Toward Self-Explanation Based Intelligent Tutoring System

Lasang Jimba Tamang

Follow this and additional works at: https://digitalcommons.memphis.edu/etd

Recommended Citation
https://digitalcommons.memphis.edu/etd/3098
TOWARD SELF-EXPLANATION BASED INTELLIGENT TUTORING SYSTEM FOR CODE COMPREHENSION

by

Lasang Jimba Tamang

A Dissertation
Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctorate of Philosophy

Major: Computer Science

The University of Memphis
May 2023
Acknowledgements

First, I am immensely grateful to my advisor and dissertation committee chair, Dr. Vasile Rus, for all his support and guidance toward completing my Ph.D. Besides my Ph.D., I value most the traits his personality has inspired me to be: kind, helpful, and supportive to others.

I am also grateful to another committee member Dr. Scott Fleming, with whom I had the opportunity to collaborate on a research project related to this dissertation, for his support and suggestions. Furthermore, I thank other committee members, Dr. Deepak Venugopal and Dr. Xiaofei Zhang, for their helpful discussions and feedback.

Additionally, I like to acknowledge grant numbers 1822816 and 1934745 from National Science Foundation (NSF), which funded my Ph.D. and the research works, and the Institute for Intelligent Systems Student Organization (IISSO) for covering all my travel expenses related to research dissemination.

Further, I am thankful to all my lab colleagues, Dr. Nabin Maharjan, Dr. Dipesh Gautam, Dr. Rajendra Banjade, Dr. Zeyad Alshaikh, Priti Oli, Rabin Banjade, and Jeevan Chapagain. I am deeply indebted to Dr. Nabin Maharjan because of whom I joined this Ph.D. and for all his support and mentorship afterward. I also like to mention and thank Dr. Anant Bhusal and Sujit Shrestha for helping me to get into the Ph.D. program.

Lastly, I would like to thank my wife, Rojina Maharjan, for all her unconditional love and support during my Ph.D.
Dedication

To my dad Kulman Tamang, mom Chandra Maya Tamang, brother Raj Kumar Lama Tamang, and sister Bishnu Maya Tamang.
Abstract

Studies have shown that 30–40% of students fail or drop out of the introductory programming course. Furthermore, enrollment in CS programs has lately shown a significant increase, making it difficult to provide personalized attention. In this context, the work in this dissertation is an effort toward developing Self-Explanation Based Intelligent Tutoring System (ITS) for Code Comprehension, which offers one-to-one tutoring to enhance learners’ source code comprehension skills.

Self-Explanation Based ITS uses self-explanation as a learning strategy. Although self-explanation has shown a positive effect in different science domains such as biology, and math, it has been studied minimally for code comprehension; it seeks further investigations. Likewise, it uses questions as hints to scaffold students to elicit the code explanation correctly. Currently, such questions are authored manually and thus costly. Therefore, this dissertation aims to examine the effectiveness of self-explanation for code comprehension and explore the approaches for automatically generating questions for code comprehension. We conducted two randomized trial experiments and developed two approaches for this purpose.

Our first study investigated the effect of merely prompting to freely self-explain code. We found it helps to induce 31% learning gain. Then, the second study compares guided self-explanation with free self-explanation. The result shows that guided self-explanation outperformed by 29%, inducing students’ learning gain.

Next, we developed two systems to generate questions using each sentence in code explanation automatically. Our evaluation shows that generated questions are linguistically well-formed, pedagogically sound, and indistinguishable from human-generated questions. Finally, we formed a CodeQG dataset specific for code comprehension and trained a transformer in the dataset to automatically generate
questions using target concepts in code explanation. Our finding shows that the model not only generated a wide variety of impressive questions (BLEU: 89, ROUGE: 94, F1: 94.64), but the model’s performance improved almost triple by training on CodeQG compared to using SQuAD.

In this dissertation, we investigated and showed that Self-Explanation is effective for code comprehension. Then, we developed approaches for automatically generating questions for code comprehension. We also constructed a large dataset called CodeQG, a question-generation dataset specific to code comprehension.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Contents</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>List of Abbreviations</td>
<td>xi</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Motivation</td>
<td>1</td>
</tr>
<tr>
<td>Self-Explanation Based ITS</td>
<td>2</td>
</tr>
<tr>
<td>Scope</td>
<td>3</td>
</tr>
<tr>
<td>Research Gap</td>
<td>3</td>
</tr>
<tr>
<td>Research Challenges</td>
<td>5</td>
</tr>
<tr>
<td>Goal</td>
<td>6</td>
</tr>
<tr>
<td>Research Questions</td>
<td>6</td>
</tr>
<tr>
<td>Significance</td>
<td>6</td>
</tr>
<tr>
<td>Structure Overview</td>
<td>7</td>
</tr>
<tr>
<td>2 Background</td>
<td>9</td>
</tr>
<tr>
<td>Code Comprehension</td>
<td>9</td>
</tr>
<tr>
<td>Self-Explanation</td>
<td>12</td>
</tr>
<tr>
<td>Intelligent Tutoring Systems</td>
<td>13</td>
</tr>
<tr>
<td>Dialogue-Based Intelligent Tutoring Systems</td>
<td>14</td>
</tr>
<tr>
<td>3 Free Self-Explanation for Code Comprehension</td>
<td>16</td>
</tr>
<tr>
<td>Introduction</td>
<td>16</td>
</tr>
<tr>
<td>Background and Related Works</td>
<td>18</td>
</tr>
<tr>
<td>Experimental Setup</td>
<td>20</td>
</tr>
<tr>
<td>Materials</td>
<td>21</td>
</tr>
<tr>
<td>Participants</td>
<td>22</td>
</tr>
<tr>
<td>Procedure</td>
<td>22</td>
</tr>
<tr>
<td>Assessment</td>
<td>23</td>
</tr>
<tr>
<td>Results</td>
<td>23</td>
</tr>
<tr>
<td>Role of Self-Explanation</td>
<td>24</td>
</tr>
<tr>
<td>Nature of Self-Explanation</td>
<td>26</td>
</tr>
<tr>
<td>Discussion and Future Work</td>
<td>28</td>
</tr>
<tr>
<td>4 Free Self-Explanation vs. Guided Self-Explanation</td>
<td>31</td>
</tr>
<tr>
<td>Introduction</td>
<td>31</td>
</tr>
<tr>
<td>Related Works</td>
<td>35</td>
</tr>
<tr>
<td>Comparative Study</td>
<td>38</td>
</tr>
<tr>
<td>Group Design</td>
<td>39</td>
</tr>
<tr>
<td>Participants</td>
<td>41</td>
</tr>
<tr>
<td>Materials</td>
<td>41</td>
</tr>
</tbody>
</table>
5 Automatic Question Generation Using Each Sentence in Code Explanation

6 Automatic Question Generation Using Target Concepts in Code Explanation

7 Conclusion and Future Work
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>User Interface for Self-Explanation Based ITS for Code Comprehension</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Major building blocks of Self-Explanation Based ITS. The text in red indicates further studies are required in that field.</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Goal of this dissertation.</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Scatter plot for content word and learning gain.</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>Interface for Guided Self-Explanation</td>
<td>40</td>
</tr>
<tr>
<td>6.1</td>
<td>Illustration of what we are trying to achieve: Generate a question for each target concept in code explanation.</td>
<td>66</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Independent sample t-test result of pretest scores between Self-Explanation First and Prediction First groups.</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>Independent sample t-test result for the prediction score for self-explanation first and prediction first group.</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Independent sample t-test result between the learning gains of high vs. low prior knowledge groups.</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Summary statistics of self-explanation per task (all self-explanations from all participants).</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Examples of self-explanations.</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Independent Sample t-test result for learning gain between Low and High prior knowledge group in Self-Explanations</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Independent Sample t-test result for learning gain between High and Low prior knowledge group in Socratic Method</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Sample Short Questions generated by Human, Machine Noun QG and Machine Verb QG</td>
<td>57</td>
</tr>
<tr>
<td>5.2</td>
<td>Sample Gap-Fill Questions generated by Human, Machine Noun QG and Machine Verb QG. The first text inside [ ] is answer and the second text represents the quality of gap-fill questions given by human annotator.</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>Performance of Machine Noun QG, Machine Verb QG, and Human on Short-questions. SD= Standard Deviation.</td>
<td>59</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance of Machine Noun QG, Machine Verb QG, and Human on Gap-Fill Questions.</td>
<td>61</td>
</tr>
<tr>
<td>6.1</td>
<td>The number of instances and unique code count in a train, dev, test, and total for the CodeQG dataset.</td>
<td>69</td>
</tr>
<tr>
<td>6.2</td>
<td>The average, minimum, and maximum count of tokens in the CodeQG dataset.</td>
<td>70</td>
</tr>
</tbody>
</table>
6.3 Distribution of question types in CodeQG dataset.
6.4 Statistics showing the average, minimum, and maximum count of tokens for the SQUaAD dataset after preprocessing for our training purpose.
6.5 Model Type with their input source sequence and name of source dataset used for train, dev, and test purposes.
6.6 Performance of Different Model Types.
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITS</td>
<td>Intelligent Tutoring System</td>
</tr>
<tr>
<td>CSE</td>
<td>Computer Science Education</td>
</tr>
<tr>
<td>CS</td>
<td>Computer Science</td>
</tr>
<tr>
<td>IP</td>
<td>Introductory Programming</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Motivation

Code comprehension is the ability to understand a piece of code. Students learning computer programming spend significant time reading or reviewing someone else’s code (e.g., source code examples provided by a textbook or introduced by the instructor). Likewise, it is estimated that software professionals spend at least half their time analyzing software artifacts to comprehend computer source code. For example, during software maintenance, professionals utilize 70% of their time just to read and understand the code before they provide any solution (Boehm & Basili, 2001; Buse & Weimer, 2008; Rugaber, 2000). Hence, offering support to enhance learners’ source code comprehension skills could have lasting positive effects on their academic success and future professional careers.

