. source of expression and communication in which audience tell the machine(and one another) about the world.

For human or computer agents to use any data/knowledge, knowledge engineering has to be performed on that data [25]. We will review knowledge engineering processes surrounding two key knowledge modeling techniques - ontology and production rules, the two representations that we employ in this research.

2.2.1 Ontology

An ontology is an explicit specification of a conceptualization to provide common understanding across domains. It is a set of concepts and relationships between the concepts [42]. Knowledge-based system can use such an organized network of concepts and relationships to uniquely capture the underlying knowledge in a shareable form. That is, the underlying knowledge of ontology is in an explicit and reusable form. Further, ontological knowledge along with problem-solving rules, support the construction and querying of knowledge in a knowledge-based system [25].
With Knowledge-based systems we come across three kinds of users: Knowledge engineers, domain experts, and end-users. To develop a Knowledge-base system, knowledge engineers provide structural concepts and build the Knowledge-Acquisition tool (KA-tool - an interface to acquire knowledge), domain experts use the KA-tool to build and edit specific knowledge bases, and end-users interact with the final expert system for decision support [37]. However, it is difficult and time-consuming to build knowledge bases with the introduction of a knowledge engineer between the domain expert and the knowledge base. Domain expert has to specify the exact requirements to the knowledge engineer, who will accordingly design the system. If this communication between domain expert and knowledge engineer is not clear and precise it could lead to errors and misunderstandings which would result in an incorrect knowledge-based system [37]. Tools such as Protege (an editor for building ontology) are being introduced to overcome these problems and to help domain experts build knowledge bases more directly.

In general, ontology offers classification of knowledge. Popular ontology classifications are - Top-Level ontology, Domain ontology, Application ontology, and Task ontology. The Domain ontology is more specific about what is in a Top-level ontology, which holds generic information about such things as time and space. The Application and Task ontologies can cooperate with each other and connect at some point to a Domain ontology. The connection of the three ontologies allows for better human and machine understanding of the problem space. The Task ontology usually drives the Application ontology [25]. For instance, in my research, the Interaction ontology would be the Task ontology that drives the Programmer style ontology (Application ontology) and SRL being a part of the Domain ontology.

The advantages of using ontologies are that they

1. Are specified in a standard, system-independent form and

2. Can be translated into specific representation languages

These two characteristics make them supported over multiple representation systems and portable across systems. Also, this helps ontologies to share knowledge across various systems to support similar understanding across overlapping domains. Preferably, ontologies are designed to be reused and easily extensible. Ontology editors allow for automatic ontology merging and alignment, and consistency checking [42].
2.2.2 Rules

Rule-based programs are quite common in the world of Artificial Intelligence and have been used in many applications. These applications range from simple mail filtering to complex tasks such as monitoring chemical plants, online tutoring system, security systems, diagnosing medical problems, and so on. Rules-based systems, if designed properly, have the ability to degrade gracefully (i.e., provide all the possible solutions) in the presence of incomplete information [46].

Rule-based programming is declarative. A purely declarative program, in contrast, describes what the computer should do, but omits much of the instructions on how to do it. Rules are a lot like the if-then statements of traditional programming languages. The if part of a rule written in this form is often called its left-hand side (often abbreviated LHS), predicate, or premises; and the then part is the right hand side (RHS), actions, or conclusions [34]. A rule-based system is a system that uses rules to derive conclusions from premises.

In order to process the rules, a rule engine is needed. It knows how to follow rules, without containing any specific knowledge itself. We have used the Jess rule engine for rule based programming in this research [1]. Jess is small, light, and one of the fastest rule engines available in public domain. Its powerful scripting language gives access to full Java API.

A typical rule engine contains: a working memory, a rule base, and an inference engine [34]. The working memory stores the data from the knowledge base in appropriate format for the rule engine to operate on. The working memory can contain all or partial information in the knowledge base. As far as Jess is concerned, all the pieces of information the rules work with are represented as facts in the working memory. In general, this collection of facts is known as the working memory.

A rule base is used to store all the rules the system deals with. They may simply be stored as strings of text, but most often a rule compiler processes them into some form that the inference engine can work with more efficiently. Jess's rule compiler builds a complex, indexed data structure called a Rete network to increase the efficiency [32]. A Rete network is a data structure that makes rule processing fast. In the Rete algorithm, the past test results are remembered across iterations of the rule loop. Only new facts are tested against any rule LHS, because, it is an empirical fact that, in most expert systems, much of the
CHAPTER 2. LITERATURE REVIEW

knowledge base is fairly fixed from one rule operation to the next. Although new facts arrive
and old ones are removed at all times, the percentage of facts that change per unit time is
generally fairly small [35].

The inference engine uses a pattern matcher, an agenda, and an execution engine to
perform its task. The inference engine decides on which rule to execute based on the
current working memory. Once it selects a rule to execute, it runs the rule which results
in updating the working memory with new facts. Inference engine follows a set of steps,
repeatedly. Firstly, the pattern matcher compares the LHS/premise of all the rules with
the current contents of the working memory to find all the activated rules. This collection
of activated rules is called the conflict set. Secondly, the conflict set is ordered to form an
agenda. Agenda includes list of rules that will be executed. With each piece of information
added to the working memory, a number of rules might be activated or deactivated. To
decide on the order of rule execution, rule engine uses a conflict resolution strategy. The
conflict resolution strategy for a given rule engine will depend on many factors, only some
of which will be under the programmer's control. Lastly, to complete the cycle, the first rule
on the agenda is executed (possibly changing the working memory) and the entire process
is repeated [34].

2.3 Human-Computer Interactions

Help systems have been extensively investigated for some time now. Houghton [50] reports
different types of early help systems that include "command and help assistance", "error
prompting", "online tutoring", "online documentation", and "help scripting". Online help
investigations also include the impact of display format [14, 45], animation [17], graphics
[105], and hypermedia [62] on help systems. Most contemporary software tools have generic
help facilities including metaphoric help (user-friendly interfaces) and online help (www
manuals). A few of them have context-specific help, as in Lumiere Project [49]. These
approaches, in an attempt to be self-sufficient, do not consider human-help as a resource
at all. Also, the help provided is generic and not specialized to the need of the helpee nor
is the help communicated properly. Let us look at the following techniques to improve the
existing help systems [86].
CHAPTER 2. LITERATURE REVIEW

2.3.1 Human-in-the-loop

Human help is inherently personalized, customized, and delivered exactly when needed [58]. The success of human helpers centers around their ability to understand contextual cues accurately and respond quickly. Having human helpers as part of a help system is termed as “Human-in-the-loop” approach [20, 41]. In this approach, a typical help scenario involves a user and the help system, where help is delivered through a dialog between the user and the help system [58]. Thus, human help complements machine help as long as they both share and agree upon a common context.

Contexts

Human help is superior to machine help as long as the helper is competent; and pertinent context is established between the person delivering help and the person receiving help. Kumar [58] explores techniques and interfaces to support the human helper who has been embedded in a human-computer help environment, where the design of the help system is capable of acquiring context information, making useful knowledge-based help responses, and ensuring delivery of help within acceptable time limits. Human-in-the-loop approach, aided by task-specific user-centric contexts, can assist the development of a pragmatic help system that is intelligent, informed of the user, tasks involved, collaborative interactions, and help resources. Successful peer help among friends and colleagues is due to the establishment of personal context. A context is a shared understanding of the help requirement. Establishing a suitable context is the heart of the problem in machine help. Most contemporary help systems are content-rich and context-poor. That is, a help request can be resolved using a variety of information and tools, hence content-rich, but it is a difficult problem to deliver the help in a personalized fashion target to the user’s needs, hence context-poor [58]. Contexts normally offer depth of information at the task level and at the meta level using which help offerings can be facilitated. Information contained within a context is, in most cases, localized. That is, the context information from one help session may not be relevant in another help session. However, this is disputable in case of Self-Regulated Learning, where the self-regulatory skills of a student observed within a session is quite applicable in another session participated by the same student. This is a reasonable assumption to make since self-regulatory skills are not transient in nature. Context information can be used during a help session in a variety of ways: to classify the style of a programmer, to
CHAPTER 2. LITERATURE REVIEW

identify compatible programming styles, to observe patterns of styles from a coherent group of programmers, to verify the suitability of a help resource, and so on. Essentially, the context is used to ensure the success of the three-way, possibly mixed-initiative, dialogue between the helper, the peer helper, and the help system [86].

2.3.2 Mixed-Initiative Interactions

Intelligent systems with the ability to support a mix of machine and human initiatives to address problems at hand are especially critical for applications of ambient intelligence - where solutions, support, recommendations, and warnings are offered typically in stream with ongoing activities [48]. There is a great opportunity for developing systems that understand how to work in a collaborative manner with users, where the system has skills in recognizing opportunities for problem solving and in understanding which aspects of problems the machine versus the person might best solve [48].

Mixed-initiative interactions attempt to model a middle-ground interaction strategy between Artificial Intelligence and Human-Computer Interactions where conversants (agents and users) contribute appropriate information when it is best suited and towards mutually negotiated goals. Also, depending on the needs of the student, the roles of a peer helper or the software system can be opportunistically negotiated [5]. These negotiations, based on a common contextual understanding, can determine which conversant has the control of the help conversation, at what point in time, and on what basis. A common contextual understanding should encapsulate the relative knowledge, preferences, and task goals of all the conversants in a help scenario.

One of the key characteristics of mixed-initiative systems focuses on the explicit representations for initiatives. At any one time, one conversant might have the initiative - controlling the interaction - while others work to assist it, contributing to the interaction as required. At other times, the roles can be reversed, or the conversants might be working independently, assisting each other only when specifically asked. Most help systems neither explicitly represent opportunities for initiatives nor regulate the interactions based on which conversant has the initiative. In a typical help-oriented interface, we find either an approach where agents instigate and control interactions through software mechanisms, or an approach where humans instigate and control interactions through direct manipulation. Mixed-Initiative refers to a flexible interaction strategy, where each agent can contribute to the task what it does
best [5]. Specific mechanisms of mixed-initiatives such as turn taking, grounding, confirmation, misrecognition repair, automation awareness, and attention management are used to interpret the conversants' objectives and establish a context. In our view, mixed-initiative interactions are driven by conversants' relative knowledge, preferences, and task-specific toward common, partially shared, and individual goals. Recently, there has been a newly found interest among researchers in combining automation with human values - “to seek valuable synergies between the two areas of investigation to avoid building complex reasoning machinery to patch fundamentally poor designs and metaphors to avoid limiting designs for human-computer interaction to direct manipulation when significant power and efficiencies can be gained with automated reasoning” [47].

Mixed-initiative interaction would also benefit by providing systems with the ability to infer subtleties of cognitive states of people so as to guide the “if and when” of interventions. Horvitz [47] presents a set of principles of mixed-initiative interaction and the value of harnessing decision-theoretic principles for guiding mixed initiative interaction under uncertainty. Work is progressing on mixed-initiative user interaction on multiple fronts, including efforts such as explorations into efficient interfaces and interactions for correcting recognition errors [94]. A number of advantages of using mixed-initiative interactions in problem-solving environments have also been reported, for example, bringing naturalness in communication, providing appropriate information at appropriate time [43, 44].

An ontology-oriented framework, called MI-EDNA, has been developed for mixed-initiative interactions in the domain of reading [91]. The framework consists of an ontology that represents information pertaining to content, learner, time, and interactions. Interactions of the conversants are automatically instantiated in the ontology. Further, patterns of specific tactics, strategies, and styles are recognized from the instantiated ontology. The recognized interactions, tactics, strategies, and styles, in an increasing order of granularity, are then mapped onto formal theoretical models (e.g., Zimmerman’s 3-phase model of Self-Regulated Learning [110]). The framework then advocates opportunities to disseminate ‘well founded’ prompts and other feedback mechanism to regulate the interactions among the conversants.

2.3.3 Feedback

To give appropriate help at opportune moment to the student one or more of the following feedback models can be used:
Engage the helpee with a pre-defined conversation model [31, 52, 75, 82]. That is, each utterance from a conversant should be interpreted and recognized within the scope of an interaction model. This allows us not only to trace interaction/mixed-initiative interactions in a theoretical framework but also verifies the validity of the theoretical foundation of the interaction model. We also contend that mixed-initiative interactions bring forth a sense of naturalness to the communication among the conversants that fosters healthy interactions among socially-oriented contexts such as ‘hallway chat’ or ‘homework’ interactions as depicted in [88]. Importantly, mixed-initiative interactions enable a more accurate conceptualization of the relation between conversant interactions and the underlying cognitive, meta-cognitive, and socio-cognitive strategies employed by the conversants.

- Introduce the helpee to ready, able, and willing human helpers [58, 10]
- Provide the helpee with timely clues and hints [69]
- Scaffold the helpee in a guided practice session [13, 63]
- Offer the helpee SRL (Self Regulated Learning)-specific feedback [89, 91]. SRL teaches the student how to learn better, monitor and evaluate oneself effectively. This technique helps to motivate the student, increase student’s efficiency, quality of product, and so on. The next section deals with this theory, its usefulness and challenges.

2.4 Self-Regulated Learning

The system that we propose in this research aims to inculcate the theory of Self-Regulated Learning (SRL) in its feedback mechanism. SRL views learning as an activity that students perform proactively, rather than as a covert event that happens to them in reaction to teaching [104].

In a classroom it has been observed that some students grasp concepts easily and are highly motivated to study, whereas others seem to have difficulty understanding the topic and are disinterested towards studying. This difference was attributed to the lack of metacognition, that is awareness of and knowledge about one’s own thinking, in the students lacking behind in studies [110]. It is an important task to teach students metacognitive skills to self-regulate, to improve their studying habits, to self-motivate, to improve their
efficiency, and so on. In this section we will discuss the characteristics of SRL and the two models of SRL being used in this research.

Students of all ages can be taught cognitive tactics and learning strategies that are used for self-regulating their learning [104, 109]. These SRL tactics and strategies are beneficial to learning in all domains in schools [80, 106]. However, it is very difficult to teach SRL in a classroom and thus not being effectively taught in classrooms [78, 79].

Winne suggests a novel solution, to learn SRL from everywhere, virtually from every learning-centric interactions, rather than just in classrooms, for example, learning from instruction and modeling by parents, teachers, coaches, and peers, learning from their prior experiences, observing others etc [104, 102]. However, there has to be extra effort from the students’ side to learn about SRL. It has been observed in Educational Psychology literature that students of same age have varying degrees of SRL capabilities [103]. This is one of the qualities that distinguish successful students from the others - successful students have and/or exhibit more SRL practice/experience than others.

Even though SRL is difficult to teach in the classroom, instructors should at least provide some triggers allowing the student to monitor one’s practice, plan and form goals appropriately, and use tactics and strategies that work for them. Students should be able to design valid individual learning experiments as part of their learning activities. For students to be able to do that, they should have extensive SRL practice and appropriate SRL-specific feedback. This feedback can be received from teachers, by comparing themselves with others, or by comparing themselves with their prior results (history) [104, 109].

2.4.1 SRL Characteristics and Components

Some of the key processes of SRL are goal setting, time management, learning strategies, self-evaluation, self-attributions, seeking help or information, and importantly self-motivational beliefs [110]. SRL is a technique that has to be learned and deliberately used initially. However, with experience, SRL becomes a non-complex, non-deliberate, non-meta-cognitive and automatic. SRL cannot be learned immediately, but can be acquired only by performing it regularly. Tactics are skills, and skills are acquired and honed through practice [104].