However, helping learners master code comprehension is a challenging and open research problem in Computer Science Education (CSE). Studies have shown that 30–40% (or even higher) number of students fail or drop out from Introductory Programming (IP) courses (e.g., CS1 and CS2) in CS programs (Bennedsen & Caspersen, 2007, 2019; Watson & Li, 2014). Evidently, students struggle to learn code comprehension skills, which is unsurprising given that learning to code is complex (Caspersen & Bennedsen, 2007; Hanks, McDowell, Draper, & Krunjajic, 2004; Jenkins, 2002), for CS concepts are inherently complex (Morrison, Margulieux, & Guzdial, 2015). Hence, there is a need to support these students beyond classroom teaching and provide them with more personalized attention and instructions. Unfortunately, the problem is more intensifying for CS enrollment has significantly increased in recent years (Glassman & Shen, 2014; Guzdial, 2017; Zweben, 2008) and is only expected to skyrocket in the coming days due to the growing popularity of the CS program.
Ideally, a human tutor providing one-to-one tutoring is the best way for personalized instruction and support for students beyond the classroom. However, human tutors are costly, and subject-expert tutors are insufficient. In this context, the alternative solution is using a computer tutor called Intelligent Tutoring Systems (ITS), which can offer one-to-one tutoring to enhance learners’ source code comprehension skills. Such computer tutors are as effective as human tutors in inducing learning gain (Kulik & Fletcher, 2016; VanLehn, 2011). Although there are a few similar efforts (Johnson & Soloway, 1985; Lane & VanLehn, 2004; Soloway, Woolf, Rubin, & Barth, 1981; Woods & Warren, 1995) to build ITS to aid students in programming, these works are far behind adequate to develop ITS for code comprehension, and they never targeted JAVA code examples (most widely used in CS1 and CS2), which we intend to do.

In this context, the work in this dissertation is an effort towards developing Self-Explanation Based ITS for Code Comprehension (Rus et al., 2019). It is also one of the efforts of the CSEdPad (NSF, n.d) project, whose goal is to investigate and scaffold students’ mental models during computer programming tasks to improve learning, engagement, and retention.

**Self-Explanation Based ITS**

Self-Explanation Based ITS for code comprehension is ITS that can converse with students, in a natural language, like a human tutor, to provide one-to-one tutoring to support code comprehension skills. It is called self-Explanation based ITS, for the ITS uses self-explanation as a learning strategy. In this ITS, students are shown java code examples and encouraged to self-explain as much as possible what the code does. When the student’s self-explanation does not match or cover all ideal or expected code explanations, they are scaffolded to self-explain the missing parts. In doing so, questions are used as hints to scaffold the students’ self-explanation for the code. Hence, it is dialogue-based ITS helping students
master code comprehension by scaffolding to correctly elicit the code explanation themselves, offering feedback and hints (often in the form of short and gap-fill questions) as needed based on individual performance on the task. The figure 1.1 demonstrates a typical conversation between a student and a computer tutor in Self-Explanation Based ITS, where the learner is being offered support to comprehend the given code example as shown figure 1.1.

Framework: Self-Explanation Based ITS for Code Comprehension (Rus et al., 2019) is built by adapting the framework of DeepTutor (Rus, Stefanescu, Niraula, & Graesser, 2014), which is conversational ITS tested and shown effective in teaching physics (Rus, Niraula, & Banjade, 2015). Thus, it has all the major building blocks of Deeptutor: AI Components, Instructional Strategy, and Instructional Materials, as shown in figure 1.2. AI components are used to select the task and instructions adaptively based on student performance, make assessments of student answers, and provide diagnostic feedback. Like Deeptutor, it also uses Self-Explanation as an instructional strategy for tutoring. As instructional materials, while it also uses questions as hints to scaffold students, this ITS uses code examples, unlike problem descriptions in DeepTutor.

Scope

Among three major building blocks of Self-Explanation Based ITS for Code Comprehension, as shown in figure 1.2, the work in this dissertation focuses only on two blocks: Instructional Strategy and Instructional Materials.

Research Gap

While developing Self-explanation Based ITS for Code comprehension by adapting DeepTutor allows faster development time and thus reduced cost, it also inherits all traits from DeepTutor. Before we inherit everything from DeepTutor, some knowledge gaps need to be filled, and problems which need a better solution.

First of all, although the positive effect of self-explanation on learning has
Figure 1.1 User Interface for Self-Explanation Based ITS for Code Comprehension

<table>
<thead>
<tr>
<th>01</th>
<th>AI Components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Adaptive selection of tasks and instructions</td>
</tr>
<tr>
<td></td>
<td>• Model for student answer assessment</td>
</tr>
<tr>
<td></td>
<td>• Diagnostic feedback</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>02</th>
<th>Instructional Strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Self-Explanation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>03</th>
<th>Instructional Materials</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Content (Reading Passage, Code, etc.)</td>
</tr>
<tr>
<td></td>
<td>• Questions (Short, gap-fill questions, etc.)</td>
</tr>
</tbody>
</table>

Figure 1.2 Major building blocks of Self-Explanation Based ITS. The text in red indicates further studies are required in that field.
been well demonstrated in different science domains such as biology (Chi, De Leeuw, Chiu, & LaVancher, 1994), physics (Conati & Vanlehn, 2000), math (Aleven & Koedinger, 2002), the effectiveness of the strategy for code comprehension has been studied minimal, and thus further investigations are required.

Secondly, there is a problem with how questions used by ITS are produced. Questions are vital components that ITS uses to scaffold students. However, the current norm is that such questions are authored manually, which is time-consuming and thus costly. ITS community commonly believes authoring questions for instructional material of 1-hour tutoring takes 100-200 hours (Aleven, McLaren, Sewall, & Koedinger, 2009; Corbett, 2002); our team reports similar time working in DeepTutor and DeepCodeTutor Rus et al. (2019), despite authoring tools aid. In addition, the lack of experts in the science domain is always a problem. While there are some efforts in Automatic Question Generation (AQG) for educational purposes (Das, Majumder, Phadikar, & Sekh, 2021; Kurdi, Leo, Parsia, Sattler, & Al-Emari, 2020; Le, Kojiri, & Pinkwart, 2014; Umardand & Gaikwad, 2017), the recent survey by (Kurdi et al., 2020) found only 3 efforts in the programming domain. Therefore, there is a further need to explore approaches for automatically generating questions, used by ITS, for code comprehension.

**Research Challenges**

Based on our discussions in 1, the following are two research challenges that we look to address in this dissertation work.

1. Investigate the effectiveness of self-explanation as an instructional strategy for code comprehension.

2. Explore approaches for automatically generating questions, used by ITS, for code comprehension.
Goal

Toward our effort to develop Self-Explanation Based ITS for code comprehension (Rus et al., 2019), the figure 1.3 depicts our two aim in this dissertation, which helps to develop Self-Explanation Based ITS for Code Comprehension. The two aims can be listed as follows:

1. Examine the self-explanation learning strategy for code comprehension
2. Explore approaches for automatically generating questions for code comprehension.

Research Questions

The key research questions (RQ) that we seek to answer in this research work can be outlined as follows:

• **RQ1**: How effective is Free Self-Explanation (i.e., the most basic form of Self-Explanation) as a learning strategy for code comprehension?

• **RQ2**: How effective is the learning/teaching strategy Guided Self-Explanation compared to Free Self-Explanation for code comprehension?

• **RQ3**: How to automatically generate questions, used by ITS for scaffolding students’ self-explanation, using each sentence in code explanation?

• **RQ4**: How to automatically generate questions, used by ITS for scaffolding students’ self-explanation, using targeted concepts in code explanation?

Significance

This study will contribute to the body of knowledge on ITS for Code Comprehension by a) reporting the effectiveness of self-explanation as a learning strategy for code tutoring and b) developing approaches for automatically generating questions used by ITS for scaffolding students’ self-explanation. This help addresses
the existing insufficient research, as seen in the effort to develop ITS for code comprehension.

**Structure Overview**

In this Chapter 1, we introduced the context of the study. We presented our research goal, outlined our research questions, and argued the value of such research. We also discussed the limitations of our study.

In Chapter 2, we introduced terminologies or background knowledge to help a general audience understand our work; it is not part of actual research work.

In Chapter 3, we report the findings of the experimental study conducted to investigate the effect of prompting free self-explanation during code comprehension. In particular, we analyze the students’ self-explanation of code to understand a) the role of free self-explanation as a learning strategy and b) the nature of students’ self-explanation.

In Chapter 4, we detail the result of a randomized trial experiment that compared the effectiveness of two forms of self-explanation learning strategy for code comprehension: free self-explanation vs. guided self-explanation. In addition,
we also present the approach and the result for the student’s mental model during code comprehension using their self-explanation of code.

In Chapter 5, we present two systems, Machine Noun QG and Machine Verb QG, for automatically generating scaffolding questions using each sentence in code explanation. We describe the underlying approach for each system and evaluate the quality of automatically generated questions relative to human expert questions.

In chapter 6, we present another approach for automatically generating questions for code comprehension using targeted concepts in code explanations (code summary). We describe in detail the approach we have taken and how we constructed the CodeQG dataset used for the training model using our approach. We then discuss the performance of our model based on the automatic evaluation performed.

Finally, in chapter 7, we conclude our work by summarizing and reflecting on our research process and providing recommendations for future work.
Chapter 2

Background

Code Comprehension

As noted, we aim to develop intelligent tutoring systems that foster code comprehension and learning. What exactly is code comprehension?

In a straightforward and short definition, code comprehension is understanding a piece of code. The goal of understanding code is for the reader, in our case, the learner, to develop an accurate mental representation of what the code does - identifying the functional parts and how they relate to each other in order to provide the holistic, higher-level functionality of the code, i.e., how they serve the overall goal of the code, and how the functional parts are brought about using computational concepts and tools. This is our operational definition of code or program comprehension. It should be noted that, more broadly, code or program comprehension implies other things, i.e., if a reader understood a target code, then they should be able to predict the output of the target code for a given input value, they should be able to trace and keep track and update the program’s memory state, i.e., the value of all the data/variables after each statement is being executed, for given inputs, explain other characteristics of the code such as memory and time characteristics for various inputs, be able to change the code for new or updated requirements, etc.

Our operational definition is based on existing literature and our experience as Computer Science educators and researchers. According to prior literature, Schulte, Clear, Taherkhani, Busjahn, and Paterson (2010), “Comprehension is usually conceptualized as a process in which an individual constructs his or her own mental representation of the program.” That is, mental (or situation) models are critical components of source code comprehension theories (Brooks, 1983; Burkhardt, Détienne, & Wiedenbeck, 2002; Good, 1999; Pennington, 1987; Shaft,
Constructing accurate mental models of code is challenging, in particular, for novices.

Code comprehension, i.e., the ability to understand code, is critical for both learners and professionals. Indeed, students learning computer programming spend a significant portion of their time reading or reviewing someone else’s code (e.g., source code examples provided by a textbook or introduced by the instructor). Furthermore, it has been estimated that software professionals spend at least half of their time analyzing software artifacts to comprehend computer source code. Reading code is the most time-consuming activity during software maintenance, consuming 70% of the total lifecycle cost of a software product (Boehm & Basili, 2001; Buse & Weimer, 2008; Rugaber, 2000). O'Brien (2003) notes that source code comprehension is required when a programmer maintains, reuses, migrates, reengineers, or enhances software systems. Therefore, offering support to enhance learners’ source code comprehension skills could have lasting positive effects on their academic success and future professional careers.

It is well known that a significant challenge in CS education is the difficulty novice programmers face with constructing accurate mental models during key learning activities, such as source code comprehension (Margulieux, Guzdial, & Catrambone, 2012; Pennington, 1987; Ramalingam, LaBelle, & Wiedenbeck, 2004; Soloway & Ehrlich, 1984). This challenge is not surprising given that constructing mental representations is considered a higher-level comprehension skill, typically engendering a high cognitive load (Graesser & McNamara, 2011; Kintsch & Walter Kintsch, 1998; Snow, 2002; Zwaan, Radvansky, Hilliard, & Curiel, 1998). Actually, it is well accepted that there is a modality-independent higher-level skill of comprehension that involves constructing situation models (Zwaan et al., 1998). This is based on research that showed that subjects arrive at similar understandings
(although not identical) when presented with textual, visual, and audio descriptions of a situation.

The importance of building accurate mental models during learning tasks has been well established for decades in domains like science (Boehm & Basili, 2001; Chi, Feltovich, & Glaser, 1981; de Jong & Ferguson-Hessler, 1991; DiSessa, 1993; Nathan, Kintsch, & Young, 1992) as well as in CS education (Margulieux et al., 2012; Pennington, 1987; Ramalingam et al., 2004; Soloway & Ehrlich, 1984). Nevertheless, further research is needed to fully understand what factors can mediate the construction of accurate mental models and learning and build effective instructional interventions to monitor and scaffold learners’ comprehension and learning processes. The intervention can be instructor-driven or computer-driven.

We want to emphasize one more important point of our work. The focus is on code comprehension for educational purposes, so the input is just source code (no additional items such as documentation). To this end, our work relies on and contributes primarily to what Schulte et al. (2010) calls an educational model of code/program comprehension, i.e., a model of learning to read and understand programs. That is, our focus is on code comprehension as opposed to program comprehension, which may involve the target code items such as documentation, diagrams, and representations of domain knowledge (Robson, Bennett, Cornelius, & Munro, 1991). Program comprehension seems to be a more appropriate task for larger programs, which are found in typical professional software development contexts and in which case just using the code to understand the large software system may be atypical. Instructional code examples used in standard intro-to-programming courses, our focus, are short, making them more amenable to code reading tasks, i.e., understanding them on the basis of the code itself.
Self-Explanation

Self-explanation theories indicate that students who engage in self-explanations, i.e., explaining the target material to themselves, while learning are better learners, i.e., learn more deeply and show the highest learning gains. Self-explanation’s effectiveness for learning is attributed to its constructive nature, e.g., it activates several cognitive processes such as generating inferences to fill in missing information and integrating new information with prior knowledge, and its meaningfulness for the learner, i.e., self-explanations are self-directed and self-generated making the learning and target knowledge more personally meaningful, in contrast to explaining the target content to others (Roy & Chi, 2005).

The positive effect of self-explanation on learning has been demonstrated in different science domains such as biology (Chi et al., 1994), physics (Conati & Vanlehn, 2000), math (Aleven & Koedinger, 2002), and programming (Bielaczyc, Pirolli, & Brown, 1995; Tamang, Alshaikh, Ait-Khayi, & Rus, 2020). A series of studies (Bielaczyc et al., 1995; Pirolli & Recker, 1994; Recker & Pirolli, 1990) found that self-explanations help to learn Lisp programming concepts. They found that skill improvement had a strong correlation with the amount of self-explanation generated. Two other studies with undergraduate students (Rezel, 2003) and high school students (Alhassan, 2017) found that students who used a self-explanation strategy while studying worked-out examples were more successful at a program construction task (Visual Basic) compared to those who did not apply the strategy. The effectiveness of the self-explanation in programming was also studied for SQL (Yuasa, 1994), JavaScript (Kwon & Jonassen, 2011), HTML (Kwon, Kumalasari, & Howland, 2011) and assembly language (Hung, 2012). Study (Bielaczyc et al., 1995) showed that university students who underwent explicit training on strategies of self-explanation and self-regulation outperformed students in a control group (no explicit training) on problem-solving performance. However, all these studies in the
programming domain have no studies with JAVA code examples, the focus of our work, which is popularly used in Introductory Programming courses for teaching code.

There are different ways to elicit self-explanations which result in different types of self-explanations such as spontaneous self-explanations (no prompting), free or open-ended self-explanations (simple prompting to self-explain), guided (a series of guiding questions are triggered to help student self-explanations), and scaffolded self-explanations (in this case students are encouraged to self-explain as much as possible by themselves and offered support in the form of hints when floundering). Other forms of self-explanations have been tried, such as “complete given self-explanations (fill in the blank self-explanations)” (Kwon et al., 2011) and “select a self-explanation/menu-based self-explanations” (Aleven & Koedinger, 2002; Fabic, Mitrovic, & Neshatian, 2019) which one may argue are not true self-explanations as the learner does not generate the explanation, i.e., the ‘self’ part of the ‘self-explanation’ is missing. Furthermore, self-explanation prompts can emphasize various aspects of self-explanations resulting in justification-based self-explanation prompts (Conati & Vanlehn, 2000) or meta-cognitive self-explanation prompts (Chi et al., 1994).

In our work, we explore two types of self-explanations for code comprehension: free elf-explanations and guided self-explanations.