Monitoring is an important component of SRL [93]. Regardless of whether monitoring is planned or opportunistic, it produces information about the current state of a task and feedback, and that information is the basis for self-regulation [11, 12]. Planning is a deliberate activity that triggers monitoring activity of a task. Monitoring can also occur if
there is no explicit plan [102]. Such opportunistic monitoring is theorized to be the result of automated tactics that operate without deliberation and that trigger attention only when the information they monitor contravenes standards that are tacit [67]. Monitoring plays an important role in self-motivation. Students who have the capabilities to detect subtle progress in learning will increase their levels of self-satisfaction and their beliefs in their personal efficacy to perform at a high level of skill. Clearly, their motivation does not stem from the task itself, but rather from their use of self-regulatory processes, such as self-monitoring, and the effects of these processes on their self-beliefs [110].

Students have some beliefs about factors such as complexity, speed and reliability of tactics [90]. When deciding which tactics and strategies to use to reach the required goal, this prior belief is used by the student. The tactic/strategy that is easy to use is used rather than using another tactic/strategy that is difficult to perform but has much more beneficial results [104]. Thus, teachers or environment teaching SRL should advocate tactics that are easier and less cognitive, at least to introduce the process of SRL to students.

Planning and setting goals for oneself is another important task in SRL. Plans need to be modified depending on the output a student gets. As far as goals are concerned, students generate their own goals to adapt the assignment/task, rather than the assignment/tasks leading the students’ plan and goal generations [104, 30]. Thus, students always generate their own goals, some of which may perfectly match those assigned and others of which will not.

Self-regulation of learning is not a single personal trait that individual students either possess or lack. Instead, it involves the selective use of specific processes that must be personally adapted to each learning task. Studies have been conducted to show the effectiveness of SRL strategies. Students who set goals for themselves displayed superior achievement. A simple task of self-recording some aspect of their learning, such as the completion of assignments, recording the tasks completion and time taken to complete each task to reach the goal, often led to “spontaneous” improvements in functioning [93]. Thus, self-awareness would lead to enhancing self-control that would in return produce motivation for a personal change [109].

2.4.2 Self-Regulated Learning model

Let us look at the two models - one is proposed by Winne and the other by Zimmerman. A combination of these models (with more emphasis on Zimmerman’s model) is used in this
research to introduce students/learners about Self-Regulated Learning. Both these models treat SRL as recursive process.

According to Winne's SRL model [104],

- A task is broken down into a set of goals. These goal completions would help to achieve the task requirements. These goals are formed based on the students' motivation, their prior knowledge (belief) about the domain, and strategy knowledge.
- Students then select tactics and strategies to accomplish each goal.
- Next, they or external source (teacher, peer, software tools etc) compare students' performance, monitor their strategies' efficiency, monitor the final product with respect to achieving the requirements of the task, and monitor others factors such as time.
- This feedback (internal - generated by the student himself, and external - generated by external factors) is fed back to the goal setting stage.
- And the cycle continues from the beginning (i.e., resetting of the goals) till the task requirements are reached.

The next SRL model was suggested by Zimmerman [110]. It breaks down the SRL into 3 phases - Forethought phase, Performance phase and Self-Regulation phase.

- There are two major classes of forethought phase processes: task analysis and self-motivation. Task analysis involves goal setting and strategic planning. Self-motivation arises from students' belief about learning, i.e., students' beliefs about having the capability to solve the problem, motivation to solve the problem, and so on.
- Performance phase processes include self-control and self-observation/monitoring. Self-control refers to the deployment of specific methods or strategies that were selected during the forethought phase. Self-observation refers to self-recording personal events. For example, students are often asked to self-record their time use to make them aware of how much time they spend studying.
- Lastly, the self-reflection phase includes: self-judgment and self-reaction. Self-evaluation is a type of self-judgment that compares performance of oneself against some standard, such as one's prior performance, another person's performance, or an absolute standard of performance. Another form of self-judgment involves causal attribution, which
refers to reasons about the cause of one's errors or successes. Attributing the failure to one's strategy rather than to one's ability can be more motivational as it implies that using another learning strategy might achieve success. One form of self-reaction involves feelings of self-satisfaction and positive affect regarding one's performance. Increases in self-satisfaction enhance motivation, whereas decreases in self-satisfaction undermine further efforts to learn. There can be defensive and adaptive self-reactions too, where a learner might drop the task/course in order to protect his image from being damaged (defensive); or he might discard or modify his learning strategy (adaptive). The self-regulation performed by novices is very distinctive from that of experts. Novice has limited experience in SRL, thus they end up comparing themselves with the performance of the others. However, this is not a correct method, because others are on a learning curve too and might have different goals. However, experts evaluate their performance against personal goals that leads to greater self-satisfaction and motivation.

The system developed for this research embeds an ontological representation of Zimmerman's SRL model (along with some inspiration from Winne's SRL model), which includes the phases of forethought (planning, task analysis, self-motivation, goal setting), performance (self-control and self-observation), and self-reflection (self-judgment, self-reaction, self-evaluation, self-satisfaction). Zimmerman's SRL model is a higher level of abstraction of Winne's SRL model. Students of all ages can be taught cognitive tactics and learning strategies that are used for self-regulating their learning. These Self-Regulated Learning (SRL) tactics and strategies are beneficial to learning in all domains [80, 106], including the programming domain. The ontological SRL model will enable us to teach students to program better using SRL tactics and strategies.

2.5 Intelligent Tutoring Systems

This section reviews key issues in Intelligent Tutoring Systems, a research area in Computer Science that integrates observed student interactions with system, the task requirements, and pedagogy in order to provide a targeted learning experience to students.

There are many challenges to build learning support systems, also known as intelligent tutoring systems, that are capable of building an appropriate context for the user, share it with an appropriate helper, include a human helper when necessary, initiate feedback
(active feedback) to the user, communicate with the user at appropriate time, try to predict user intentions and preferences by observing the user, to build appropriate user models, giving feedback to encourage the usage of SRL tactics, and use mixed-initiative interaction appropriately. Intelligent tutoring centers around the notion of providing personalized learning experiences to students in light of their capabilities. Systems that perform machine learning, inference, and decision making fall under two major categories - preference and intention systems. Preference systems are those that employ rules to predict the preference of the users based on the context, prior knowledge, prior conversations, and demographics. Intension systems are those that try to predict the user's activities and goals (current as well as future). An example of the intension machine is the web search engine that tries to reason about the users goals based on limited search keywords [48].

Intelligent Tutoring Systems, in most cases, require student-system interactions to infer information about student's subject knowledge and to identify the most effective pedagogical strategy for the given learning situation. The next section outlines some of the challenges and opportunities in building such intelligent, interactive, mixed-initiative, learning systems.

2.5.1 Challenges and Opportunities

Eric Horvitz [48] lists the various challenges and opportunities present while building predictive and intentions systems. Here are some of the claims and examples as summarized by Horvitz. When predictive and intention systems make decisions, they take various factors into consideration before executing the action. Some of the factors are:

- bother to the user (when to interrupt the user),
- if wrong prediction is executed, what are the consequences,
- cost of eliminating the wrong actions, and
- time saved by taking those decisions.

For example, the Superfetch predictive component within Microsoft's Windows operating system learns by watching sequences of application launches over time to predict a computer user's application launches. These predictions, coupled with a utility model that captures preferences about the cost of waiting, are used to determine whether or not to prefetch the application. This Superfetch service helps to minimize the average wait time for applications to be ready to use after the user launches them.
CHAPTER 2. LITERATURE REVIEW

Another example would be LookOut; an interface for calendar scheduling that automatically extracts email information and updates a person’s calendar. It establishes a context around beliefs about a user’s goals. In addition, Lookout also allows the user to directly manipulate the calendar. To collect cases, Lookout runs in the background and notes when users examine an email message and then turn, within a time horizon, to a calendar view or scheduling task. Lookout also uses in-stream supervision to learn about the ideal timing of actions. Thus, enabling Lookout to courteously withhold potentially distracting engagements while a user reviews a message.

One of the essential features of mixed-initiative systems is to take initiative. The system or the user can take initiative and interrupt the other. However, when the system takes initiatives to give active feedback to the user, it should be careful enough to not interrupt the user frequently. The system should be programmed to give accurate and correct feedback and at appropriate times. It should take into consideration the user’s workload, the user’s focus of attention, and so on; before giving the feedback [48].

These machines should also be able to predict surprise detection and surprise forecasting. That is, these systems should be able to detect and alert the user when current or future events that they might divulge in, might surprise them. Thus, the system should be able to detect a variation between the expected outcome and actual outcome, for which the system will have to be capable to predict the users’ next few actions, know of the users’ history and context, and to be able to calculate the users’ expected outcome and the actual outcome. Such mixed-initiative intelligent tutoring systems might want to present students with various other options to reach the desired goals, than following a pathway that would surprise them.

Trust is another important factor that these predictive and intension machines have to deal with. Typically, users find it difficult to trust the decisions made by the system, hence they avoid seriously considering suggestions of the system. One of the method to increase the trust is to let the user be involved in supervision, when the user model is created along with the decision making libraries. Another method is for the system to provide explanation for its suggestion, whenever the user asks for it.

Prediction and intention systems tend to store user information in the user model. There is a concern about the privacy of this data collected about the user. To overcome this problem, the user model can be stored on the user’s local computer, or this personal data could be made anonymous in order to share it with others.
Overcoming these challenges would result in the humans trusting the decisions and feedback from the machine.

2.5.2 Model-Tracing and Constraint-Based Models

There are two novel ways to build a contemporary intelligent tutoring system: model-tracing (MT) and constraint-based paradigm (CBP) [54]. A brief comparison of these two methods is presented below:

- “MT and CB Models are based on fundamentally different assumptions about tutoring. MT technique (MTT) is a process-centric approach wherein the tutor tries to infer the process by which a student arrived at a solution. CBT, on the other hand, can be considered to be product-centric in that remediation is based solely on the solution state that the student arrived at, irrespective of the steps that the student took to get there.”

- Also, as far as feedback/remediation for Model Tracing Technique is concerned, it is based on how student arrived to the solution. Whereas in Constraint Based Modeling Technique the feedback/remediation is based on the message associated with the violated constraints with respect to the final product.

- Importantly, the effort required to develop a model-tracing tutor is much more than building a constraint-based tutor.

A number of studies have assessed the impact on student performance of Model Tracing Techniques (MTT). The general results of these studies indicate at least one standard deviation improvement in student performance with respect to the performance of students in normal classroom instruction [54].

A Model-Tracing Tutor consists of expert rules, buggy rules, a model tracer and a user interface. Expert rules model the steps that a proficient individual might take to solve the problem in question. Experts can take different strategies to solve the problem. Thus, there can be multiple expert rule paths for a single solution. Buggy rules being activated while tracing a students model, indicate misconception or erroneous path being followed by the student. The tutor then provides feedback/remediation associated with the buggy rule. According to Kodaganallur et al. [54], “the crux of an MTT is to “trace” the student’s input, where tracing consists of finding a sequence of rule executions whose final result
CHAPTER 2. LITERATURE REVIEW

matches the student's input. If the tutor is able to do this, it is taken to mean that the tutor has understood the process by which the student arrived at an answer”.

As far as CBM is concerned [54], “The central construct in CBM is that of a constraint. A constraint specifies certain conditions that must be satisfied by all correct solutions. When a student’s work violates a constraint, we gain specific information about the student’s mental model. The paradigm does not consider it important to know how the student arrived at a specific problem state; what is important is simply the set of violated constraints. If any constraints are violated the tutor provides remediation based on messages associated with each violated constraint. When a problem state does not violate any constraint, the tutor simply lets the student proceed.”.

Since we are interested in correcting students programming style, we need to know how students program rather than just knowing whether the solution is efficient. Thus, we follow the process-centric model tracing technique.

Most of the models in Carnegie Mellon Universities Human-Computer Interaction Institute are based on model tracing technique using ACT-R human cognition theory model. ACT-R is a cognitive architecture: a theory about how human cognition works. These assumptions are based on numerous facts derived from psychology experiments [99]. Based on the tasks, ACT-R models are created by using not only ACT-R’s view of cognition but also adding their own assumptions about the particular task. These assumptions can be tested by comparing the results of the model with the results of people doing the same tasks [99]. Some of the examples using these models are:

- ALPS: Active Learning and Problem Solving tutor [22]: “The Active Learning and Problem Solving (ALPS) project involves constructing and evaluating educational technology that emulates human tutors by integrating a state-of-the-art educational technology called Cognitive Tutors with an innovative interactive questioning environment called Synthetic Interviews to produce an active learning environment that rivals the effectiveness of human tutors.”

- PACT: Pittsburgh Advanced Cognitive Tutor Center [21]: “The PACT Center uses cognitive tutor technology to create an integrated classroom and computer lab curriculum that supports students’ understanding of mathematical and real world concepts. Based on a computational model of thought, cognitive tutors can automatically generate the most sensible solutions to any given problem, follow students step-by-step
as they work, and provide individualized feedback and advice."

- CTAT: Cognitive Tutor Authoring Tools [55]: "Cognitive Tutors have been successful in raising students' math test scores in high-school and middle-school classrooms, but their development requires considerable time and expertise. We are developing a set of authoring tools to make modeling both easier for experts and possible for novices in cognitive science. The tools draw on ideas of programming by demonstration, structured editing, and others. Careful application of HCI methods is key."

As we can see from the above examples, the goal of ACT-R models is to replace human-help and try to predict users' next step. However, as far as our approach is concerned, we plan to develop models that will try to provide context to the human helpers rather than completely trying to replace them. Also, personalized and customized feedback is given not only by the instructors but also the system.

Model Tracing MICE system that we propose tracks the user interactions and builds a programmer model customized for the user. A set of rules can be applied to the interaction student model to detect a programmer's programming style. These rules can be developed based on the programming style detected by the current version of MICE. These rules and interactions help update user's programming style and user's SRL model. Now, feedback can be based on rules being violated depending on how the interaction, programming style, and SRL model are updated.

The interaction and programmer style model become our task-level models that track interactions and give feedback based on the domain ie they are domain specific. However, these interactions are mapped to a theory of SRL. This SRL theory is not domain specific and it is our meta-level model.

Similar to the ACT-R model, we are trying to track people's work at task-level, provide feedback at task-level, map task-level interactions to meta-level, and provide feedback at meta-level. They both deal with cognition; thinking processes of humans. However, the meta-level in ACT-R is based on using human cognition theory, which is a broad category, difficult to model and easily misinterpretable. However, we propose to use meta-level tracking based on SRL theory, which is a rather stable, long-term, and easily classifiable theory, with capabilities for direct mapping of real-world interactions to the theory irrespective of the domain. For example, we can have different SRL models for a student for the reading domain, writing domain, and problem-solving domain. These different SRL models of a
single student can be compared to find why he performs better in one domain as compared to other domains. Or, we can have a common SRL model for a single student where the interactions of all domains are mapped to a single SRL model.

In general, model-tracing systems from Carnegie Mellon University exhibit dynamisms that are quite constrained and work only within well-defined task domains. That is, the interpretation of a sequence of learner interactions in one system could be very different from an interpretation of the same set of interactions by another system, assuming first of all that both systems have valid 1-to-1 mappings for each other's metadata tags.

2.6 Stages of Programming

Programmers go through a variety of stages when they design, develop, test, and integrate code. Some of these stages lie well within the realms of Software Engineering.

IEEE defines Software Engineering as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. Software Engineering involves people, process, project, and product [9]. Software Engineering's main focus is to increase the quality of the product for which a set of process and methods are to be followed. Designing an elegant program for the wrong requirements is useless. There are tools available to guide students through engineering processes to develop software specific to the design requirements. While Software Engineering deals with all aspects of software lifecycle, code engineering deals mostly with how an individual programmer designs, develops, and maintains code.