**Intelligent Tutoring Systems**

One-on-one human tutoring is one of the most effective solutions to instruction and learning that has attracted the attention of many for decades. Encouraged by the effectiveness of one-on-one human tutoring (Bloom, 1984), computer tutors that mimic human tutors have been successfully built with the hope that a computer tutor could be available to every child or learner of any age with access to a computer (Rus, D’Mello, Hu, & Graesser, 2013).
How effective are state-of-the-art ITSs at inducing learning gains in students? An extensive review of tutoring research by VanLehn (2011) showed that computer tutors are as effective as human tutors. VanLehn (2011) reviewed studies published between 1975 and 2010 that compared the effectiveness of human tutoring, computer-based tutoring, and no tutoring. The conclusion was that the effectiveness of human tutoring is not as high as it was originally believed (effect size $d = 2.0$) but much lower ($d = 0.79$). The effectiveness of computer tutors ($d = 0.78$) was found to be as high as the effectiveness of human tutors. So, there is something about the one-on-one connection that is critical, whether the student communicates with humans or computers.

Graesser, Person, and Magliano (1995) argued that the remedial part of tutorial interaction in which tutor and tutee collaboratively improve an initial answer to a problem is the primary advantage of tutoring over classroom instruction. Furthermore, one-on-one instruction has the advantage of engaging most students’ attention and interest as opposed to other forms of instruction such as lecturing/monologue in which the student may or may not choose to pay attention (VanLehn et al., 2007).

**Dialogue-Based Intelligent Tutoring Systems**

Intelligent Tutoring Systems (ITSs) with conversational dialogue form a special category of ITSs. The development of conversational ITSs is driven by explanation-based socio-constructivist theories of learning and the collaborative constructive activities that occur during human tutoring (Rus et al., 2013). Based on constructivist theories of learning, learners construct their own knowledge. Accordingly, this kind of ITS, such as DeepTutor (Rus et al., 2014), monitors students’ learning process and encourages the learner to construct their own self-explanation, only intervening when the learner struggles. Based on socio-cultural theories of learning (Vygotsky, 1978), learners learn best from a
significantly more knowledgeable other (SKO), which in our case is the tutoring system.

Conversational ITSs have several advantages over other types of ITSs. They encourage deep learning as students are required to explain their reasoning and reflect on their basic approach to solving a problem. Such conceptual reasoning is more challenging and beneficial than the mechanical application of mathematical formulas (Hestenes, Wells, & Swackhamer, 1992). Furthermore, conversational ITSs have the potential to give students the opportunity to learn the language of scientists, an important goal in science literacy. A student associated with a more shallow understanding of a science topic uses more informal language as opposed to more scientific accounts (Mohan, Chen, & Anderson, 2009).
Chapter 3

Free Self-Explanation for Code Comprehension

Introduction

Computer Science (CS) education is critical in today’s world, where computing skills, such as computer programming, have become an integral part of many disciplines, including science, math, engineering, and technology. Although such skills are in high demand, and the number of aspiring CS students is encouraging, a large gap between the supply of CS graduates and demand persists. For example, college CS programs suffer from high attrition rates (30-40%, or even higher) in introductory CS courses (e.g., CS1 and CS2) (Beaubouef & Mason, 2005; Guzdial & Soloway, 2002; Wilson, Sudol, Stephenson, & Stehlik, 2010).

One reason for the high attrition rates in CS1 and CS2 courses is the inherent complexity of CS concepts and tasks (Du Boulay, 1986; Morrison et al., 2015). Programming is a highly complex process involving a multitude of cognitive activities and mental representations related to problem understanding, programming methods, program design, program comprehension, change planning, debugging, and the programming environment. Morrison et al. (2015) argue that using textual languages (to name and keep track of variables and handle related processes) while at the same time understanding and controlling an external agent (i.e., the computer) involves a level of complexity not seen in science, math, or engineering. Thus, it is unsurprising that many students in introductory programming courses feel overwhelmed.

This work is part of a project whose aims are to develop effective and engaging instructional interventions to improve comprehension and learning in introductory CS courses at the college level, reduce attrition rates and increase retention, and ultimately produce more and better-trained graduates. Furthermore, the plan is to incorporate this effective and engaging intervention in advanced
education technologies such as intelligent tutoring systems (ITSs), such as, Rus et al. (2019). The result will be a win-win-win situation for aspiring students, CS programs and their organizations, and the overall economy.

One of the working hypotheses of our project is that instructional strategies such as eliciting self-explanations will result in more accurate mental models, which in turn will positively impact comprehension, learning, self-efficacy, and retention (Best, Rowe, Ozuru, & McNamara, 2005; Chi et al., 1994; Ramalingam et al., 2004). The positive role of self-explanations is well documented for science learning but less so for computer programming learning. Therefore, our work fills this gap by answering the following broad research questions:

- **Role of Self-Explanation**: Does self-explanation help in learning computer programming? Is there any relationship between the volume of self-explanation generated by learners and their learning gains? Does self-explanation as a teaching strategy have a discrepancy in teaching low and high-prior knowledge students?

- **Nature of Self-Explanation**: How exactly do students explain source code? Are there any peculiarities and significant differences compared to self-explanations of, say, narrative or scientific texts?

Understanding the role of self-explanation during source code comprehension could provide an effective instructional strategy for addressing the high-attrition rates in introductory computer programming courses. Additionally, studying the nature of self-explanations during source code comprehension instructional activities will enable us to understand the challenges that need to be addressed in order to develop methods to assess student responses during such code comprehension activities automatically. This, in turn, will enable the development of intelligent tutoring systems that could implement self-explanation elicitation strategies and
automatically evaluate students’ responses based on which tailored feedback and support could be provided to each learner. The preliminary results presented in this paper are the first step toward understanding the effectiveness and the nature of self-explanations students generate during source code comprehension activities and building fully automated intelligent tutoring systems to help students in introductory to computer programming courses.

**Background and Related Works**

Self-explanations are learner-generated explanations of learning materials (Crippen & Earl, 2004; McNamara & Magliano, 2009; Roy & Chi, 2005; Van Merriënboer & Sluijsmans, 2009). Self-explanations usually include meta-cognitive statements, e.g., learners’ judgment of their level of understanding and inferences based on the information from the learning materials and learners’ knowledge. According to Roy and Chi (2005), several cognitive mechanisms are involved: generating inferences to fill in missing information, integrating information within the learning materials, integrating new information with prior knowledge, and monitoring and repairing faulty knowledge. The mix of self-assessment and inference leads to improved understanding, more coherent mental models, and better learning compared to activities of learning where students, for instance, just read the instructional materials.

There are three major types of self-explanations: spoken or thinking out loudly, typed or written down, and silent reflection. The first two types are widely used. In the thinking out loudly type of self-explanations, the learner thinks aloud about the instructional materials presented to them (McNamara, 2009). This type of self-explanation has been found to benefit proficient readers in general compared to less proficient readers (Muñoz, Magliano, Sheridan, & McNamara, 2006). For written self-explanation, learners write down their thoughts while engaging with particular learning material, such as reading scientific texts to understand and learn
target concepts or while trying to solve a problem (Muñoz et al., 2006). This form of self-explanation has been more suitable for learners who have difficulty with demanding reading tasks such as the reading of scientific texts that requires a higher cognitive load (as opposed to less demanding reading tasks such as reading narrative texts). In our case, the learning material is computer code which we can argue that it is more demanding than reading scientific texts. Indeed, Computer Science is considered even higher than science and math when it comes to cognitive load. For this reason, that understanding source code requires a high cognitive load; we opted for the written-down type of self-explanations. Importantly, written or typed self-explanations were shown to enable readers to make more inferences, e.g., bridging inferences that link a target instructional material to prior knowledge, as opposed to text-bound processes such as paraphrases (Muñoz et al., 2006). In particular, when typing their explanations of science texts, less skilled readers were more inclined to make bridging inferences than speaking self-explanations. Typing seems to afford readers more time to reflect and access and express their thoughts. Our work here contributes to this line of research by exploring the role of typed explanations for a novel task: source code comprehension.

Even though self-explanation is found to improve the construction of mental models (Chi et al., 1994; Ramalingam et al., 2004), very few studies have been conducted to investigate its effectiveness in doing so. A series of studies (Bielaczyc et al., 1995; Pirolli & Recker, 1994; Recker & Pirolli, 1990) found that self-explanations help learning Lisp programming concepts (their experimental population was undergraduate students). While they found that skill improvement had a strong correlation with the amount of self-explanation generated, they also noted that the type of explanation also accounts for the performance improvement: explanations of high-performing students were much more structured into goal-based episodes compared to those of low-performing students. This suggests
that analyzing in more depth the nature of self-explanations could help us better understand their impact on performance and how that impact is mediated by other factors such as students’ prior knowledge. In the study, we investigated both the effectiveness of self-explanations and the nature of students’ self-explanations.

Study with undergraduate (Rezel, 2003) and high school (Alhassan, 2017) students found that students who used self-explanations while studying worked-out examples were more successful at a program construction task (Visual Basic) compared to those who did not apply it. Further studies for JavaScript (Kwon & Jonassen, 2011), HTML (Kwon et al., 2011) and assembly language (Hung, 2012) also found that self-explanations helped in learning programming. Our work contributes to this line of research on the role of self-explanation in learning JAVA programming. Furthermore, most notably, all programming languages studied in the past are procedural languages, while we study JAVA, Object-Oriented Programming Language. To the best of our knowledge, there are no studies on the role of self-explanations in learning Java.