The spiral model of Software Engineering classifies software processes into 5 stages (stages of programming). They are - communication, planning, modeling, construction, and deployment phase. It is an iterative model, thus the 5 stages are repeated sequentially. The iterations are continued until the final product matches the requirements of the end-users [81]. Rather than focusing on the software, one can model how an individual member of the software team contributes to the Spiral Model of software development. Also, within each stage of the Spiral Model, one can interpret a programmer's activities with respect to Zimmerman's SRL model. That is, the self-regulatory abilities of a programmer can be traced throughout the 5 stages of the Spiral Model.

Pressman [81] discusses various issues in software development and maintenance. In the following list, we present Pressman's arguments with our comments on how model-tracing
software systems can augment these issues.

- A company/manager/supervisor should not just educate the developers' software engineering standards, but also ensure that developers actually implement and practice them. This is quite possible only if the project manager has concrete notions about the developer's coding styles.

- Adding more programmers to meet a deadline does not help to hasten the software development process because new programmers will require training time. Also, the existing programmers in the project will have to devote their software development time for training these new programmers. Adding new programmers will still help the project meet its deadlines, as long as the project manager has quantitative notions about training requirements of individual programmers, which can be traced from observing individual's coding styles.

- Outsourcing a project, doesn't imply one can relax and forget about the project. Continuously monitoring the progress by the outsourcers and giving them feedback is essential. Continuous monitoring at real-time and offering feedback at real-time is possible in model-tracing systems.

- To determine the quality of the program, one need not wait for the program to be running. The formal technical reviews are proved to be more effective than testing for finding certain classes of software errors. It is quite possible to develop quality metrics that use both formal reviews and test results, which is possible if there is an integrated model of software development.

- Software engineering is not to be looked as a process to create voluminous and unnecessary documentation that will invariably slow the code development process. In fact, following the software engineering process increases the quality that leads to reduced reworks on the software. And reduced reworks on the software results in faster delivery times. If we trace the code development process in a formal framework, it would be convenient to figure out such loopholes.

Thus, model-tracing can contribute enormously to software engineering in general, and in particular to Personal Software Process (PSP).
CHAPTER 2. LITERATURE REVIEW

2.7 Summary

The inspiration for this research originates with Helper's Assistant [58], a support tool for helpers in the domain of Java programming, and I-Help [41, 40], a tool that offers peer help. Both the systems employ the human-in-the-loop approach, where human helpers can augment system-generated help. As an extension to this approach, my work focuses on modeling the self-regulatory skills of programmers and enabling the system to provide theory-centric, formative, non-intrusive feedback.

Our main goal is to help Java programmers and enhance their programming skills. The literature review supports our belief that programmers need more help concerning their programming style processes. A strong programming style would lead the pathway to efficient and less error-prone programs. The focus of our work is to detect coding styles of programmers and suggest modifications to their programming styles based on the theory of Self-Regulated Learning.

We developed a system to assist programmers to regulate their development style using an Integrated Development Environment (IDE) - BlueJ. The objective of MICE, abbreviated for Mixed-Initiative Coding Environment, system developed for this research is to capture various interaction of the programmer and use this interaction data to determine the development style of the programmer and the amount of SRL being used, and give feedback to improve their programming/development style.

Based on the review, the following major tasks have been identified for inclusion in the MICE system: 1) capture of code development styles of the programmer; 2) engage programmers in Mixed-Initiative (MI) interactions; 3) inculcate the theory of Self-Regulated Learning (SRL) in the feedback mechanism; 4) allow students to engage MICE at real time. The next section discusses how these features have been designed and implemented in the MICE system.
Chapter 3

MICE Architecture

This section presents the architecture of MICE under two categories: functionality of the system and technical requirements of the system.

3.1 The Functional Architecture of MICE

The functional architecture describes the flow of data and flow of control in the system. The flow starts with programmer interactions in an IDE. The current MICE software uses BlueJ as the IDE with research-specific extensions. These interactions trigger events in the IDE to instantiate appropriate elements in the programmer’s interaction ontology. Current implementation only tracks events with respect to ‘compilation of code’. That is, with every instance of execution of the compile command within the IDE, the ontology is updated with information concerning the code, the bugs, and the programmer’s self-regulatory behaviour. Specifically, across multiple compiles, MICE keeps track of the compiled filenames, the number of bytes of the compiled files, the types of errors and warnings observed under each compile command, and the line numbers of the errors and warnings that identify the programmer’s code construction and debugging patterns. Each session of the programmer with the programming IDE is stored in an interaction ontology. On start of a new session between the programmer and the IDE, a new interaction ontology is instantiated for the programmer. All the interaction ontologies of a programmer are stored across multiple sessions (Interaction Ontology Log) for future reference.

The MICE Programmer Style Viewer tool is a part of the MICE system that enables
students to view their coding style at any point in time. The style viewer presents programmers with a set of 10 graphs that enable them to reflect their own coding styles as well as compare their coding styles with the styles of a particular other student. The graphs are accompanied with information to help students interpret the graphs. These graphs are explained in Section 5 Results and Analysis. The functional architecture of MICE is presented in Figure 3.1. The dotted line in Figure 3.1 bifurcates the system into external system integrated with MICE (part above the dotted line) and MICE system (part below the dotted line).

For the sake of the experiment, MICE checks whether students have started to compile and test code after a predetermined time period (currently set to 10 minutes). If a student exceeds this limit, then MICE pops a message to state that the time limit has been violated. This shows that MICE is capable of intervening with respect to any combination of
obsorable events in the IDE and pedagogical requirements of the task. Further, MICE Programmer Style Viewer is popped-up at specific time intervals (presently, 15 and 20 minutes) if the programmer has not activated this viewer. In the last 5 minutes (similarly, this time set is susceptible to change accordingly) of the duration of the experiment, programmers are asked to check the quality of their code (to make the input output user-friendly), to reduce the complexity of the code by removing the unused code (using PMD tool or complexity evaluator tool CEB), and to check whether the input/output requirements have been met (by compiling and executing the code and checking whether the goals they set had been met) [these include the MICE initiated feedback for the programmer as shown in Figure 3.1]. These interactions are more proactive in nature and hence can be classified as interactions initiated by the software system.

The Programmer Style Viewer can also be activated actively by the programmer. Instructors can set up time constraints under which students can learn various stages of coding processes.

3.2 The Technical Architecture of MICE

MICE presently uses the BlueJ IDE as the coding environment. Programmer interactions within BlueJ are tracked using MICE. MICE listens to BlueJ events such as compilation, clicking on a menu item, moving the mouse cursor, usage of the tools available in BlueJ and so on. Once an event is triggered, Jena, a Semantic Web toolkit, instantiates or updates the Interaction Ontology at run-time. For example, once a compile event is triggered, Jena updates the programmer's Interaction Ontology by creating an instance in the compile class with the following information: time of compilation, number of bytes of the code compiled, number of errors and warnings present, line number of errors, and name of program file compiled. We also store these interactions in text files as backup. A sample scenario is presented in Figure 3.2, where interactions are stored in an ontology over a period of time and graph is being constructed by MICE (which is presented to the programmer when he request for it).

\[ \text{http://jena.sourceforge.net} \]
CHAPTER 3. MICE ARCHITECTURE

3.3 MICE Ontology

Presently, the Interaction ontology keeps track only of the interactions observed between the programmer and the IDE, with respect to each compile event. However, the interaction ontology can be extended to include all interactions of the programmer. The next section will present the structure of the entire interaction ontology [85, 83].

3.3.1 Interaction Ontology

As programmers start to code, most of their interactions with the BlueJ IDE are recorded and represented as instances within the Interaction Ontology. Events that are presently
tracked by MICE and recorded in the interaction ontology are:

1. Compile Event - For every compile event, the Compile instance is created that contains the compile ID, timestamp of the compile, names of files that were compiled, size of files, errors and warnings encountered at compile-time, and line numbers of errors.

2. Run Event - Similar to the compile event, when a programmer executes the code, a run event creates an instance of the Run class with the run ID, timestamp of the execution, name of the executable file, size of the file, errors and warnings produced at run-time, line numbers of the errors and warnings, and the output produced.

3. Added/Deleted Event - This event is triggered whenever there is a change in the source code with respect to adding or deleting or modifying language constructs. It creates instance of what was added/deleted and modified, at what time these changes happened, on what line numbers they happened, and how this change reflects on the rest of the data.

In order to extend this interaction ontology further, MICE can collaborate with other systems and receive a number of events from external applications such as gStudy, iHelp or Internet browsers. For example, gStudy can be modified to record the following events to be distributed when a programmer attempts to read a programming task, refers to various online resources, comprehends the task in terms of a code design, and chats with a fellow programmer to validate the design. Events that can be recorded as a result of these added applications are:

4. Link Event - This event is triggered when a programmer creates a link between the two sections of content. The content could be a description of the programming task of any multimedia (text, graphics, audio, and video) resource that the programmer is referring to. This event triggers and instantiates data such as link type, link created from, link created to, and so on.

5. Highlight Event - This event is triggered when the programmer attempts to highlight any portion of the content he is reading. Further, the programmer can attach qualitative clues/tags such as 'important', 'doubt', 'to discuss', and so on.

6. Browse Event - This event records the links that the programmer has followed while performing the coding task.
7. Search Event - This event records when the programmer engages in a search activity within a custom-built search tool.

8. Chat Event - This event is recorded when the programmer chats with another programmer from within gStudy’s gChat tool. The gChat tool also enables the programmers to use pre-built, semi-constructed, or freely formulated queries. The event records who chatted with who, when, for how long, on what content, using what qualitative queries, sent what responses, and so on.

9. Posting Event - This event is triggered when the programmer posts an article to the custom-built discussion board or visits the discussion board to read and respond to others’ postings. This event records the type of participation by the programmer, the time of the participation, and so on.

In summary, the Interaction Ontology is a model of events observed in the interaction environment. They are mostly normative in nature - i.e., they are captured by simple observations within the IDE environment. It is quite possible to generate simple, pedagogical inferences based on these observed data. The next section discusses the types of feedback facilitated by MICE.

3.4 Feedback

As part of their education, Computer Science students develop hundreds of computer programs and receive only summative feedback on the end results of their program designs and code. MICE proposes a formative feedback mixed-initiative interface that uses compilation-time code segments to automatically capture interactions and encourages the interpretation of these interactions by the programmer by displaying their programming style and providing comparative graphs to compare their own programming style with experienced and inexperienced programmers (using MICE Programming Style Viewer).

The very premise of MICE is that, MICE can be expanded to allow programmer to reflect on their programming style which can be captured irrespective of the programming language or programming IDE they use (not just limited to Java programming language and BlueJ IDE). MICE works on capturing events generated by the programmer, for example, compile event. These events are generic and not specific to a particular programming language. Also, if the IDE allows MICE to track these events, we can expand MICE to any IDE.
Thus, programming style would be a reflection of not just one particular programming language, but a programmer's programming style irrespective of the programming languages or IDE used by the programmer to program.

MICE's feedback can be passive (programmer-initiated query) or active (system-initiated suggestions to the programmer without the programmer asking for it). Presently, MICE implements two types of active feedback. Firstly, when the programmer exceeds certain time constraints set by the instructor. Secondly, when the programmer ignores critical reflection opportunities on coding styles (for example, the current implementation of MICE keeps track of whether the programmer used the MICE Programmer Style Viewer tool. If programmer does not actively use MICE Programmer Style Viewer tool to view his own programming style and reflect on it, then MICE will display these graphs to remind a programmer to reflect on his programming style. However, the programmer is not forced to reflect on his style and can choose to close the MICE Programmer Style Viewer when it's popped to the programmer). In both cases, MICE can proactively interact with the student with appropriate feedback.

MICE also encourages the use of other techniques and tools that promote self-reflection not only on programmer's programming styles but also on the quality of the programmer's code. There are several tools embedded within BlueJ that help programmer perform software engineering techniques automatically rather than performing them manually. Since all these tools are embedded within the IDE and available at a click to the programmer, they are easily accessible. MICE reminds the programmer to use appropriate tools at appropriate time to improve their code, for example, when the last 5 minutes were left for the programmer to complete their Java problem, MICE reminded the programmer about the CEB complexity evaluator tool that helps them evaluate the complexity of the code or use PMD tool to remove the unused parts in their code or commenting the code appropriately to make it understandable.

Importantly, MICE system and its feedback are not intrusive. MICE system is not intrusive because tracking of coding style happens in the background, surreptitiously. Programmer continues to perform his normal coding activities at the task level and MICE tracks the interactions in the background without burdening the programmer with any extra work. However, as far as MICE feedback is concerned, MICE does take initiative and provide feedback to remind programmer to reflect by displaying their programming style and the tools available in IDE to help them with their programming tasks at that instant.
MICE feedback may be considered as intrusive, however, the intrusion if present is minimal. Programmers may or may not accept MICE's feedback. Afterwards, when programmers engage in meta-level activities such as reflection, MICE offers meta-level, theory-centric support.
Chapter 4

Research Design

This thesis investigates the following research questions:

1. Do student programmers have “identifiable” programming styles?

2. Do student programmers “exhibit” self-regulatory abilities and are they inclined to use self-regulatory tools embedded in the IDE, even when these tools are only available passively?

3. Do intermediate student programmers exhibit more self-regulatory abilities than novice student programmers?

4. Are novice programmer’s faster learners of SRL principles than the intermediate programmers? Can SRL be taught progressively?

5. Are programmers open enough to change their programming style and does this change lead to better programs?

6. Does student programmer’s data provide appropriate context for the teachers to identify their programming styles and give feedback to students to improve upon their programming styles identified in this manner? Does this data offer help to instructors to provide more customized and personalized help to the student? Will instructors take appropriate measures to encourage usage of SRL in their courses? Do teachers trust the data collected by the system?

To answer these research questions, we conducted two independent experiments - one dealt with the Java programming aspects corresponding to the first 5 research questions, and the
other dealt with how instructors perceived MICE's formative, theory-centric approach (the sixth research question).

### 4.1 Framework of the Method

An experiment was conducted targeting the first 5 research questions with java programmers as participants. The participants were each given a short demonstration about the MICE system - how to use it, how to interpret the data, why to use it, various features of MICE, and how the BlueJ IDE (Integrated Development Environment) works. Participants were allowed to practice for 5 minutes. Then they were each given 2 different Java problems. There were 2 sets of Java problems - one for novice programmer and one for intermediate programmer [Appendix D]. Programmers with Java programming experience of less than one year were treated as Novice programmers else they were classified as Intermediate programmers. Participants were given 30 minutes to solve each Java problem. After each Java program, they were asked to fill a questionnaire [Appendix E]. The questionnaire included questions about their experience with the system, ease of adapting to it, usefulness of MICE's feedback (with respect to MICE helping programmers identifying their programming style, identifying their weak and strong programming styles, identifying concepts in Java they have troubles with, and localizing their bug in the program), and the various tools and techniques they employed during coding.

Participants were allowed to use any of the tools provided by BlueJ and the added extensions to BlueJ, use any books for reference, use the Internet, or cross-reference other code in BlueJ examples or any other code. They were provided with sheets of paper to perform any rough work. The program code submitted by each participant was stored along with interactions tracked by MICE. MICE extracted data from student-system interactions generated graphs depicting their coding styles as and when participants asked for it passively or actively when MICE was trigged by built-in instructions. At any point in time programmers can view their style of coding along with areas of concern in their development style. The questionnaire data along with the interactions stored by MICE in the ontology, and the pieces of code developed by students formed to answer our research questions.