**Experimental Setup**

Our goal is to investigate self-explanation as a learning strategy for code comprehension, particularly the role of self-explanations and the nature of students’ self-explanations during code comprehension. For this reason, our experiment was designed to collect self-explanations from all participants. On the other hand, to study the effectiveness of self-explanations, some participants needed to work on code examples with self-explanation and others without it to see the effect of the strategy on students’ learning. We randomly assigned participants to one of the following two experimental groups to balance both needs. Participants in both of these groups interacted with an online system.

**Self-Explanation First:** First, participants in this group were shown 4 Java code examples, one at a time. For each code example, they were prompted to
predict the output and self-explain their understanding of the code in writing.

Then, they were shown another 4 code examples but were only asked to predict the output of the code examples (no need to self-explain the code).

**Prediction First**: Participants in this group followed the same procedure as the participants in the Self-Explanation First condition, except the order of the prediction and self-explanation tasks was switched. First, participants were asked to predict only the output of 4 code examples, followed by four other 4 code examples, for which they were also asked to predict output and self-explain their understanding of the code.

It should be noted that we focus on free self-explanations instead of supported/scaffolded self-explanations. We encourage students to self-explain while reading code examples without further support, e.g., we do not assess their self-explanations and do not provide feedback and hints in case any misconceptions are detected, or the self-explanations are vague or incomplete. We opted for the free self-explanation because we wanted to explore the nature of students’ freely generated explanations, i.e., when they are encouraged to self-explain without much guidance.

**Materials**

All participants first provided answers to a background questionnaire and then took a pretest (6 code examples for which they had to predict the output). After that, each participant was randomly assigned to one of the two experimental groups described above. All participants took a posttest (predict the output of 6 Java code examples). The pretest and posttest were not identical but equivalent in terms of concepts tested and difficulty level. The posttest was created from the pretest just by altering variables values or by a minimal alternation of certain parts of the code, such as the condition of `while` or `for` loops. The main programming concepts covered by the experiment were: operator precedence, nested `if – else`,
for loops, while loops, arrays, creating objects and using their methods. Each of these concepts was present in the code examples used in the pretest, posttest, and experimental tasks.

Participants

A total of 26 college students (7 female and 19 male) from an urban university in Asia participated in the experiment. The participants were briefed about the experiment’s goal, including the fact that they could solidify their programming skills by participating in the experiment. The participants were all volunteers and received a small compensation for their participation in the experiment. All participants were in the fourth semester of an undergraduate program in computer science and had undertaken the same courses in their prior semesters. In addition, these participants have a fair understanding of computer programming concepts as they took C and C++ courses in prior semesters. However, they have not taken the JAVA course yet. Participants were randomly assigned to one of the two experimental groups using a group-balancing approach (half of the participants were assigned to one experimental condition and half to the other).

Procedure

The experiment was conducted in a computer lab in the presence of two experimenters. All the materials were shown using a web-based system via browser. They were only allowed to ask questions related to the experimental procedure and the use of the web-based system. The participants were first debriefed about the purpose and nature of the experiment and given a consent form. Upon their agreement, they were shown the background survey followed by pretests. Then, they worked on the main part of the experiment that showed them prediction tasks and self-explanation tasks in the order corresponding to the experimental group in which they were randomly assigned. Finally, they worked on the posttest.
Participants could see all pretest and posttest questions while working on the corresponding sections. However, they were shown the prediction or self-explanation experimental tasks one at a time; they could proceed to the next task only after submitting the current task’s answer. All participants’ responses and interactions were automatically logged for posthoc analysis.

Assessment

Each question/task was scored 1 if the answer was correct; otherwise, 0. This means the maximum score was 6 for the pretest, 4 for the self-explanation tasks, 4 for the prediction tasks, and 6 for the posttest. For each participant, we also calculated the learning gain score as suggested by Marx and Cummings (2007). If \( \text{posttest} > \text{pretest} \), \( \text{gain} = (\text{posttest} - \text{pretest})/(6 - \text{pretest}) \). If \( \text{posttest} < \text{pretest} \), \( \text{gain} = (\text{posttest} - \text{pretest})/\text{pretest} \). If \( \text{posttest} = \text{pretest} = 0 \) or 6, drop the cases. If \( \text{posttest} = \text{pretest} \), \( \text{gain} = 0 \).

Results

Out of 26 participants, we dropped data from 3 participants because they had a perfect score in both pretest and posttest. Overall, participants had an average of 1.7 (SD/stddev = 1.5) years of programming experience, 0.07 (SD = 0.31) years professional programming experience, a score of 3.3/6 (SD = 1.9) in pretest, and 3.87/6 (SD = 1.63) in postest. We also computed their scores for the main experimental tasks. The average score was 2.2 (SD = 1.53) for the 4 prediction tasks, 1.61 (SD = 1.37) and a learning gain score of 0.31 (SD = 0.42), calculated as suggested by Marx and Cummings (2007). More detailed analyses are presented below. All the t-tests applied below in our study met the standard assumptions of t-test (continuous scale for the dependent variable, random sampling, independence of observations, normal distribution, and homogeneity of variance).
Role of Self-Explanation

The goal of our study was to test the hypothesis that self-explanation helps both program comprehension and learning. Furthermore, we investigated whether the more participants self-explain, the more they learn or not.

**Does Self-Explanation enhance the learning of core computer programming concepts?** To answer this question, we first check the equivalency between Prediction First and Self Explanation First group in terms of their prior knowledge, i.e., pretest score, by applying an independent sample t-test. The result in table 3.1 shows that groups are equivalent. To answer our question, we performed an independent-sample t-test between Self-Explanation First and Prediction First groups to compare their prediction scores. The results in table 3.2 shows that there is a statistically significant difference in prediction score for self-explanation first (M=2.54, SD=1.67) and prediction first group (M=1.3, SD=1.06; t(20)=1.62, p=0.042). The magnitude of the difference in the means (mean difference = 1.24, 95% confidence interval: 0.05 to 2.43) is large (Cohen’s d = 0.9) as suggested by Sawilowsky (2009).

Participants in the Self-Explanation First group performed \((2.54-1.3)/4\times100=31\%\) better, as measured by prediction score, than participants in the Prediction First group. An analysis of covariance (ANCOVA) with the experimental condition as the grouping/factor variable and pretest score as covariate also indicated that there was a significant difference \((F(1, 20)=5.093, p=0.035)\) in mean learning gains between the Prediction First and Self-Explanation First groups while adjusting for pretest scores (prior knowledge). The effect size is small \((0.20)\). The order of the self explanation can explain 20% of variance in learning gain. The better performance of the former group can be attributed to the Self-Explanation learning strategy. The participants in Self-Explanation group first self-explained four tasks before working on prediction tasks. The use of self-explanations had both
Table 3.1 Independent sample t-test result of pretest scores between Self-Explanation First and Prediction First groups.

<table>
<thead>
<tr>
<th>Group</th>
<th>N</th>
<th>Mean</th>
<th>SD</th>
<th>t-val</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self Explanation</td>
<td>13</td>
<td>2.9</td>
<td>1.91</td>
<td>-1.31</td>
<td>0.20</td>
</tr>
<tr>
<td>Prediction</td>
<td>10</td>
<td>3.9</td>
<td>1.92</td>
<td>-1.31</td>
<td>0.20</td>
</tr>
</tbody>
</table>

Table 3.2 Independent sample t-test result for the prediction score for self-explanation first and prediction first group.

<table>
<thead>
<tr>
<th>Group</th>
<th>N</th>
<th>Mean</th>
<th>SD</th>
<th>t-val</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self Explanation</td>
<td>13</td>
<td>2.54</td>
<td>1.67</td>
<td>2.17</td>
<td>0.042</td>
</tr>
<tr>
<td>Prediction</td>
<td>10</td>
<td>1.30</td>
<td>1.06</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

improved their learning and comprehension of source code with positive effects on the prediction tasks. On the other hand, the Prediction First group only used the self-explanation strategy after working on the prediction tasks.

*Is there a relationship between the amount of self-explanation generated and learning gains?* For this question, we first analyzed the relationship between the count of content words (the most informative words such as nouns, verbs, adjectives, and adverbs) and learning gains. The scatter plot in Fig. 3.1 indicates a positive relationship between content word count and learning gains. The relationship between self-explanation (as measured by content word count) and learning gains was further investigated using a Pearson product-moment correlation coefficient. We found a strong, positive correlation between them, \( r = 0.62 \), \( n = 23 \), \( p = 0.001 \), with a higher count of content words associated with higher learning gains. Content word production helps explain nearly 38 percent of the variance in learning gains.

*Is self-explanations biased towards students’ prior knowledge?* We divided the participants into two groups, high and low prior knowledge, using the mean pretest score of 3.3 as the cut-off value. Table 3.3 shows that the difference in learning gain is not statistically significant for the groups; thus, we did not find any such evidence of a discrepancy in the strategy based on students’ prior knowledge.
Figure 3.1 Scatter plot for content word and learning gain.