The second experiment was conducted targeting the sixth and final research question with programming instructors. It started with a discussion with the instructors. They were given a demonstration of the MICE system. Instructors were shown how data was collected,
how student data was presented, how to interpret the data to give personalized feedback to the students, and how to change their teaching strategy according to the data. The data collected from the first experiment (with Java Programmers) was analyzed and presented to the teachers/TAs to give them a glimpse of the functionalities of MICE. Further, the participants were also given an introduction to the process of tuning MICE to provide appropriate feedback to students based on the real data collected in the first experiment. After this demonstration, teachers and TAs were asked to fill a questionnaire [Appendix F] that asked them to evaluate MICE with respect to identifying the student patterns, identifying student programmers' weak and strong programming habits, identifying Java concepts student programmer might have difficulties with, helping them modify their teaching strategy/style/curriculum, their confidence in the data that MICE presents, and finally the way MICE represents and presents the data. They were asked to evaluate MICE on a scale of 1 to 5 - 1 being the least preferred value and 5 being the most preferred value.

4.2 Data Collection and Management

This section presents the methodology we followed to collect data and how we maintained and protected the ethical considerations, integrity, and sensitivity of the data.

4.2.1 Sample

The sample for the first experiment consisted of undergraduate and graduate students at Simon Fraser University. Participants with varied experience in programming were employed to capture the different programming styles and strategies used while programming, their reaction towards feedback and improvement in their code. The experiment consisted of 23 participants ($N = 23$) whose Java programming skill varied from 4 months to 5 years. There were 10 novice and 13 intermediate programmers.

In the second experiment, the sample consisted of teachers and TAs from Simon Fraser University who had taught programming courses in the past or were currently teaching them. Five instructors and five teaching assistants took part in the experiment.
4.2.2 Data Management

The MICE system uses ontologies for data storage of real-time interactions. Instances of the desired interactions are created dynamically and stored in the programmer’s interaction ontology. While a user is programming, interaction data is also logged in a text file to supplement the results of the data in the ontology. The final program code submitted by the participants is also stored (this is done because MICE gives feedback not only with respect to their coding process i.e. formative feedback, but also summative feedback). This submitted code determines the completion, complexity and quality of the program. The questionnaire and the sheets of paper provided are also collected from the participants. This helps to detect additional interactions that MICE cannot capture, such as drawn flowcharts, written algorithms, or debugging notes on paper.

The questionnaires from the teachers and TAs are also collected to determine their confidence in the data presented by MICE and their willingness to consider MICE’s view of their teaching strategy. It also helps us to evaluate the data representation style used by MICE to represent programmer’s development styles. Suggestions from teachers and TAs are noted down to improve the readability and understandability for the future versions of MICE.

4.3 Measurement of Variables

In the experiment, the independent condition was the compilation of the code in BlueJ. The dependent variables measured were “total number of bytes”, “number of errors”, “error type”, “line numbers of error”, “files compiled” and “time between compiles”. Complexity determination tool, quality enhancing tools, referencing other files, usage of MICE and other tools being used to facilitate coding are also being tracked. This helps to track their programming style, usage of SRL (which is measured by mapping programming interactions to Zimmerman’s SRL model using the Software Engineering theory), and their acceptance towards MICE feedback.

4.3.1 Research Instruments

For conducting the experiment, a computer that had MICE and BlueJ set-up along with all tools (for example, using Sequence Diagram Editor and building test cases tool for planning;
or using Jeliot graphical debugger for the debugging and testing of their code; or using MICE Programmer Style Viewer tool for reflecting on their programming style; or using CEB complexity evaluator, PMD tool to improve the readability of the code and making it less complex) to help programmers self-regulate their tasks was used. The computer was connected to Internet all the time. Participants were allowed to use the Internet whenever they want. Various Java books were accessible for reference. Some sample programs were made available for cross-referencing.

4.4 Data Analysis

The data used by MICE’s Programming Style Viewer was collected only when participants compiled their code. That is, interactions are captured only when participants compiled their code. For each compile, MICE keeps track of the names of compiled files, the number of bytes of the files that were compiled, the time at which the files were compiled, the types of errors and warnings that occurred, and the line numbers of errors and warnings.

MICE captures the data dynamically and displays it whenever the programmer requests it. We have used graphs for displaying programmer’s styles along with appropriate highlighting and explanatory text to infer the graph and to detect the programmer’s problematic areas (Graphs are used to display the programmer style because they provide visualization of programming style that is easier to interpret and compare with other experienced and inexperienced programmers). MICE takes initiative and provides feedback to remind programmers to reflect on their coding styles (recommendation is based on the elapsed time).

Data about programmers’ styles can also be collected from the tools/strategies they use to offload their cognitive process (the different programming styles captured will be discussed in Chapter 5 on Result and Analysis). Some of these interactions are tracked by MICE whereas others are tracked by the questionnaire submitted by the participants. MICE also tracks the time when the BlueJ IDE was started. It also tracks the start and end time when the MICE Programmer Style Viewer was opened and closed actively by programmer or passively by MICE (tracing this interaction helps to know whether the programmer actively made effort to use the tool or do programmers have to be reminded about using the tool. Also, we track start and end time of the MICE Programmer Style Viewer tool usage, to know whether programmer accepted the feedback and reflected on their programming style; or did they reject the feedback by closing the MICE Programmer Style Viewer tool as soon
as it was popped to the programmer). Questionnaire filled by the student participants after each program keeps track of programmer's interactions such as using Internet, books, cross-referencing, usage of various tools embedded within BlueJ - UML representation, Sequence Diagram editor, Test Cases, Jeliot graphical debugger, CEB Complexity Evaluator, and PMD unused code remover. Participants were given empty sheets of paper that they could use, which was collected at the end of the experiment to track whether they have written algorithms, drawn flowcharts for planning, tried debugging on paper, or build test cases on paper.

By looking at the code submitted by participants we can see whether their code is readable, has used proper naming convention, uses comments, and indents properly. Also, execution of this code helps to evaluate whether they met the input and output requirement specification as specified in the Java problem given to the participants. Complexity of their code is evaluated by the CEB complexity evaluator in the BlueJ IDE [57, 15]. Feedback based on the final product code (as discussed above) are summative feedback that help MICE/programmer themselves/instructor to evaluate the effectiveness of a particular programming style for that programmer. MICE feedback encourages programmers to reflect not only on their programming style but their final product too.

The data collected from the first experiment with the programmers was later analyzed in the second experiment by teachers and TAs to understand their student's learning better and at individual level to provide customized and personalized help with the goal to modify/improve the individual programmer’s style. By viewing the students programming style and the usage of tools in IDE, an instructor can look at the process student used to arrive at the final solution. Thus, if a student is performing poorly in the class or if a student comes to instructor for help with his programming, then the instructor can now provide individualized help by suggesting the student to use another programming strategy or to accommodate some tactics to the existing programming style. MICE can also help the instructors to modify their own teaching strategies. By looking at the self-regulation performed by the entire class, instructors can find concepts that are missing in the entire class on the whole. Thus, they can accommodate more information to teach students how to perform those missing strategies and/or self-regulate, inform them of its advantages, provide the tools to perform the missing task, and encourage its usage.
4.5 Assumptions, Strengths, Weaknesses

It is assumed that the entire time programmers were using MICE they were thinking about the program and reflecting on their style. As far as other tools in BlueJ are concerned, except MICE, we have tracked them by the participants' response in the questionnaire. We assume student participants were truthful when asked about the usage of the tools. Participants were not allowed to use humans for help, talk over the phone or chat with friends while programming which might have changed their coding style.

MICE works in the background, students are unaware of its presence while MICE is tracing their programming interactions with the IDE and students can concentrate on their task at hand. MICE takes initiative and provides feedback to improve their programmers style or code quality by not only displaying their own coding style but also by allowing comparison of their own coding style with experienced and inexperienced programmers. However, the programmer has the option to accept the feedback or ignore it. The feedback is not forced on the programmer.

It is more likely that students will react differently when they become aware that their interactions with the software system are being tracked and that this information will be available to the teacher and TA. Also, interactions are tracked only with respect to compile event or usage of tools, many interactions that can be tracked via execution event, added/deleted event are missing. Thus, detecting programming style is limited to only those interactions that we could track with respect to compiling and usage of tools.

The next chapter shows the results of the experiment along with the strength and weakness of MICE as identified by us, the instructors, and participants themselves.
Chapter 5

Results and Analysis

The results reported in this section are based on the interaction data tracked in the ontology, responses from participants’ questionnaire, their submitted programs, and responses from instructors’ questionnaire. This core data, in its raw form, is presented first, followed by the interpretation of the data, corresponding to each research question.

5.1 Research Question 1

"Do student programmers have "identifiable" programming styles?"

This research question was intended to identify individualized programming styles of students, irrespective of students’ background or the types of coding problems they addressed. The aim is to confirm whether contemporary technologies are capable of detecting individualized programming styles, without resorting to either judging the quality of the style (good or bad or popular style) or associating preferred styles to categories of programmers (who used what style). Further, we explored the possibilities of detecting programming styles in an automated manner and offer individualized feedback to students.

As per our definition, Programming Style of a programmer refers to the ability of programmers:

- to plan their goals,
- to plan the sequence of tasks to achieve the goal,
- to follow code conventions,
CHAPTER 5. RESULTS AND ANALYSIS

- to engineer code in a disciplined manner,
- to systematically debug code,
- to optimize code development and delivery through appropriate settings in IDE (Integrated Development Environment),
- to regulate completion rates and quality of programming tasks, and
- to efficiently collaborate with other programmers and resources.

Current implementation of MICE is equipped to track interactions corresponding to all the items listed in the definition. Collected information is then presented to programmers in graphs and programmers could then interpret the graphical information to reflect on their coding styles. Current implementation of MICE is not designed to make any coding-style related inference at this time.

From the core data set, we identified a number of programming styles with respect to the categories mentioned above. Programmers' coding patterns were detected based on their code construction style, error solving pattern, error occurrence pattern, compiling pattern, ability to reuse/cross-reference, and usage of tools to perform software engineering.

The usage of tools by each programmer for each program was traced by MICE. Additional data were collected from the questionnaire that participants filled in after completing each program in the experiment. In this section, we also present how various graphs presented by MICE's Programmer Style Viewer can be interpreted to identify a programmer's coding style, as well as the strong and weak programming patterns of the programmer.

The following sections present how individual components of programmers' style were traced and presented graphically.

5.1.1 Number Of Bytes (NOB) versus Compile Time Graph

NOB versus Compile Time graph identifies a student's code development style with respect to:

1. How often students compile (their compiling pattern). Whether programmers compile periodically, in quick intervals or with large time gaps, or compile at irregular intervals, or at random compile patterns. By observing how close or apart these dots are
placed in the graph and translating this observation into a pattern, we converge on programmers' compilation style.

2. How programmers construct code with respect to the size of the code. They can have an incremental approach - increase the lines of code one method at a time; or zig-zag approach - where programmers write, delete, or cut-and-paste code in a random order. The zig-zag approach includes starting with a generic program and then deleting parts that are not needed. These patterns can be observed by looking at the lines plotted on this graph, as the size of the code is represented in the Y axis.

3. How much time do programmers dedicate for initial planning, code development; and code testing? Participants were given 30 minutes to solve each program. The 0 on the x-axis represents the time of their first compile. Time dedicated by the programmer for initial planning and initial code development can be calculated by subtracting the value in minutes for the last compile from the total time taken for the program (30 minute for this experiment).

4. To know whether programmers reused another code or wrote the code by themselves. This can be detected by observing the increase in code size with respect to time taken to code.

The X Axis of the NOB Versus Compile Time graph represents time from the start of the first compile (in minutes) and the Y Axis represents the size of the code (in bytes) developed so far. The points are plotted with respect to each compile. When participants compile multiple classes, each class was represented in a different color.

Figure 5.1 shows the NOB versus Compile Time graph of programs 1 and 2 created for participant 4 based on this participant’s interaction with MICE. For the first program, the participant wrote code (without any testing) during the first 20 minutes and then tested the code in the final 10 minutes. We observe that in the 1st graph the size of the code remains static, stays around 2000 bytes, and the X-axis shows the last compile at the 10th minute - thus the participant spent 10 minutes in checking the correctness of the code by compiling. However, in the second program, the programmer followed an incremental approach - wrote parts of the code, tested it; and then added another part to the program, followed by a test.

\(^1\) It is quite possible to observe the code developmental style with respect to language constructs. However, we have restricted the scope to this research to deal only with the size of the code.
Thus, the programmer changed the style from writing all code at once and then testing to writing some code then testing and so on. For the 1st program he dedicated only 10 minutes for testing his code and 20 (30 minutes - value of the last number on the X axis \(= 30 - 10 = 20\) minutes) minutes on planning and initial code development. On the other hand, in the second program the programmer spent more time in testing and code development (28 minutes i.e. value on the x-axis corresponding to the last compile) and 2 minutes (30 - 28 = 2 minutes) in planning and initial code development. Since the programmer took just 2 minutes for planning and initial code development and wrote 1000 bytes in that time, it is obvious that most of the initial code for program 2 was copied from another code.

Figure 5.2 shows the NOB versus Compile Time graph of the programs 1 and 2 created for participant 3. For both the programs, this participant used all 30 minutes for the coding both the problems. Participant 3 used the same code construction strategy for both his programs - code everything and then perform testing. For the first code, the participant spent 29 minutes (30 - value associated with last compile on the x-axis = 30 minutes - 1 minute = 29 minutes) in planning and initial code development and last 1 minute in testing the code. For the second code, the participant spent 27 minutes in planning and initial code development and 2 minutes in planning and initial code development and wrote 1000 bytes in that time, it is obvious that most of the initial code for program 2 was copied from another code.
CHAPTER 5. RESULTS AND ANALYSIS

This cannot be considered as a bad style. Programmers have their own styles that they are comfortable with. However, if this programmer is performing poorly in the class or comes to the instructor for help, looking at these graphs the instructor can advice him to use another code development strategy.

Figure 5.3 presents the NOB versus Compile Time graph of the programs 1 and 2 created for participant 17. This participant used different strategies to solve two problems. For the first code, the participant used a zig-zag approach, where he would write a code, test it, if unsuccessful - eliminate a part of the code and think of another strategy to solve it rather than trying to correct the error. Whereas, in the second code he uses an incremental approach; where he codes part of the program, tests it, codes another part, tests it and so on. Thus, this programmer doesn’t have a fixed style, he adapts his style based on the kind of problem.

Figure 5.4 presents the NOB versus Compile Time graph of the programs 1 and 2 created for participant 14. This participant used different strategies for solving the problem. For the first code, he wrote the entire code chunk and then eliminated the problematic or unused
CHAPTER 5. RESULTS AND ANALYSIS

(a) Program 1
(b) Program 2

Figure 5.3: NOB Versus Compile Time - Participant 17

(a) Program 1
(b) Program 2

Figure 5.4: NOB Versus Compile Time - Participant 14
code. It might be possible, that he copied most of the code and then went on compiling to remove the stuff he did not need and adding the new functionality as needed. For the second program, he used an incremental approach.

Thus, a programmer might or might not adhere to a particular code construction strategy. We have observed how they change their styles from program 1 to program 2. We have also observed how they adhere to the same strategy while constructing program 1 and program 2. However, if a student has difficulty with his programming, the instructor or student himself might look at his previous code construction styles and try changing his code construction for better results. Analysis has not been performed to see which code construction style is better. However, code construction style seems to change based on the type and complexity of the given problem. Also, we believe that there is no one particular good style that suits everyone. If a programmer is not achieving a high quality programming code while using a style, he might change his style. A code construction style that suits one programmer might not be good for another programmer.

Also, for most of the programmers, we have noticed that the planning and initial code development times for the second program was much shorter compared to the first program. Mostly all participants have reused parts of their previous code and indicated that they are not comfortable reusing others’ code. This might be because either participants did not know where to look for a reusable code or they did not trust others’ code.

5.1.2 Un/Successful Compiles Graph

The Un/Successful graph’s X Axis represents time from the start of the first compile (in minutes) and the Y Axis represents the size of the code (in bytes). Points are plotted on this graph with respect to each compile. In case the programmer develops multiple classes, each class would be represented in a different color. It is the same as the previous NOB versus Compile time graph, except that each point plotted is red or blue in color indicating unsuccessful error-prone compiles or successful error-less compiles, respectively. This graph also traces the following patterns:

1. Detects presence of useless compiles i.e., many compiles in a short time span with no change in size of code.

2. How the code construction changes when an error occurs; this change can be observed with respect to change in size of code and frequency of compiles. What we mean
by observing a change with respect to change in size of code once an error occurs is, do programmers solve the error by eliminating the problematic code chunk and using another method; or do programmers adhere to the same method of code construction; or do programmers not care about the error(s) and go on incrementing their code ignoring the errors. Also, change in frequency of compiles (ie increase or decrease in the number of compiles) can be observed once an erroneous compile is encountered.

The following data argues in support of this claim:

![Figure 5.5: Unsuccessful and Successful Compiles - Participant 2](image)

Figure 5.5 depicts the pattern of Un/Successful Compiles for participant 2 with respect to code construction of program 1 and 2, respectively. His first program, the compiles at the 2nd minute and the 3rd minute have the same number of bytes. This can be because the programmer didn’t make changes to the code and just compiled again or he did make a change where the code deletion and insertion resulted in the same amount of bytes. The former is a case of useless compile. In this case there is a presence of just one useless compile. However, a number of useless compiles indicate waste of time (as we see in the following). The compiles at 12, 13, 14, and 15th minutes can be considered as useless compiles since there are no changes in the number of bytes. Even for the second program, compiles at
the 1st and 2nd minute; compiles at the 9th and 11th minute; and compiles at the 13, 16, 17, and 18th minutes can be considered as useless compiles. Presence of useless compiles indicate that student might have encountered an error and could not come up with another strategy or a solution to solve that error; or doesn’t know what to do next (this is due to lack of planning or they are trying to fix a typographical error).

In the first program, the programmer started with an erroneous code. He tried to solve the error by deleting small parts of the code and compiling regularly. But, after failing to solve the error, he eliminated the entire problematic chunk of code that resulted in a successful compile (in the 1st minute). After that, he added another chunk which resulted in errors that forced him to remove the entire chunk of newly added code, and then slowly increment the code and test it (rather than add chunks and remove chunks of code).

![Figure 5.6: Unsuccessful and Successful Compiles - Participant 17](image)

Figure 5.6 depicts Un/Successful Compiles with respect to code construction of program 1 and 2 created for participant 17. In both programs there are no useless compiles. After each compile there was a significant increase/decrease in code.

A change is observed in the coding pattern with respect to frequency of compiles as well as the size of the code once an erroneous compile occurs for both the programs created by
participate 7 (Figure 5.7). For the first program, once the error (at the 1st minute) occurs, the frequency of compiles increases drastically (3 compiles in less than 15 seconds with 20 bytes of code change). No useless compiles are observed in the second program.

Once an error occurs, participant 21’s frequency of compile reduces (Figure 5.8) considerably. Either he solves the error immediately or takes a long time to solve it.

5.1.3 Error Pattern Graph

The X-Axis of the Error Pattern graph represents time from the start of the first compile (24 hr clock) and the Y-Axis represents 'compile instance number' when the error occurred along with the type of error. This graph shows the time bifurcation on whether the time was spent in solving the error or in constructing the code. The horizontal red block indicates the time taken to solve the error. The longer the block the more the time taken to solve the error. Missing parts in red blocks indicate absence of errors.

This graph identifies the programmer’s code development styles with respect to:

1. Time spent on solving an error and planning/developing code. This graph helps to bifurcate the time spent by the programmer into a) time spent on solving an error
and b) time spent on planning and code construction.

2. Programmer’s learning pattern. Helps to see if students learn from his mistakes e.g. if an error occurs at a particular compile, future compiles can be observed to see whether that error occurred again. Also, a similar type of error can occur multiple times while coding. However, time taken to solve the same kind of error should progressively decrease.

3. This graph can help identify concepts that a student has difficulty with. The larger the red block the more the time programmers spend solving the error. The more the time programmers spend the more the difficulty they have with the particular concept.

Figure B.1 and B.2 in Appendix B depicts the error pattern graph for participant 17 corresponding to programs 1 and 2. There is a significant reduction in the number of errors from program 1 to 2. Also, errors observed in program 1 were not repeated in program 2.

Figure B.3 and B.4 in Appendix B shows that there is an increase in the number of errors from program 1 to 2 for participant 18. However, errors from program 1 were not repeated in program 2.
For participant 14, even though there is a reduction in the number of errors from program 1 to 2 [Figure B.5 and B.6 in Appendix B], a number of types of errors were repeated between programs 1 and 2. Also, the time taken to solve the repeated errors in program 2 does not decrease when compared with the time taken to solve these errors in program 1.

5.1.4 Error Type Encountered Graph

The X-Axis of this graph depicts the category of the encountered error and the Y-Axis represents the percentage of occurrence of that type of error with respect to the total number of errors encountered while coding a given program. This graph in combination with the previous graph can be used to identify concepts student have difficulties with. Each error can be mapped to concepts or syntax constructs in Java. This graph:

1. Groups errors encountered into types; the error types can be related to specific Java concepts or Java language constructs.

2. Identifies recurrence of errors across multiple programs written by the same programmer.

Figure C.1 and C.2 in Appendix C depicts the error type encountered graphs of program 1 and 2 created for participant 21. The graph indicates that participant 21 repeats similar types of errors, leading to a conclusion that the student might have difficulties with concepts associated with these repeated errors.

5.1.5 Error Solving versus Compile Time Graph

The X-Axis of this graph represents time from the start of the first compile (in minutes) and the Y-Axis represents the number of errors present at various compile instances. Points on X-axis (i.e., 0 no. of errors) indicate absence of errors; i.e., successful compile instances. The other points indicate error-prone compile instances with the corresponding number of errors. This graph depicts students' debugging patterns.

Figure 5.9 depicts the Error Solving versus Compile Time graph of programs 1 and 2 created for participant 6. Corresponding to the first program, an increase in the frequency of compiles is observed whenever errors are encountered, implying that the student focused his attention in solving errors. For example, at the 0th minute, 4 compiles were observed in
less than 30 seconds, when a number of errors were encountered. Predictably, the frequency of compiles reduced considerably once the errors were solved.

Figure 5.10 depicts the Error Solving versus Compile Time graph of programs 1 and 2 created for participant 17. In the second graph, we observe that the frequency of compile increases once an error is incurred. There were no compiles from 0 to 10 minutes because there were no errors during that time (compile at 0 minute was successful). However, once an error occurred at the 10th minute, frequency of compiles increased until the error was solved. Similarly, there were no compiles between 13 to 21 minutes.

5.1.6 Files Referenced Graph

The X-Axis represents the BlueJ files referenced and compiled; and the Y-Axis represents the number of times the file has been compiled (in % with respect to the total number of compiles while coding the program in full). It is good programming habit to reuse code after checking it for correctness. This graph reveals whether programmers made any effort to check the correctness of the code by compiling reused code before integrating it in their own code.
CHAPTER 5. RESULTS AND ANALYSIS

Figure 5.10: Error Solving Versus Compile Time - Participant 17

(a) Program 1

(b) Program 2

Figure 5.11: Files Referenced - Participant 1

(a) Program 1

(b) Program 2
Table 5.1: Tool/Strategy Usage and Code Analysis - Participant 1

In program 1 (Figure 5.11a), participant 1 checked the correctness of the code before reusing the code from Hello.java. Num-Multiples.java and Num-Triangle.java are the names of programs 1 and 2 respectively written by participant 1.

5.1.7 BlueJ Tools and Software Engineering/SRL techniques

BlueJ includes a number of software engineering tools that programmers can put to use. By tracking these interactions, one can observe whether students follow specific software engineering practices or not.

Let us consider the tool usage by participant 1 [Table 5.1]. The columns 1 and 2 represent Programs 1 and 2, respectively. In the Table, '1' indicates the usage of tool by the programmer and '0' indicates that the tool was not used by the programmer. The complete list of BlueJ tools are split into 3 blocks corresponding to the 3 phases in SRL - forethought, performance, and reflection.
Table 5.1 indicates that participant 1 planned the development of the first program with the help of an 'algorithm'. In the performance phases of both programs 1 and 2, participant 1 used ‘print’ statements for debugging. While constructing program 1, the programmer sought help from books, the Internet, and the cross referencing other code (i.e. reusing parts of other code). Cross-referencing is also observed in the second program. Participant 1 used MICE to reflect on his style and the CEB Complexity Evaluator tool to check his code complexity. Also, participant 1 ensured the readability of his code by using proper indentation, appropriate commenting, and suitable names. Thus, students’ coding trend can be observed with respect to specific strategies and tools, corresponding to the phases of planning, performance, and reflection. MICE can generate feedback specific to strategies and tools preferred by the instructor. For example, Participant 1 used an algorithm for planning the first program but did not plan for the second program. It is quite possible for MICE, under instructions from the teacher/TA, to offer suggestions corresponding to this specific observation.

Similarly, for participant 7 [Table 5.2], one can see minimal usage of tools across the three phases of code development. His planning only involves algorithms; he only uses ‘print’ statements for debugging; he only uses Internet, books, cross-referencing for additional resources; he only indents and applies proper naming convention towards readability; he completely ignored self-reflection of his style using MICE. If such a minimalist trend continues for a long time and if the instructor felt the need for a review, MICE can offer specific feedback to encourage the student to improve specific aspects of his coding style.

Further, based on the interviews that we conducted with the instructors, 90% (i.e., 9 teachers/TAs) agreed that MICE was successful in identifying students’ programming patterns, whereas 10% (i.e. 1 teacher/TA) thought the system was unsuccessful in identifying the styles.

In summary, we conclude that the data collected by MICE identifies specific programming styles of students and presents the same in graphical forms.

5.2 Research Question 2

“Do student programmers exhibit self-regulatory abilities? Are they inclined to use self-regulatory tools embedded in the IDE, even when these tools are only available passively?”

Before the start of the experiment, student programmers were given a demo of the MICE
## CHAPTER 5. RESULTS AND ANALYSIS

![Flowchart Algorithm Forethought Phase UML Sequence Diagram](image)

### Reflection Phase
- Jelict
- Debug Paper
- Compiling+Print
- Test Cases
- Internet Help
- Books
- Cross-Referencing

<table>
<thead>
<tr>
<th>Program</th>
<th>Tool/Strategy</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forethought Phase</td>
<td>Flowchart</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Algorithm</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>UML</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Sequence Diagram</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Test Cases</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Performance Phase</td>
<td>Jelict</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Debug Paper</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Compiling+Print</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Test Cases</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Internet Help</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Books</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Cross-Referencing</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Reflection Phase</td>
<td>MICE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>CEB Complexity Evaluator</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>PMD</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Proper naming</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Comments</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Indentation</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Code Analysis</td>
<td>Size of code</td>
<td>557</td>
<td>1038</td>
</tr>
<tr>
<td></td>
<td>Achieve Input Specification</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Achieve Output Specification</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Achieve Code Quality</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Code Complexity</td>
<td>1/6</td>
<td>2/6</td>
</tr>
</tbody>
</table>

Table 5.2: Tool/Strategy Usage and Code Analysis - Participant 7
system and they were informed how to reflect on their style or get help from MICE’s Programmer Style Viewer. Participants were also introduced to other tools available within BlueJ that correspond with various software engineering tasks. It was made clear that participants could choose or not choose to use the tools. All of BlueJ’s tools were made available passively except MICE’s viewer of programming styles. This viewer observes participant interactions for about 20 minutes and pops-out a message concerning the coding style of the participant.

Students filled a questionnaire after completing the coding task of each program. One of the questions in the questionnaire asked the participants whether they used BlueJ tools and if not, did they use any other technique to perform software engineering tasks associated with coding.

Figure 5.12: Forethought Phase of All Participants

Figure 5.12 depicts the tools and strategies used in the forethought phase by all participants. Forethought phase refers to participants’ planning of the coding task. It involves
programming interactions such as drawing flowcharts, writing algorithms, drawing UML diagrams, drawing sequence diagrams and writing test cases to verify conformity to I/O specifications. Noticeably, none of the students used the UML viewer, use the sequence diagram editor, or write test cases. Also, flowcharts were drawn by just 4% of the participants in all. Algorithms were used by 35% and 22% of participants in the 1st and the 2nd program, respectively. Thus, participants did not engage much in the forethought phase.

![Figure 5.13: Performance Phase of All Participants](image)

Figure 5.13 depicts the tools and strategies used in the performance phase by all participants. This phase indicates how students go about constructing their code, including code development and modification, debugging of code, (e.g., using Jeliot graphical debugger, debugging on paper, compiling with ‘print’ statements, and so on), writing test cases, seeking help from the Internet/discussion board/books, and cross-referencing other programming code.
As we can see from Figure 5.13, all participants (100%) use compiling with print statement for their code construction and debugging. However, a very small percentage of participants use either the graphical debugger tool or debug on paper. Not one of the participants wrote test cases to check the correctness of the code. Participants (70% and 74% in 1st and 2nd program respectively) preferred the Internet to seek information about bugs and related code. They looked at discussion boards, tutorials, examples code on the Internet, online Java documentation, and so on. 35% and 20% used the book for help in the 1st and the 2nd program, respectively. Also, reusing the code was done by 22% of participants in the 1st program and 57% of participants in the 2nd program. The increase in cross-referencing from program 1 to program 2 can be attributed to the fact that many participants reused code from their own program 1.

Participants executed the performance phase tasks mostly manually. Quite surprisingly, they do not seem to rely on automated, built-in tools (Jedi software debugger and Test Case writer/tester).

Figure 5.14: Reflection Phase of All Participants
CHAPTER 5. RESULTS AND ANALYSIS

Figure 5.14 shows the tools and strategies used in the reflection phase. This phase shows whether participants attempted to make their code readable and whether they reflected on their coding style. MICE's Programmer's Style Viewer enables them to reflect on their programming style, the CEB complexity evaluator and the PMD unused block remover enables them to reflect on their code, and proper naming, commenting, proper indentation enables them to make their code readable.

Although MICE is a relatively new tool, 44% and 65% of participants used it to reflect on their coding style in the 1st and the 2nd programs, respectively. Very few participants used the complexity evaluator for their code and the PMD tool to remove the unused block in their code. Most participants attempted to make their code readable by using proper naming for their variables/program name/method name/class name (70% and 74% in the 1st and the 2nd programs, respectively), commenting their program (31% and 52%), and having proper indentations (100% and 100%).

Only a handful of participants attempted to explore and use the tools in the performance and reflection phase. However, the forethought phase seriously lacked participant involvement. Yes, participants do exhibit self-regulatory abilities, but their involvement is meager when the automated tools were made available passively.

5.3 Research Question 3

"Do intermediate student programmers exhibit more self-regulatory abilities than novice student programmers?"