<table>
<thead>
<tr>
<th>Group</th>
<th>N</th>
<th>Mean</th>
<th>SD</th>
<th>t-val</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Prior</td>
<td>11</td>
<td>0.36</td>
<td>0.58</td>
<td>0.49</td>
<td>0.63</td>
</tr>
<tr>
<td>Low Prior</td>
<td>12</td>
<td>0.27</td>
<td>0.21</td>
<td>0.49</td>
<td>0.63</td>
</tr>
</tbody>
</table>

Table 3.3 Independent sample t-test result between the learning gains of high vs. low prior knowledge groups.

Nature of Self-Explanation

We found that the nature of self-explanation in program comprehension varies a lot in terms of the amount of self-explanation generated by participants and how they talk about the code examples shown. As indicated in table 3.4, participants, on average, generated 3.52 sentences, 61.35 words, and 34.52 content words per self-explanation task. The amount of self-explanation generated varies greatly; while some participants did not self-explain at all (0 sent count - they only predicted the output), others wrote 17 sentences, 148 content words, and 296 words.

Indeed, self-explanations vary widely. For instance, the first self-explanation in Table 3.5 in row 1 is very detailed and long as opposed to the one shown in row 4, which simply explains the execution of the code using a mathematical expression. In the latter case, the student succinctly indicates that time is equal to 10 and therefore less than 12, which in turn (this is implied) should lead to the execution of
<table>
<thead>
<tr>
<th>ID</th>
<th>Self-Explanation Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Factorial of 4 is 24. In main we see that num is 4. We also have a method named factorial that takes in an int number and multiplies it by itself minus one again and again until n is one. That number that is multiplied is what factorial returns. In our case we sent to it the number 4. 4 is not 1, so 4 is multiplied by a number that is returned from factorial. So, 4 multiplied by (4-1) multiplied by (3-1) multiplied by 2-1 multiplied by 1. Going backwards, 1 multiplied by 2 multiplied by 3 multiplied by 4 results in 24</td>
</tr>
<tr>
<td>2</td>
<td>The for loop keeps iterating as we add one to i until i is greater than 10. Once it is, we exit and print out the result of sum divided by i which is 64 divided by 11 which is 5 (because the variables are ints)</td>
</tr>
<tr>
<td>3</td>
<td>n(n 1)/2 = in this case, 55, and we divide it by 10, the last value of i. 55/10 is 5.5, but since these are ints, the decimal is truncated off and we are left</td>
</tr>
<tr>
<td>4</td>
<td>time=10&lt;12</td>
</tr>
<tr>
<td>5</td>
<td>0 plus 1 plus 2 plus 3 plus 4... plus 10 equals 45.int 45 divided by int 11 is int 4.</td>
</tr>
</tbody>
</table>

Table 3.4 Summary statistics of self-explanation per task (all self-explanations from all participants).

<table>
<thead>
<tr>
<th></th>
<th>Word Count</th>
<th>Content Word Count</th>
<th>Sent Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>61.35</td>
<td>34.52</td>
<td>3.52</td>
</tr>
<tr>
<td>SD</td>
<td>74.2</td>
<td>37.92</td>
<td>4.48</td>
</tr>
<tr>
<td>Max</td>
<td>296</td>
<td>148</td>
<td>17</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.5 Examples of self-explanations.

a certain branch of the if statement in the code. In other words, the explanation is correct but incomplete if only the explicit parts should be accounted for. However, since the student correctly predicted the output, the implied parts can be assumed to be correctly considered (but not articulated verbally) by the student.

The second explanation and the fifth indicate students who understand the code but make simple computation errors and therefore incorrectly predict the output. Both students compute the sum of the integers from 1 to 10 as 64 and 45, respectively, instead of the correct value of 55, even though they semantically understand the loop’s purpose. The two explanations, which are about the same
code example, reveal two different ways to explain loops. Explanation 2 in Table 2 describes the loop succinctly, while explanation 5 describes the loop in its “unwrapped” or sequence-of-iterations form.

Explanation 3 is also about the same code example as explanations 2 and 5, and this is a clear example of students not paying close attention to the code. The value of the iteration variable i is 11 after the loop ends and not 10 (the loop condition is \( i \leq 10 \)). While the student understands that the main idea of the loop is to add numbers, it does not pay attention to the details of the loop.

All shown explanations in Table 3.5 and all 92 explanations we collected from the 23 students (4 explanation tasks per participant), exhibit a combination of language, code references, and mathematical expressions. This has significant implications for any automated methods to assess such self-explanations and, therefore, any intelligent tutoring system that uses scaffolded self-explanation as a key instructional strategy. Such systems need to automatically evaluate self-explanations in order to provide adequate feedback. Novel assessment methods will need to be developed as we are not aware of any such methods to automatically evaluate self-explanations generated in the context of source code comprehension activities like those shown in Table 3.5.

**Discussion and Future Work**

Our preliminary work has demonstrated that self-explanation is a promising learning strategy that could help students improve their source code comprehension skills and learn complex programming concepts. Indeed, our experiments with free self-explanation prompting support the positive role of self-explanation for source code comprehension and learning. We also found no evidence of biases in the strategy with students’ prior knowledge. Like Recker and Piroli (1990), Piroli and Recker (1994) and Bielaczyc et al. (1995), we also found that learning gain is strongly correlated with the amount of generated self-explanation. We also found
that learners self-explain in different ways, providing different details. Furthermore, the self-explanations include references to code, mathematical expressions, and natural language, which poses new challenges for automatically assessing these self-explanations.

Because of the small sample size (n=23) and sample population from Asia, the finding of this study may not be generalized to a student in the USA or other parts of the world. Students from two continents might perceive knowledge under the same learning strategy differently, which is not considered during this. Furthermore, this study only covered a few basic concepts of JAVA, such as operator precedence, nested *if−else*, *for* loops, *while* loops, arrays, creating objects and using their methods; thus, the study should be further investigated in the context of more examples and broader topics if we are to claim about entire JAVA learning using self-explanations.

There are several future work directions we envision. First of all, a study that would compare Free Self-Explanations vs. Guided Self-Explanations (the form of self-explanation that uses a sequence of questions to guide students) for source code comprehension tasks is essential to investigate if ITS effectively incorporate self-explanation or not. No such studies have been done before, and we plan to do this in our next work. Furthermore, a source code reading skill assessment instrument is needed to assess learners’ source code comprehension level. We also plan to run an experiment where students are shown code examples one line at a time and prompted to explain their thoughts on any new line of code. Exploring all these new directions in the future will further reveal details about source code comprehension processes and create necessary instruments to assess code comprehension skills. Finally, we plan to annotate self-explanations and develop a machine-learning model for the automatic grading of self-explanations. Self-explanations in the context of programming varies a lot, so it is not possible to
compare them with reference answer and apply the similarity method (Maharjan, Banjade, Gautam, Tamang, & Rus, 2017); thus, it seeks a novel approach to assessment.
Chapter 4

Free Self-Explanation vs. Guided Self-Explanation

Introduction

A key challenge in undergraduate Computer Science (CS) programs is high attrition rates. At a global level, a recent study by Bennedsen and Caspersen (2019) indicated that the mean worldwide failure rate in these courses in 2019 was 33%. In the United States, Introductory CS courses (e.g., CS1 and CS2) often have attrition rates of 30-40% or even higher (Beaubouef & Mason, 2005; Guzdial & Soloway, 2002; Wilson, Sudol, Stephenson, & Stehlik, 2010). On the contrary, local and global demand for skilled programmers has increased substantially and is expected to grow even more, e.g., the United States Bureau of Labor and Statistics predicts software developer jobs will grow at the rate of 21% until 2028 (BLS, n.d.). Thus, it is imperative to develop and explore effective instructional strategies that will help students better understand the programming concepts and achieve higher learning gain, which will improve their chances of successfully completing CS1 and CS2 courses.

To this end, CS education researchers have spent considerable effort exploring effective instructional interventions that facilitate students learning programming (Hosseini et al., 2020; Hosseini, Akhuseyinoglu, Petersen, Schunn, & Brusilovsky, 2018; Pears et al., 2007; Robins, Rountree, & Rountree, 2003; Rus et al., 2019; Zhang, Zhang, Stafford, & Zhang, 2019). A large number of automated tool papers are frequently published for instruction support. For example, tools that perform Automated Program Repair (Monperrus, 2020). Yi, Ahmed, Karkare, Tan, and Roychoudhury (2017) found that while the graders seem to gain benefits from automatically generated repairs shown as a hint, novice students do not seem to know how to make use of it effectively. Ahmed, Srivastava, Sindhgatta, and Karkare (2020) found that the advantage of automatic feedback over manual feedback to
resolve errors in code is primarily logistical and not conceptual; the performance benefit seen during lab assignments disappeared during exams wherein feedback of any kind was withdrawn. Thus, a tool that helps novice programmer has to be developed (as mostly CS1 and CS2 students are novice programmers), and any strategy used in such a tool should support long-term mastery of a concept or better mental model construction (rather than just immediate support for bug fixing). In order to do so, we need to understand the nature of programming and why teaching programming to novices remains a challenging task (Kortsarts, Akhuseyinoglu, Barria-Pineda, & Brusilovsky, 2020), which are discussed below.