The self-regulatory abilities of beginner and intermediate programmers have been evaluated based on their usage of SRL tools. There are 3 phases in Self-Regulated Learning and for each phase we have 2 graphs - one for beginner programmers and another for intermediate programmers.

Figure 5.15 depicts tools and strategies used in the forethought phase. The first graph represents usage by beginner participants (10 beginner participants) and the second graph represents usage by intermediate participants (13 intermediate participants). We do not observe any significant difference between these two usages between beginner programmers and intermediate programmers in the forethought phase.

Figure 5.16 depicts tools and strategies used in the performance phase. There is no significant difference in the usage of the Jeliot graphical debugger tool and the Test Case
CHAPTER 5. RESULTS AND ANALYSIS

Figure 5.15: Forethought Phase

(a) All Beginners

(b) All Intermediate

Figure 5.16: Performance Phase

(a) All Beginners

(b) All Intermediate
tool. However, a significant number of beginner programmers rely on reusing code and using books than intermediate programmers.

![Graph](image)

(a) All Beginners  
(b) All Intermediate

Figure 5.17: Reflection Phase

Figure 5.17 depicts the usage of tools in the reflection phase. There is no significant difference in the usage of tools for self-reflection (MICE, CEB, and PMD) between beginners and intermediates. Also, attempts to make their code readable, proper naming, commenting, and indentation, remains the same for both groups.

In conclusion, we observe that there is no significant difference in the self-regulatory abilities of the beginner and the intermediate programmers.

### 5.4 Research Question 4

"Are novice programmer’s faster learners of Self-Regulated Learning principles than the intermediate programmers? Can SRL be taught progressively?"

Data gleaned from graphs presented in the earlier section is used to compare the proportion of tools and strategy usage between the intermediates and the beginners, across programs 1 and 2. The increase in SRL tools and strategy from program 1 to 2 shows that their awareness of self-regulation is increasing and thus participants are progressively
learning about the utility of SRL tools and strategies. This increase in SRL can be mainly attributed to MICE’s feedback.

A comparison of programs 1 and 2 for the beginners (Figure 5.15) indicates that the percentage of participants using flowchart remains the same. However, there is a marked decrease in the usage of algorithms. A similar trend is observed among the intermediate programmers.

Other than the usage of cross-referencing strategy for the beginners group (Figure 5.16), there is no significant change in the proportion of students using the tools and strategies for SRL between the first and the second program. There is a significant increase in the usage of MICE tool and comments usage (Figure 5.17) among the beginners.

We can conclude that there is no difference in the SRL learning curves between the beginners and the intermediates.

5.5 Research Question 5

"Are programmers open enough to change their programming style and does this change lead to better programs?"

As we have already discussed, except the forethought phase, there is an insignificant increase in the SRL tool and strategy usage from program 1 to program 2 (Figure 5.12, 5.13, 5.14). Also, as we have discussed under research question 1, many participants have changed their programming styles (code construction and error solving style) in programs 1 and 2. Thus, we conclude that programmers are open enough to change their coding style.

Let us now look at its effect on the code quality.

As depicted in Figure 5.18, there is an increase of 26% in participants to achieve input requirements, 5% in achieving output requirements, and 13% increase in achieving the code quality. Further, the complexity has increased by 11%.

Figure 5.18 indicates an increase in the efficiency of the code. The code efficiency is measured with respect to

1. the programmer meeting the input and output requirement (as specified in the problem given to each participant),

2. the quality of the code with respect to it being user friendly interface,

3. the complexity of the code
Let us elaborate this further. Input requirements are measured with respect to how input was taken - was it taken from the user, was it randomized, or hard-coded. Output requirements are measured according to the output being displayed matching the output requirements of the problem.

We have limited programs code quality evaluation to input and output being user friendly. For example, if input was taken from the user, was the user confused about providing the input, about the need for the input, about the kind of input, and so on; or was it clearly specified; or when output was displayed does the user know what it is, how is output related to the input, is the output representation readable, and so on.

The complexity of the code is measured by CEB complexity evaluator which uses the following metrics [57](Appendix A):

1. NOC (Number of Children)

2. DIT (Depth of Inheritance Tree)
3. LCOM (Lack of Cohesion in Methods)

4. WMC (Weighted Method per Class)

5. RFC (Response for a Class)

6. CBO (Coupling between Objects)

We conclude that programmers are open enough to change their programming styles and this change leads to better programs with respect to achieving input/output requirements and code quality. We also note an increase in code complexity as they change their programming styles.

5.6 Research Question 6

"Does student programmer's data provide appropriate context for the teachers to identify their programming styles and give feedback to students based on their programming styles? Does this data offer help to teachers/TAs to provide more customized and personalized help to the student? Will teachers take appropriate measures to encourage usage of Self-Regulated Learning in their courses? Do teachers trust the data collected by the system?"

To answer this question, we conducted interviews with the instructors and summarized their opinions. Instructors were given a demonstration of the MICE system. In that, they were shown how data was collected, how student data was presented, how the data gets interpreted, and how feedback gets generated.

The data collected from the experiment involving student Java programmers was analyzed and the results were presented to the teachers/TAs. After this demo, teachers and TAs were asked to fill a questionnaire that asked them to evaluate MICE with respect to identifying the student patterns, their weak and strong programming habits, identifying the Java concepts student programmers might have difficulties with, helping them modify their teaching strategy/style/curriculum, their confidence in data that MICE presents and the way MICE represents the data. 5 teachers and 5 TAs who had taught programming were asked to participate in this part of the experiment.

The exact questions asked in the questionnaire were:

1. How successful was the system in determining the students programming patterns (development styles)?
2. How helpful was the data in identifying the student’s weak and strong programming habits?

3. How helpful was the data in identifying the Java concepts that the student might have difficulties with?

4. How useful is this data in helping you modify your teaching strategies e.g. to include topics to teach students the necessary skills that you observe are missing while viewing their programming style; or encourage/educate self-regulated learning or stress on importance of planning or any other software engineering techniques that you observe are missing while seeing the self-regulatory ability bar graphs of entire class?

5. How confident do you feel about using the data?

6. How happy are you with the way the data is represented?

Let us look at the results overall and then deal with each question separately.

The responses provided by the instructors are averaged and presented in the bar graph (Figure 5.19) with the X-axis presenting the questions that were asked. The scale used in the questionnaire was 1 to 5, with 1 being not very successful and 5 being very successful and 3 being neutral. As we can see graph 5.19, the average for all questions is 3 or above. Thus, on the whole teachers and TAs agreed with the success of MICE to make a positive difference to student programming.

9 out of 10 teachers/TAs agreed the system was capable of detecting students’ programming style. Some of the excerpts by the instructors regarding this question were:

- “The visuals are very helpful in identifying patterns”
- “the system does not produce integrated graphs. The data is there, but not integrated.”
- “Bytes versus Time graph really helps because it tells a lot about the persons coding style.”
- “From the presented material related to the simple java problems it was possible to see the patterns. It has to be seen how this would scale to the larger projects”.
9 out of 10 teachers/TAs agreed the system generated data was helpful for them to identify their students' weak and strong programming styles. Some of the comments by teachers were:

- "It provides a great place to start talking with students (or students to reflect) on their habits. I really like the idea of comparing experienced and novice programmers."

- "Helps find people who are over-thinking and under-using tools."

- "It would be interesting to get some information about types of edits student does."

- "Un/Successful compiles and error pattern show what kind of problems the student has and how he/she solve them."

- "It gives a fair deduction of possible strengths and weakness. Definitely a place to start evaluating the student."
"It is very obvious what habits they have. Whether it's a strength or a weakness requires more analysis".

For the next question, 3 instructors felt that the system data was useful to them to identify the concepts programmers might have difficulty with, and another 3 felt that the system was not useful to do so and the remaining 4 had no opinions. Some of the comments by the instructors for this question were:

- "Considering that there can be typos and missing semi-colons in a written code: most of the time it is hard to determine whether the problem is just a simple typo or a misunderstanding of Java concepts."
- "The mapping of errors to concepts needs to be done first."
- "Error Pattern and Error Type Encountered will show what kind of difficulties they encountered. But the errors are quite general."
- "The concepts listed are the cryptic Java errors. It would be better to have some classification or interpretation of the errors."

8 teachers/TAs felt that the system helped them identify concepts that might be missing in the entire class and thus, they could modify their teaching strategy to include those concepts, teach the importance of those tasks, and tools available to accomplish those tasks; the remaining 2 teachers/TAs had no opinions on this. Some of the comments by teachers/TAs were:

- "I can see it being used by instructors and TAs to give individual feedback, to students to reflect on their own, and the "whole class" charts to see where the class as a whole needs help."
- "I believe the view of the data to be non-integrated. I want to see this data related to type of code construct."
- "It is clear what aspects of programming are being used and ignored."
- "The data is very useful in self-regulated learning. One good way of this is to write specific programs to strengthen certain concepts in programming."
9 teachers/TAs trusted the data presented by MICE and were confident enough to use it in their classrooms. Whereas 1 teacher/TA was neutral about bestowing confidence in MICE's data. Some of the comments by teachers/TAs were:

- “I would still need to look at the product - the code itself and the output. But this is an excellent tool for exploring further into the programming styles. I would feel most confident with this combination.”
- “Correlation with actual program code is absent.”
- “Error lines versus Compile Time is not very useful as feedback. But if it was tied to seeing versions of code that relates to those errors it might be very useful for an instructor.”
- “Easy to understand and explain to others.”
- “The graph is very easy to understand.”
- “It would be good to identify how the data could be incorrect or interpreted differently. This would increase confidence in the data.”

7 teachers/TAs felt that the type of graphs (data representation) used by MICE to present the data to student programmers as well as the teachers/TAs were quite fitting. 1 felt that the data representation was inappropriate, whereas 2 expressed neutral opinions. Some of the comments by teachers/TAs were:

- “A set of disconnected graphs make it very difficult for a person to get the “overall” picture”
- “There are some non-intuitive representations but after getting used to them it is OK.”
- “The timeline is sometimes scaled down, and it makes it harder to compare two data. It should be fixed to 30 minutes.”
- “Its adequate but could be more polished and processed for quality and interpretation.”.

Some comments with respect to the system were:
• “I think this could be an extremely powerful tool for learning, giving feedback, and reflecting on programming.”

• “I was hoping to see some proactive feedback from the system happening”

• “Neat software! Good job :)”

• “It’s a very good idea and removes the problem of the student saying they are doing something but not actually doing it. I do have my concerns about the quality and accuracy of the data that is collected. People do not necessarily use the tool in the intended fashion.”.

5.7 Programmer Participants Evaluation of MICE

MICE provides help to the programmer not only to reflect on his style, but also to identify his weak concepts in Java, to compare his style with other programmers (experienced and inexperienced) and to localize bugs in his program.

To summarize, the programmer participants were given 2 different Java problems each. After each Java program, participants were asked to fill a questionnaire. The questionnaire asked them questions about their experience with the system, ease of adapting to it, usefulness of the feedback by MICE, and the various tools and techniques they used to perform various programming tasks. This questionnaire data not only helps to answer the research questions, but also makes the participant programmers aware of the strategies they used or that they could use thereby encouraging further reflection on their programming style. The questionnaire also asks the participants to evaluate MICE with respect to feedback, effort, usefulness etc. Thus, there is a questionnaire after the 1st and 2nd program, to observe change in SRL tools/strategies and evaluation of MICE.

Following are the questions that student programmers answered in the questionnaire to evaluate MICE and its feedback:

1. How confident do you feel about using the system after this experiment?

2. How much effort did it require to learn?

3. How helpful was the system in providing feedback to identify your good and bad programming habits?
4. How helpful was the comparison between your programming style; and an experienced and a non-experienced programmer?

5. How helpful was the system to localize your bugs?

6. How helpful was the system in helping you identify concepts in Java that you might have difficulties with?

7. How much did you like the BlueJ IDE for your programming needs?

Programmers response to the above questions have been represented by pie charts. Since there were 2 questionnaires, one after each program, there are 2 pie charts for each evaluation question. Both the questionnaire were exactly the same. Result for each evaluation question are analyzed not only individually, but also comparatively with respect to response obtained from the 1st and 2nd questionnaire.

![Pie charts](image.png)

(a) Questionnaire 1  
(b) Questionnaire 2

Figure 5.20: Q.1 How confident do you feel about using the system after this experiment?

**Q.1 How confident do you feel about using the system after this experiment?**

In the first questionnaire (Figure 5.20a), 13 students trusted the data presented and feedback provided by the system, whereas 3 participants had no opinion and 8 were not too confident about the system and its usefulness.
As the experiment progressed, the number of participants having confidence in the system increased to 16 and number of participants not trusting the system decreased from 8 to just 3 participants (Figure 5.20b). Thus, we can see, with more usage, there is an increase in the evaluation of the system.

Comments by student programmers for MICE were: “GUI/MICE help is VERY USEFUL”, “the pop-ups are distracting”.

Comments for the BlueJ IDE were: “I didn’t like the environment at all”, “I didn’t know how to use help in BlueJ. When I use other editors, I use “help” menu a lot to find out all available methods.”, “GUI help is very useful in program - implementation on a visual basis”, “The second time was better to use the BlueJ. But since I had limited time, I couldn’t fully utilize all the tools that were available. I still like JCreator Pro which shows all the available methods as I type.”, “Much easier compared to Eclipse”.

![Figure 5.21: Q.2 How much effort did it require to learn?](image)

**Q.2 How much effort did it require to learn?**

The next question was to estimate the effort required to learn and get accustomed to the new IDE and MICE. 15 participants (Figure 5.21) felt comfortable to get adjusted to the system quickly.
CHAPTER 5. RESULTS AND ANALYSIS

Comments by student participants related to this question were: “The program is a bit hard to get used to at first.”, “Feel more comfortable in using the program for the second time.”, “2nd program was easier done because of the warm up with the 1st program”.

Q.3 How helpful was the system in providing feedback to identify your good and bad programming habits?

The success of MICE graphs to help student programmers to identify their good/bad programming styles increased from program 1 to 2 as can be seen from the results obtained from questionnaire 1 and 2 (Figure 5.22a and b respectively). In the first evaluation questionnaire, 5 student programmers felt the system was useful in identifying their good/bad programming habit and 9 felt that the system was unsuccessful to do so. However, in the 2nd questionnaire the evaluation of MICE based on identifying the programmers good/bad programming habit increased from 5 to 10 student programmer participants.

Q.4 How helpful was the comparison between your programming style; and an experienced and a non-experienced programmer?

The number of students finding the comparative graphs useful to improve their programming style increased from 6 to 11 student participants from questionnaire 1 to 2 (Figure
CHAPTER 5. RESULTS AND ANALYSIS

5.23a and b respectively).

Comments by student participants for this question were: “Features like comparison of errors with experienced programmers help the users to let him/her see his/her current progress”, “The data/comparison graphs that pops up can remind me about my progress and other errors I’ve made”.

Comparative graphs can help students observe their erroneous styles with respect to other fellow good and bad programmers. An instructor need not use a real world example of good/experienced and bad/inexperienced student programmer. Instead, the instructor can himself develop the good and bad styles that he expects the student to follow or not for a particular problem.

As far as comparative graphs for the experiment are concerned, we used data from experienced and inexperienced programmer. This data was collected from the experiment conducted on the prototype version of MICE (the prototype version of MICE collected data from programmer interaction with the BlueJ IDE and no feedback was given on his style or feedback to present his style) in November 2006.

MICE Programmer Style Viewer presents comparative graphs for the programmers’ code
construction style (Comparative Number of Bytes versus Compile Time) and error solving style. Let us look at these comparative graphs.

5.7.1 Comparative Graphs

Comparing Code Construction Graph

The X-Axis represents time from start of the first compile (in minutes) and the Y-Axis represents the size of the code (in bytes). The points are plotted with each compile. This graph is same as the Number of Bytes versus Compile Time discussed previously. The only additions are Number of Bytes versus Compile Time plots of experienced and inexperienced programmers. The red line and blue line represents code construction of an inexperienced and experienced programmer respectively.

Using this graph programmers can compare their code construction style with experienced and inexperienced programmers. Programmer can interpret this graph to adapt to the traits of the experienced programmer and avoid mistakes by the inexperienced programmer. For example, some of the traits that can be observed for an inexperienced programmer are having many useless compiles (i.e. compiling without changes to the code), writing large chunks of code at once and trying to debug it all later; and experienced programmer writes small chunks and increments the code slowly with timely testing, regular compiling, switching between different code construction style if one style doesn't work or causes problem.

In Figure 5.24, participant 3's code construction style (in black line) is compared with the code construction style of experienced and inexperienced programmer. Participant 3's number of compilation is very less compared to experienced programmer, for that matter even compared to inexperienced programmer. Also the time taken by participant 3 on testing the code is much less (just 1 and 4 minutes in program 1 and 2 respectively) compared with other programmers shown in the graph. As we can see, there is no change in planning time, testing time, code construction style, compiling pattern between the 1st and 2nd program. Thus, participant 3 is not using the information provided by the graphs to reflect on his style. Teachers and TAs can teach the programmer to reflect using these comparative graphs, if the student comes to the instructors for help with his coding.

The 2 comparative code construction graphs calculated by MICE for participant 17's 1st and 2nd program (Figure 5.25) shows the student participant changed his style from
CHAPTER 5. RESULTS AND ANALYSIS

(a) Program 1

Figure 5.24: Comparing Code Construction - Participant 3 with In/Experienced Programmers

(b) Program 2

Figure 5.25: Comparing Code Construction - Participant 17 with In/Experienced Programmers
a zig-zag approach to an incremental approach (which was followed by the experienced programmer). This change in code construction style, among other factors, can also be attributed to participant 17’s reflection ability on the code construction comparative graph.
error prone compile instances with the corresponding number of errors that are present. The red and blue lines represent error-solving patterns of inexperienced and experienced programmers correspondingly.

![Figure 5.27: Comparing Error Solving - Participant 21 with In/Experienced Programmers](image)

Participant 21’s error solving pattern for his program 1 closely resembles that of an inexperienced programmer (Figure 5.27a). As a result of which, the programmers first code was buggy too - neither did he meet the input/output requirements nor the code quality. However, in the second program (Figure 5.27b), he reflected on his mistakes and changed his error pattern to match that of an experienced programmer. He tried to keep his errors as low as possible. As soon as an error occurred, the frequency of compiles increased drastically until the error was eliminated. The code 2 met not only the input requirement but also the output requirements.

Q.5 How helpful was the system to localize your bugs?

Now that we are done discussing the comparative graphs, let’s get back to the questions answered by the programmer participants to evaluate MICE. Figure 5.28a shows that for the first program, 8 participants felt the system generated graph was useful to localize their bug, whereas 7 felt the graph was not successful to do so and 8 participants had no opinion.
CHAPTER 5. RESULTS AND ANALYSIS

5.7 RESULTS AND ANALYSIS

(a) Questionnaire 1  (b) Questionnaire 2

Figure 5.28: Q.5 How helpful was the system to localize your bugs?

The success of MICE graphs to help student programmers to localize their bug increased from 8 to 11 participants with respect to program 1 and 2 (Figure 5.28a and b respectively). However, 9 participants felt that the system was unsuccessful to do so. The graph used to help students localize their bug has been discussed next.

5.7.2 Error Line Number versus Compile Time Graph

The X-Axis represents time from start of the first compile (in minutes) and Y-Axis represents line number in their code where the error occurred. Points are plotted on this graph with each compile only if there are errors corresponding to the time of error occurrence and the line number of error. Dots indicate the occurrence of errors. This graph helps to find sections of the code that are problematic thereby helping to debug the code by localizing the error.

The hardest part of debugging is to find where the bug is (localizing the bug). A compiler can help locate only syntax errors. However, if its another kind of error, for example a semantic error, this graph can be used to localize the code by locating the part of the program with maximum error. It is quite possible, the next error that occurs, might be
in the part where the maximum errors occurred. When programmers encounter an error and cannot localize or locate the error, they should first try to debug these identified sections prior to exploring the rest of the code.

Once programmers identify the section of code where the bug is, they can check the variable values, parameters passed to any routines, and the returned values. According to MICE, the area of the code where there is presence of many error dots can be treated as their problematic code area. Thus, MICE gives feedback to place print statements in this area to check the values of variables, or place breakpoints (where debuggers stop execution) while debugging or analyzing this area manually.

Though, one of the problems of this approach is that MICE does not update the line numbers if there is a code pasted on the top of the file. Also, the error plots are syntax errors by the users, and using those to find semantic errors may not be that useful. For effective localization of error a better approach has to be used.

As far as 1st program is concerned, participant 7's code can be localized to approximately line 14 to 17. Thus, his search for error in the 29 lines program has been reduced to just 4 lines (Figure 5.29a). As far as the code 2 by participant 7 is concerned, his errors are all

![Figure 5.29: Error Line Number Versus Compile Time - Participant 7](image)
over the code, thus there is no localization for his program 2 (Figure 5.29b) by MICE.

Q.6 How helpful was the system in helping you identify concepts in Java that you might have difficulties with?

Getting back to the participants evaluation of MICE, for the 6th question most of the student participant’s weren’t satisfied with the usefulness of MICE data to identify their weak concepts in Java. MICE presents only the types of errors students faced with. However, students found it difficult to map these errors to Java concepts. To perform this mapping of errors to Java concepts is the possible next step of MICE software development. In both programs, only 6 participants felt MICE was useful in identifying their weak Java concepts, whereas 8 and 9 participants for the 1st and 2nd program felt it did not (Figure 5.30).

Q.7 How much did you like the BlueJ IDE for your programming needs?

Figure 5.31 shows many participants were not happy with the BlueJ IDE for their programming need - 10 participants in the 1st program and 8 participants in the 2nd program. 8 and 9 participants in the 1st and 2nd program respectively were happy with the BlueJ IDE. Even though there is an increase in the liking of BlueJ IDE and decrease in the number of students hating it, it’s not that significant.
CHAPTER 5. RESULTS AND ANALYSIS

Figure 5.31: Q.7 How much did you like the BlueJ IDE for your programming needs?

Seeing the evaluation data, we conclude within the limitations of the study, that MICE was successful with both the end users - instructors as well as the programmers. The evaluation of MICE increased with usage (observed over a span of just 1 hour - from program 1 to 2). Thus, programmers and instructors are acceptable towards MICE; and are confident of its usefulness and ability to improve programmers programming style.
Chapter 6

Future Work

MICE research opens up a number of new avenues of explorations. We will briefly discuss some of these avenues.

MICE can be modeled to engage programmers in Mixed-Initiative (MI) interactions. The current implementation takes minimal system-oriented initiatives based only on time elapsed and compile events. It is quite possible to enable conversants, in our case, programmers and the MICE system, to initiate discussions on topics related to strategies to develop programming styles. Also, programmers can initiate discussions with the system concerning MICE’s feedback; i.e., ask the system for an explanation as to the basis of that feedback. Once the system provides the programmer with a feedback, the programmer need not blindly accept or reject the feedback. Instead, the programmer could negotiate with the system with appropriate goal orientation, where goal-orientation could be based on SRL. Thus, theory-oriented mixed-initiative feedback in coding could help programmers to trust the feedback more, give importance to computer-generated feedback, reflect and change their styles based on the feedback, and receive explanations from the system. The future version of MICE could include 1) capability to explain feedback, 2) capability to offer feedback with respect to specific coding strategies, 3) capability to visualize and manipulate feedback through interactive GUI, 4) capability to estimate and present weak and strong coding styles, 5) capability to estimate knowledge corresponding to a domain concept, and 6) capability to estimate optimal paths to guide instructors.

Production rules can be written to perform most of the tasks and subtasks related to coding. Events represented through changes in ontologies would be able to provide information to the working memory and also trigger execution of rules. Specifically, the
purposes of MICE rules would be:

1. to computationally recognize programming style components;

2. to identify opportunities for system-initiated interaction, opportunities such as programmers spending too much time debugging a piece of code or programmers consistently failing to construct task models [91];

3. to engage programmers in mixed-initiative dialogues [5, 18] with MICE that promotes SRL and demands improvement in their coding style. For example, the MICE system and the programmer can engage in a well-defined, role-playing conversational model that targets introduction of specific SRL concepts to programmers.

To capture a complete set of interactions of the programmer while programming, MICE has to interact with other systems such as iHelp, gStudy, and Internet Browsers [85]. Events observed in external systems can also be brought into the programmers' Interaction Ontology. For example, collaboration sessions in iHelp between programmers (via chatting, posting, and program-sharing) and gStudy events related to links-creation, highlighting, browsing, and searching can be recorded in the Interaction Ontology. Figure 6.1 presents the proposed architecture for this extension to MICE.

A module that accumulates the overall programming skill development can also be incorporated, which can then be used to revise the rules. The double-dotted lines in the Figure 6.1 bifurcates the system into two parts — the interaction environment that contains the external interfaces for the programmers and the MICE environment that contains the model-tracing components.

The Interaction ontology supports interactions that can be tracked only with respect to compile actions. The next step is to track not only the compile events but also events observed at program execution, code modification, application of software engineering techniques (such as unit testing), and so on. Thus we can perform code analysis at much higher level of abstraction.

The Programming Style ontology, as described earlier, consists of six components, namely, code conventions, code engineering, code debugging, optimal IDE settings for coding, regulating coding tasks, and collaboration while coding. Ontology for each of these components can be extended further contributing to specific coding styles. For example, code-convention could include ontological components that recognize 3 styles of commenting of
code - a) comments-goody, where the code has extensive comments from the programmer, b) comments-versioning, where the programmer minimally includes comments about the name of the code, author, date-of-creation, last edited, and so on, and c) comments-nil, where the programmer completely ignores to comment the code. Other such examples of coding styles that could be represented in the Programming Style ontology include compile-for-pop\(^1\), code-till-you-drop\(^2\), hill-climbing-code-construction\(^3\), end-of-days-debugging\(^4\), end-of-world-debugging\(^5\), and SRL-sincere\(^6\) [85].

\(^1\)Programmer compiles code at every opportunity such as every time he/she takes a sip of pop
\(^2\)Programmer develops code non-stop for longer terms and compiles only toward the end of coding
\(^3\)Programmer constructs code incrementally with reasonable number of breaks, compiling intermediate code from time to time
\(^4\)Programmer starts to debug only at the very end of task completion
\(^5\)Programmer debugs at every opportunity
\(^6\)Programmer sincerely adheres to SRL and regularly reflects on his/her programming habits
Once an entire interaction and programming ontology has been constructed, we can map it, in more depth, to SRL ontology. Again, in our framework, rules can be written to map each interaction and style directly to specific SRL models. Further, we should have these mappings of programming interactions to detailed SRL phases verified by SRL experts.

Another future work includes providing feedback based on the code submitted. The current system lets us calculate the complexity of the code. However, other code analyses such as viewing the program constructs used in the code (i.e. code comparison with respect to Java Syntax Tree), or percentage of code competition indicator in terms of meaningful functions should be attempted as well.

Some of the comments we got from participants and teachers indicated that the experiment problem was too small to see the usefulness of planning and other software engineering techniques. Also, they complained that the time wasn't enough to complete the program after getting to know a new IDE and the SE tools. Another follow-up experiment should be conducted on a larger scale, preferably in a class observed over a semester, involving not only individual contributions but also team contributions. Also, SRL is a technique learned over a longer period of time. Thus, to observe significant increase in SRL usage significant amount of time is required. Same is the case to observe the importance of Self-Regulated Learning in improving learning/programming. It takes time to observe the benefits of using Self-Regulated Learning in one’s programming.

Also, we aim to develop an interface that lets the instructor define successful and unsuccessful strategies. This can be used in the comparative graphs instead of the experienced and inexperienced programmers. Student programmers can now compare their coding style with successful and unsuccessful strategy instructed by the professor. The professor can use the same successful and unsuccessful strategy for his entire programming problems in the semester or he can customize them to each problem. Further, an interface has to be designed to let the professor define the completion point of a given problem. This can help the student see how close he is to the problem with respect to the final product as required by the instructor.
Chapter 7

Conclusion

We found that most programming courses do not teach coding styles to students. Students, in most cases, develop their own coding styles, for better or for worse. We started with a generic yet practical definition for the term 'Programming Style'. The definition includes components namely, code conventions, code engineering, code debugging, optimal IDE settings for coding, regulating coding tasks, and collaboration while coding.

We then reviewed literature from related areas, particularly on 'programming style', 'software engineering', and 'artificial intelligence in education'. The review concluded that only a handful of online learning environments offer formative, theory-centric, and mixed-initiative feedback.

We then designed and developed MICE, an add-on software system to the programming IDE called BlueJ, to test how well one can nurture the self-regulatory abilities of learners through real-time interactions in online learning environments. The current version of MICE operates in the domain of Java Programming. MICE is a model-tracing system, quite similar in its design to model-tracing systems in the domains of reading and writing [91]. It traces a variety of Code Engineering practices of students and offers them opportunities to improve their coding abilities. One of the core aims of MICE is to formatively keep track of students while they perform their tasks and give SRL-oriented feedback, actively as well as passively, to promote their self-regulatory abilities.

We conducted two experiments to show that employing systems that promote self-regulation would increase students' ability to self-reflect and self-evaluate, irrespective of the domain. Based on the data collected from these experiments and within the limitations of our study we conclude that:
1. Student programmers have “computationally identifiable” programming styles.

2. Student programmers do “exhibit” self-regulatory abilities and, in general, are not inclined to use tools that promote self-regulation when these tools are available passively.

3. Self-regulatory abilities of beginner and intermediate programmers are virtually the same, except that beginners tend to rely more on reusing code than intermediates.

4. The principles of Self-Regulated Learning can be taught progressively.

5. The learning curves of SRL skills of beginner and intermediate programmers are similar.

6. Student programmers are open to review and accept changes in their programming styles based on formative feedback offered at opportune moments.

7. Data collected formatively provides appropriate context for instructors to identify aspects of students’ programming styles and offer personalized feedback to individual students.

We predict that the effectiveness of MICE can be at its peak when used regularly for longer periods of time. We arrive at this conclusion since self-regulation is a skill that is usually acquired over longer periods of time [93]. By keeping track of students’ coding habits and capabilities, for longer periods of time, MICE will be able to trace their coding styles, continuously, at real time. This will lead to MICE's ability to produce formative feedback based on long-term self-regulatory practices.

This research indicates that Self-Regulated Learning principles can be equally effective in the programming domain as in the reading and writing domains. Efforts should be made to inculcate Self-Regulated Learning principles in all aspects of computing, particularly in programming.
Appendix A

Complexity Analysis

The complexity of the code submitted is measured by CEB complexity evaluator which uses the following metrics [57]:

NOC (Number of Children)
- Depth is generally better than breadth in class hierarchy, since it promotes reuse of methods through inheritance.
- Not all classes should have the same number of sub classes. Classes higher up in the hierarchy should have more sub-classes, then those lower down.
- NOC gives an idea of the potential influence a class has on the design. Classes with large number of children may require more testing.