The primary reason that a large number of students fail in CS1 and CS2 courses is the inherent complexity of CS concepts themselves (Morrison et al., 2015). Programming is considered a complex cognitive activity where a student has to simultaneously build and apply several higher-order cognitive skills for solving a particular problem (Robins et al., 2003). First encounters between a learner and a computer system are nothing short of a “shock” (du Boulay, 2013) as programming is a highly complex process involving a multitude of cognitive activities and mental representations related to problem understanding, programming methods, program design, program comprehension, change planning, debugging, and the programming environment. CS concepts are considered more complex than those of other fields traditionally considered challenging, such as mathematics. Thus, it is not surprising that many students in introductory programming courses feel overwhelmed.

The secondary hindrance in students’ path towards mastery or learning of programming concepts is difficulties constructing accurate mental models (Milne & Rowe, 2002). Indeed, a major challenge in CS education is the difficulty novice programmers face with constructing accurate mental models during key learning activities, such as source code comprehension (Margulieux et al., 2012; Pennington, 1987; Ramalingam et al., 2004; Soloway & Ehrlich, 1984). This challenge is not
surprising given that constructing mental representations is considered a higher-level skill of comprehension, typically engendering a high cognitive load (Graesser & McNamara, 2011; Kintsch & Walter Kintsch, 1998; Snow, 2002; Zwaan & Radvansky, 1998). The importance of building accurate mental models during learning tasks has been well established for decades in domains like science (Chi et al., 1981; de Jong & Ferguson-Hessler, 1991; DiSessa, 1993; Nathan et al., 1992) as well as in CS education (Margulieux et al., 2012; Pennington, 1987; Ramalingam et al., 2004; Soloway & Ehrlich, 1984).

In the previous Chapter 3, we investigated and found that simply promoting free-self explanations during code comprehension induces learning gain. In this study, we compare Free-Self-Explanation with Guided Self-Explanations. The free Self-Explanation strategies only prompt students to self-explain the code during program comprehension, with no further guidance or help. The Guided Self-Explanation has the same overall goal of eliciting self-explanations for a given code but differs in that it draws more attention to key aspects of the code using a series of guiding questions. The self-explanation strategy employed by ITS is Guided Self-Explanations, which is similar to the Socratic method of tutoring. In this comparison study, the goal is to understand if Guided Self-Explanation incorporated by ITS is better than mere Free Self-Explanation; if not, there is no reason to incorporate self-explanation to ITS while simply prompting self-explanation during code comprehension can do the job effectively. To better understand the students’ code comprehension process, we also propose the model and do students’ mental model using student-generated self-explanations for the code.

Our study aims to answer the following research questions:

- **RQ1**: Does eliciting free self-explanations help learners better comprehend code examples? Is there a discrepancy between low vs. high prior-knowledge
students concerning the impact of this strategy of eliciting self-explanations on student performance and learning?

• **RQ2**: Does Guided Self-Explanations for eliciting self-explanations help learners better comprehend code examples? Is there a discrepancy between low vs. high prior-knowledge students concerning the impact of this strategy on student performance and learning?

• **RQ3**: Which of the two strategies is more effective: Free Self-Explanations or Guided Self-Explanations for code comprehension?

• **RQ4**: How do mental models of low prior knowledge students differ from those of the high prior knowledge group?

We hypothesize that both self-explanation elicitation strategies improve comprehension of code examples (resulting in more accurate mental models) and that both will lead to learning gains. This is based on self-explanation theories and prior evidence about the positive effect of self-explanation on learning and problem-solving in science domains such as biology (Chi et al., 1994), physics (Conati & VanLehn, 2000), math (Aleven & Koedinger, 2002), and programming (Bielaczyc et al., 1995; Tamang et al., 2020). It should be noted that there could be many ways to prompt for self-explanations and understanding which types of prompting work best for whom is still an open question for code comprehension and learning. In other domains, several types of prompts have been studied already, such as eliciting free explanation or justification-based self-explanation prompts (Conati & Vanlehn, 2000) and meta-cognitive self-explanation prompts (Chi et al., 1994). In our case, we study two different strategies for eliciting self-explanations: free Self-Explanations and guided Self-Explanations.

The randomized controlled trial experiment results indicate that the learning gains of the free Self-Explanation strategy were 30% (M = 0.30, SD = 0.47) whereas
the guided Self-Explanation based on the Socratic method was 59% \((M = 0.59, SD = 0.39)\). The Socratic method outperformed the free Self-Explanation strategy by 29% - the difference was statistically significant \((p < 0.05 \text{ level})\). There was no significant difference in mean learning gains for low vs. high prior-knowledge students for both the free Self-Explanations and the Guided Self-Explanations. Our key findings are: 1) Both eliciting free Self-Explanations and the Guided Self-Explanations is an effective intervention for learning programming, and 2) both strategies help equally low and high-prior-knowledge students.

While investigating the effectiveness of eliciting self-explanations in the area of CS Education has been explored before under certain experimental conditions and types of prompting and instructional activities such as problem-solving, the main contributions and novel aspects of our work are the comparisons of free Self-Explanation and the Guided Self-Explanations for code comprehension and learning. There has been no prior work comparing these two forms of self-explanations in the CS Education literature to the best of our knowledge.

**Related Works**

Self-explanation theories indicate that students who engage in self-explanations, i.e., explaining the target material to themselves, are better learners, i.e., learning more deeply and showing the highest learning gains. Self-explanation’s effectiveness for learning is attributed to its constructive nature, e.g., it activates several cognitive processes such as generating inferences to fill in missing information and integrating new information with prior knowledge, and its meaningfulness for the learner, i.e. self-explanations are self-directed and self-generated making the learning and target knowledge more personally meaningful, in contrast to explaining the target content to others (Roy & Chi, 2005). The positive effect of self-explanation on learning has been demonstrated in different science domains such as biology (Chi et al., 1994), physics (Conati &
Vanlehn, 2000), math (Aleven & Koedinger, 2002), and programming (Bielaczyc et al., 1995; Tamang et al., 2020).

A series of studies (Bielaczyc et al., 1995; Pirolli & Recker, 1994; Recker & Pirolli, 1990) found that self-explanations help learning Lisp programming concepts. They found that skill improvement strongly correlated with the amount of self-explanation generated. Two other studies, Rezel (2003) and Alhassan (2017), found that students who used a self-explanation strategy while studying worked-out examples were more successful at a program construction task (Visual Basic) compared to those who did not apply the strategy. Effectiveness of the self-explanation in programming was also studied for SQL (Yuasa, 1994), JavaScript (Kwon & Jonassen, 2011), HTML (Kwon et al., 2011) and assembly language (Hung, 2012). The study by Bielaczyc et al. (1995) showed that university students who underwent explicit training on strategies of self-explanation and self-regulation outperformed students in a control group (no explicit training) on problem-solving performance.

In our case, we explore the role of two strategies for eliciting self-explanations for code comprehension and learning. In particular, our work is part of our larger efforts to explore the role of an advanced online education technology to train learners on using self-explanations to improve code comprehension processes and outcomes (accurate mental models, learning). More specifically, we explore two particular strategies for eliciting self-explanations: free or open-ended self-explanations versus guided explanations using a Socratic method (Alshaikh, Tamang, & Rus, 2020a, 2020b) that elicits self-explanations by using a series of guiding questions that draw attention to crucial aspects of a given code.

Indeed, there are different ways to elicit self-explanations which result in different types of self-explanations such as spontaneous self-explanations (no prompting), free or open-ended self-explanations (simple prompting to self-explain),
and scaffolded self-explanations (in this case students are encouraged to self-explain as much as possible by themselves and offered support in the form of hints when floundering). Other forms of self-explanations have been tried, such as “complete given self-explanations (fill-in the blank self-explanations)” (Kwon et al., 2011) and “select a self-explanation/menu-based self-explanations” (Aleven & Koedinger, 2002; Fabic et al., 2019) which one may argue are not true self-explanations as the learner does not generate the explanation, i.e., the ‘self’ part of the ‘self-explanation’ is missing. Furthermore, self-explanation prompts can emphasize various aspects of self-explanations resulting in justification-based self-explanation prompts (Conati & Vanlehn, 2000) or meta-cognitive self-explanation prompts (Chi et al., 1994). Again, we explore here two types of self-explanations for code comprehension and programming concept learning: eliciting free or open-ended self-explanations and guided self-explanations using a Socratic method.

The Guided Self-Explanations using the Socratic method has been adopted for instruction in many domains; however, it has been rarely used for computer programming instruction. A study Rosé, Moore, VanLehn, and Allbritton (2001) compared the effectiveness of Guided Self-Explanations versus didactic tutoring in a simulated problem-solving environment for teaching basic electricity and electronics concepts. They found that participants in the Guided Self-Explanations condition learned more than students in the didactic condition. In the area of computer programming, there is one prior effort that studied the use of Guided Self-Explanations with Socratic dialogue for teaching recursion and reported positive results (Chang, Lin, Sung, & Chen, 2000). Our study further investigates the effectiveness of the Guided Self-Explanations using the Socratic method to develop an adaptive instructional system for code comprehension and learning. We targeted six concepts: operator precedence, nested if – else, for loops, while loops, arrays, creating objects and using their methods. Targeting a broader set of
computer programming concepts helps test the generality of the Socratic methods across many concepts.