DIT (Depth of Inheritance Tree)
- Depth of Inheritance is the maximum length from a given class to the root of the inheritance tree. In Java, as all the classes inherit from Object class, the minimum DIT in Java is 1.
- The deeper a class is in the hierarchy the more methods it is likely to inherit, making it more complex. Deeper trees indicate greater design complexity.

LCOM (Lack of Cohesion in Methods)
- Cohesiveness of methods within a class is desirable, since it promotes encapsulation.
• Lack of cohesion implies classes should probably be split into two or more sub-classes.

• Any measure of disparateness of methods helps identify flaws in the design of classes. Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

WMC (Weighted Method per Class)

• The number of methods and the complexity of methods (i.e. Cyclomatic Complexity) included is a predictor of how much time and effort is required to develop and maintain the class.

• The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

RFC (Response for a Class)

• RFC = NLM + NRM; NLM = number of local methods in the class and NRM = number of remote methods called by methods in the class.

• The larger the number of methods, testing and debugging becomes more complicated the greater the complexity of the object the harder it is to understand.

CBO (Coupling between Objects)

• If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester.

• The larger the number of methods that can be invoked from a class, the greater the complexity of the class.

• A worst case value for possible responses will assist in appropriate allocation of testing time.
Appendix B

Error Pattern Graphs

The X-Axis of this graph represents time from start of the first compile (24 hr clock) and the Y Axis represents compile instance number when the error occurred along with the error encountered. Shows the time bifurcation on whether the time was spent in solving the error or in constructing the code. Horizontal Red Block indicates the time taken to solve the error. Longer the block, more the time taken to solve the error. Missing parts in red blocks indicate presence of no errors.

This graph helps to identify the programmers code development style wrt:

1. Time spent on solving an error and planning/developing code. This graph helps to bifurcate the time spent by the programmer into a) time spent on solving an error and b) time spent on planning and code construction

2. Programmers' learning pattern. Helps to see if student learn from his mistake e.g. if an error occurs at a particular compile, future compiles can be observed to see whether that error occurred again. Also, a similar type of error can occur multiple times while coding. However, time taken to solve the same kind of error should progressively decrease.

3. This graph can help identify concepts that student are have difficulty with. The larger the red block, the more the time they spent solving the error; thus more the difficulty programmers have with that concept.
APPENDIX B. ERROR PATTERN GRAPHS

Figure B.1: Error Pattern - Participant 17 - Program 1
Figure B.2: Error Pattern - Participant 17 - Program 2
Figure B.3: Error Pattern - Participant 18 - Program 1
APPENDIX B. ERROR PATTERN GRAPHS

Figure B.4: Error Pattern - Participant 18 - Program 2
Figure B.5: Error Pattern - Participant 14 - Program 1
Figure B.6: Error Pattern - Participant 14 - Program 2
Appendix C

Error Type Encountered Graphs

The X Axis of this graph the category of error encountered while programming and the Y Axis represent the percentage of occurrence of that type of error wrt the total number of errors encountered while coding that program. This graph in combination with the previous graph can be used to find the concepts student have difficulties with. Each error can be mapped to concepts in Java. If the student themselves can’t identify the java concept they have difficulty with based on the errors, they can save this graph and show it to their teachers/TAs to help them identify their weak Java concepts. Thus, this graph:

1. Groups all the errors encountered into types. These types can be related to the Java concepts. Thus programmers themselves/instructors looking at this graph can help identify programmer his weak Java concept.

2. Helps to view the reoccurrence of errors from one program to another.
Error Type Encountered Graphs

![Error Type Graph]

Figure C.1: Error Pattern - Participant 21 - Program 1
Figure C.2: Error Pattern - Participant 21 - Program 2
Appendix D

Question Set

D.1 Sample Novice Question Set

D.1.1 Question Set 1

1. Input a number and generate its multiples (generate first 10 multiples)
   
   Example:
   If the input is 5
   The output should be 5, 10, 15, 20, 25, 30, 35, 40, 45, 50

2. Input 10 integer number and print the largest number and smallest
   
   Example:
   If the input is 10, 3, 6, 123, 23, 12, 89, 1, 10
   The output should be largest: 123 and smallest: 1

D.1.2 Question Set 2

1. Input a number and generate its multiples (generate first 10 multiples)
   
   Example:
   If the input is 5
   The output should be 5, 10, 15, 20, 25, 30, 35, 40, 45, 50
APPENDIX D. QUESTION SET

2. Input a number between 1 to 10. Then print the following till the number.

   Example:
   
   If the input is 5, the output is:
   
   1
   1 2
   1 2 3
   1 2 3 4
   1 2 3 4 5

D.2 Sample Intermediate Question Set

D.2.1 Question Set 1

1. Input 2 strings. Concatenate them. Check whether the resultant string is a palindrome.

   Example:
   
   String1 = 123
   String 2 = 321
   Concatenation = 123321
   Resultant string is a palindrome
   
   Example:
   
   String1 = 123
   String 2 = 21
   Concatenation = 12321
   Resultant string is a palindrome
   
   Example:
   
   String1 = 123
   String 2 = 123
   Concatenation = 12312
Resultant string is not a palindrome

2. Input a set of numbers and arrange them in ascending order. Display the unordered set of numbers input and the ordered numbers.

Example:
Input: 34, 45, 2, 667, 2, 23
Output:
Unordered list: 34 45 2 667 2 23
Ordered list: 2 2 23 34 45 667

D.2.2 Question Set 2

1. Display the following:

```
   1  1  1
     1  2  3
      1  2  3  4
       1  2  3  4  5
```

2. Take a number less than 100 as input. Until you exceed the value, print like the following:

Example:
If the Input is 23; print till you reach the maximum value or the closest value to maximum that can be reached

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
D.2.3 Question Set 3

1. Display the following:

2. Input 5 numbers. Display which of these numbers are present in the Fibonacci series.

Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13

D.2.4 Question Set 4

1. Input 2 strings. Concatenate them. Check whether the resultant string is a palindrome.

Example:

String1 = 123
String 2 = 321
Concatenation = 123321
Resultant string is a palindrome

Example:

String1 = 123
String 2 = 21
Concatenation = 12321
Resultant string is a palindrome

Example:

String1 = 123
String 2 = 123
Concatenation = 12312
Resultant string is not a palindrome

2. Input 5 numbers. Display whether these number are prime numbers or not.
Appendix E

Programmer Questionnaire

An experiment was conducted targeting the first 5 hypotheses with Java programmers as participants. The participants were each given a short demonstration about the MICE system. Then they were given 2 different Java problems. Participants were given 30 minutes to solve each Java problem. After each Java program, they were asked to fill a questionnaire. The questionnaire included questions about their experience with the system, ease of adapting to it, usefulness of MICE’s feedback, and the various tools and techniques they employed during coding. Following is the questionnaire filled by the programmer participants:
MICE Evaluation – Programmer Questionnaire – Part 1

Participant # _______ Level (Beginner/Intermediate) _______

In the following questions, rate an answer according to the scale provided. Some questions may require written answers. If the space provided is not sufficient please use the back of the sheet.

Please ask if you have any questions about completing this questionnaire.

Section 1

1.1. How confident do you feel about using the system after this experiment?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(not very)</td>
<td>(no opinion)</td>
<td>(very)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.2. How much effort did it require to learn?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(not very)</td>
<td>(no opinion)</td>
<td>(very)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.3. How helpful was the system in providing feedback to identify your good and bad programming habits?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(not very)</td>
<td>(no opinion)</td>
<td>(very)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.4. How helpful was the comparison between your programming style; and an experienced and a non-experienced programmer?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(not very)</td>
<td>(no opinion)</td>
<td>(very)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.5. How helpful was the system to localize your bugs?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(not very)</td>
<td>(no opinion)</td>
<td>(very)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.6. How helpful was the system in helping you identify concepts in Java that you might have difficulties with?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(not very)</td>
<td>(no opinion)</td>
<td>(very)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.7. How much did you like the BlueJ IDE for your programming needs?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(not very)</td>
<td>(no opinion)</td>
<td>(very)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Section 2

2.1 Did you use Sequence Diagram editor provided with the system for planning?

| Yes | No |

2.1b Did you use any other planning method (please mention the technique; example: drawing flowcharts, writing algorithms, writing test cases, cross-referencing code from other sources to reuse, goal setting, use books etc.) OR None?

| Yes | No |

2.2a Did you use the Class Wizard tool provided with the system to create or edit classes and class' components (attributes, methods, constructors, and so on) so as to have an error free program structure that is suitably commented and indented?

| Yes | No |

2.2b Did you use any particular style for structuring the program for this experiment to make it readable (please mention the technique; example: comments, indentation, proper naming) OR None?

| Yes | No |

2.3a Did you use the Jeliot 3 tool provided with the system for debugging?

| Yes | No |

2.3b Did you use any other debugging method for this experiment (please mention the technique; example: compiling, debugging with break points, using test cases) OR None?

| Yes | No |

2.4a Did you use the PMD tool provided with the system to reduce the complexity of your code by eliminating unused blocks of code?

| Yes | No |

2.4b Did you use any other complexity reduction method for this experiment (please mention the technique) OR None?

| Yes | No |
APPENDIX E. PROGRAMMER QUESTIONNAIRE

2.5a Did you use the CEB tool provided with the system to evaluate the complexity of your code?

| Yes | No |

2.5b Did you use any other complexity evaluation method for this experiment (please mention the technique; example: size of your code, or Function Point Analysis etc) OR None?

2.6a Did you seek help from other external sources for this experiment (please mention the technique; example: internet, posting, use reference books etc)?

2.6b How would you seek help if it was a real world situation, and a larger project with more time on hand (please mention the technique; example: internet, posting, email or talk to a friend/human helper, online chatting, email or talk to teachers/TAs for help, use reference books etc)?

Section 3

3. Any additional comments?

Thank you for completing this questionnaire and participating in this research!
Appendix F

Programming Instructor Questionnaire

The second experiment was conducted targeting the sixth and final hypothesis with programming instructors as participants. It started with a discussion with the instructors. They were given a demonstration of the MICE system. Instructors were shown how data was collected, how student data was presented, how to interpret the data to give customized and personalized feedback to the students, and how to change their teaching strategy according to the data. The data collected from the first experiment (with Java Programmers) was analyzed and presented to the teachers/TAs to give them a glimpse of the functionalities of MICE. Further, the participants were also given an introduction to the process of tuning MICE to provide appropriate feedback to students based on the real data collected in the first experiment. After this demonstration, teachers and TAs were asked to fill a questionnaire to evaluate MICE. They were asked to evaluate MICE on a scale of 1 to 5 - 1 being the least preferred value and 5 being the most preferred value. Following is the questionnaire filled by the instructor participants:
**MICE Evaluation - Instructor Questionnaire**

Participant # __  Level (Teacher/TA) __

In the questions below, we ask you to rate an answer along a scale of 1-5. Some questions we ask you to provide written answers. If the space provided is not sufficient please use the back of the sheet.

Please ask if you have any questions about completing this questionnaire.

Section 1

1.1.a How successful was the system in determining the students' programming patterns (development styles)?

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(not very)</td>
<td>(neutral)</td>
<td>(very)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.1.b If not, why? If yes, any comments?

---

1.2.a How helpful was the data in identifying the student's weak and strong programming habits?

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(not very)</td>
<td>(neutral)</td>
<td>(very)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.2.b If not, why? If yes, any comments?

---

1.3.a How helpful was the data in identifying the Java concepts that the student might have difficulties with?

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(not very)</td>
<td>(neutral)</td>
<td>(very)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
1.3. If not, why? If yes, any comments?

1.4. (a) How useful is this data in helping you to modify your teaching strategies e.g., to include topics to teach students the necessary skills that you observe are missing while viewing their programming style; or encourage/educate self-regulated learning or stress on importance of planning or any other software engineering techniques that you observe are missing while seeing the self-regulatory ability bar graphs of entire class?

1.4. (b) If not, why? If yes, any comments?

1.5. (a) How confident do you feel about using the data?

1.5. (b) If not, why? If yes, any comments?

1.5. (c) Please make suggestions on anything more you might want to be included in the student programmers style, for example, capturing some other interactions while programmers program, removing some graphs, and/or adding some graphs etc.

(space continued on next page too)
1.6.a How happy are you with the way the data is represented?

<table>
<thead>
<tr>
<th>1 (not very)</th>
<th>2</th>
<th>3 (neutral)</th>
<th>4</th>
<th>5 (very)</th>
</tr>
</thead>
</table>

1.6.b Any suggestions on anything you might want to be modified?
We have provided a list of graphs to help you make your choice. You can choose any of the graphs mentioned in the list or specify other any other mode of representation example tabular, text, images etc.

Section 2

2.1 Any additional comments?

Thank you for completing this questionnaire and participating in this research!
Bibliography


line help affect user performance and attitudes? In Proceedings of the International


[16] Moon-Heum Cho. The Effects of Design Strategies for Promoting Students Self-
regulated Learning Skills on Students Self-Regulation and Achievements in Online-
Learning Environments. In Association for Educational Communications and Tech-


user models for just-in-time workplace training. In The Sixth International Conference

http://pact.cs.cmu.edu/.

[22] A. Corbett, B. Myers, S. Stevens, and K. Koedinger. ALPS: Active Learning and

in Skill Acquisition. Computer assisted instruction and intelligent tutoring systems:


Kumar. Recognizing Opportunities for Mixed-Initiative Interactions in Novice Pro-
gramming. In AAAI Fall Symposium on Mixed-Initiative Problem-Solving Assistants,
2005.

[27] D. Downey, S. Dumais, and E. Horvitz. Models of Searching and Browsing: Languages,


BIBLIOGRAPHY


Index

BlueJ IDE, 34
CEB, 9, 73
Code Construction, 9
Code Conventions, 8
Code Debugging, 10
Code Engineering, 8
Code Quality, 9
Collaboration, 11
Compile Event, 38
Complexity, 98
Constraint-Based Techniques, 28
Contexts, 18
CT-SEG, 7
Expert System, 13
Feedback, 20, 92
formative feedback, 39
FPA, 9
Graph Comparing Code Construction, 84
Graph Comparing Error Solving, 86
Graph Error Line Number versus Compile Time, 88
Graph Error Pattern, 58
Graph Error Solving versus Compile Time, 60
Graph Error Type Encountered, 60
Graph Files Referenced, 61
Graph Number Of Bytes(NO)B) versus Compile Time, 50
Graph Un/Successful Compiles, 55
Human Help, 18
Human-Computer Interactions, 17
Human-in-the-loop, 18
IDE, 11
Intelligent Tutoring Systems, 25
Interaction Ontology, 6, 37, 93
Jeliot, 11
Jena, 36
Knowledge Engineering, 13
LOC, 9
MICE Functional Architecture, 34, 93
MICE Programmer Style Viewer, 36
MICE Technical Architecture, 36
MICE tools, 6
Mixed-Initiative Interactions, 19, 27, 92
Model-Tracing Techniques, 28
Ontology, 14, 37
Planning, 8, 23
PMD, 9
Programmer Style Ontology, 93
programming context, 6
Programming Style, 1, 2, 7
Programming Styles, 49
Rules, 16, 92
Self-Evaluation, 24
Self-Monitoring, 22
Self-Motivation, 24
Self-Regulated Learning, 21
Software Engineering, 31
Spiral Model, 31
SRL (Self-Regulated Learning), 3
summative feedback, 1

Testing, 9
theory-oriented feedback, 5

Winne's SRL Model, 24

Zimmerman's SRL Model, 3, 20, 24