Importantly, to the best of our knowledge, there is no prior comparative study of free self-explanations versus the Guided Self-Explanations using the Socratic method, definitely not in the context of source code comprehension and learning programming concepts. The closest such study is O’Reilly, Symons, and MacLatchy-Gaudet (1998) which examined the effectiveness of two learning strategies, self-explanation, and elaborative interrogation, for the retention of scientific facts. They indicated that self-explanation participants significantly outperformed elaborate interrogation and repetition control participants on cued recall and recognition measures. The elaborate interrogation strategy is related to the Socratic method because both are based on using a line of questioning of the students to elicit elaborative answers.

Comparative Study

We conducted a randomized control trial experiment in which participants were assigned to three approximately equal experimental groups: a free Self-Explanation group, the Guided Self-Explanation group, and a Prediction Only group. Then, participants were shown code examples and asked to either self-explain what the code does (free Self-Explanation) or answer a series of guiding questions related to critical aspects of the code examples (Guided Self-Explanation), or only predict the output of the code examples (Prediction Only). All participants took a pretest before being assigned to an experimental condition and a posttest afterward. The pre-/post-test scores were used to calculate learning gains as a measure of the effectiveness of the interventions. We used a web-based software system to run the experiment.

We conducted a one-way between-group analysis of variance (ANOVA) to compare the mean learning gains of the experimental groups. Furthermore, we
categorized participants into low and high-prior knowledge groups based on their mean pretest scores for the free Self-Explanation and the Guided Self-Explanation groups separately. Then, we conducted an independent sample t-test to compare learning gains between the low and high prior knowledge groups. More details about group design, participants, materials used, experiment protocol, and measures are given in the following sections.

**Group Design**

**Free Self-Explanation**: Participants in this condition were given one JAVA code example at a time. They were then asked to first self-explain their understanding of the code in as much detail as possible. An input box where they could type their self-explanation was available on the interface below the given code example. The code examples were shown to the learners as a whole as opposed to one line at a time. Each code example focused primarily on a major programming concept such as loops. After they submitted their self-explanation, learners were asked to predict the output of the code. Once they submit their prediction, they are shown the next code example, and the process repeats.

**Guided Self-Explanation**: Participants in this group experienced a Guided Self-Explanation using the Socratic method for reading and comprehending the code examples. They were shown one code example at a time and asked a series of questions that guided them in their process of examining the code and paying attention to key aspects such as the target topic or concept. These guiding questions for all six code examples used were developed after extensive discussions by our team of researchers. The ideal interaction with Socratic tutor that uses Guided Self-Explanation looks like in Fig. 4.1

**Prediction Only**: This is the control group. The participants in this group were shown code examples and asked to predict the program’s output.
The code above your screen contains a JAVA program. Please take your time to read each line to understand the code and answer the following questions.

**Tutor** How many times the loop will be executed?

**Me** 6 times

**Tutor** Let me help you out. What is the initial value of the counter i?

**Me** 0

**Tutor** Good! When does the loop stop?

**Me** When i is 5

**Tutor** Great job! How does the value of the counter i changes each time the loop executes?

**Me** The value of i increases by one each time the loop executed

**Tutor** Good! So, what are the values of i during the execution?

**Me** 0, 1, 2, 3, 4

**Tutor** Great job! So, how many times the loop will be executed?

**Me** 5 times

**Tutor** Great job! Let’s move to the next question.

**Tutor** What is the output of line 7?

Type your response here.

Figure 4.1 Interface for Guided Self-Explanation
All participants in all conditions were shown the correct prediction for each code example.

Participants

All 105 students of the third semester in an undergraduate program in computer science, 35 in each group, from an urban university in South East Asia took part in this experiment. All participants had undertaken courses on programming in prior semesters, including structured programming C, and they just had taken an introductory Java course for two weeks at the time of this experiment. The participants were recruited with the help of the instructor and were paid for their participation. The instructor briefed students about the experiment’s goal, including the fact that it is an excellent opportunity to solidify their programming skills. They were also told that participation in the experiment is purely voluntary.

Materials

Participants in each experimental group were shown the same set of 6 source code examples. We call these six code examples the main task. Before this main task, participants were given a pretest consisting of 6 code examples matching in terms of content (i.e., target concepts and code examples) in the main task. Furthermore, participants took a posttest consisting of 6 code examples matching in content in the main task. For the pretest and posttest, learners were supposed just to provide the predicted output. As already noted, the pretest and posttest were not identical, but they were equivalent in terms of concepts tested and difficulty level. The main programming concepts covered by the experiment were: operator precedence, nested if – else, for loops, while loops, arrays, creating objects and using their methods. Each of these concepts was present in a code example used in the pretest, posttest, and the main task.
Protocol

The experiment was conducted in a computer lab in the presence of the instructor but not the researchers of this study. Participants were told to ask questions, if required, only about the experimental procedure and system usage issues. The participants were first debriefed about the purpose and nature of the experiment and given a consent form. Upon their agreement, they were given a background survey followed by a pretest. Then, they worked on the main task. Finally, they took the posttest. Participants could see all pretest and posttest questions at once on a single screen. They were shown the main tasks one at a time, and they could proceed to the next task only after submitting the answer for the current task. All participants’ responses and interactions were automatically logged for posthoc analysis.

Measures

To score the student performance on the main task, we used the predicted output for each code example. Each question was scored 1 if the final predicted output was correct and 0 for incorrect answers. This means the maximum score was 6 for the pretest, the main task, and the posttest, respectively. For each participant, we also calculated the learning gains score using the following procedure (Marx & Cummings, 2007).

- If posttest score > pretest score, learning gain = (posttest-pretest)/(6-pretest).
- If posttest score < pretest score, learning gain = (posttest-pretest)/pretest.
- If posttest score = pretest score = 6, discard the data.
- If posttest score= pretest score ≠ 6, learning gain = 0.

When posttest = pretest = 6, the learning gain = 0; such scores are discarded, i.e., participant data with perfect scores of 6 in both pretest and posttest.
is discarded. Our study ended with valid learning gains for 88 participants (28, 29, and 31 students in the Prediction only, Self-Explanation, and Socratic conditions, respectively). A total of 17 participants’ data points were discarded because they either achieved perfect scores in both pretest and posttest (i.e., pretest = 6 and post-test=6) or did not complete all parts of the experiment.

**Results**

The three randomly sampled experimental groups are equivalent in terms of pretest score (i.e., prior knowledge): they have the same mean (M) pretest score of 3.4 with standard deviation (SD) of 1.95, 2.04, and 1.76 for prediction only, self-explanation and Guided Self-Explanation, respectively. Likewise, in the same order, the mean posttest score is 3.57 (SD=1.53), 4.17 (SD=1.73), and 4.77 (SD=1.33), whereas the mean task score is 3.29 (SD=2.26), 4.07 (SD=1.96) and 4.84 (SD=1.19). It is to be noted that any parametric techniques mentioned in this section 4 and section 4 met all the underlying assumptions, such as normal distribution, random sampling, independence of observations, and homogeneity.

The result of ANOVA shows that there is a statistically significant difference ($p < 0.05$ level) in learning gains for the three groups (Self-explanation, Guided Self-Explanation, and Prediction only): $F(2,85) = 13.5$, $p=0.001$. The difference in mean learning gain score is large, as suggested by Sawilowsky (2009). The effect size, calculated using eta squared, was 0.24. Post-hoc comparison using Tukey’s test indicated that mean score of the Prediction only ($M = 0.047$, $SD = 0.31$) was significantly different from the score of the Self-Explanation ($M = 0.30$, $SD = 0.47$). We also found that the Self-Explanation ($M = 0.30$, $SD = 0.47$) and the Guided Self-Explanation group ($M = 0.59$, $SD = 0.39$) were significantly different. There was also a significant difference between the Prediction only ($M = 0.047$, $SD = 0.31$) and the Guided Self-Explanation group ($M = 0.59$, $SD = 0.39$). These results indicate that both self-explanation (with an average learning gain of 30%) and the
Guided Self-Explanation (with an average learning gain of 59%) are effective at helping students learn computer programming concepts in JAVA. The Guided Self-Explanation outperforms the free Self-Explanation by 29%.

The results of the independent-sample t-test to compare the mean learning gain score for low and high prior-knowledge groups for Self-Explanation and Guided Self-Explanation are shown in Table 4.1 and Table 4.2, respectively. Note that any participants with pretest score ≤ mean pretest score for the groups falls into low prior-knowledge, and the rest are in the high prior-knowledge group. The mean pretest score is 3.38 and 3.39 for Self-Explanation and Guided Self-Explanation groups, respectively. Table 4.1 shows that there is no significant difference in mean learning gain score for the participants in the Self-Explanations group, and the magnitude of the difference in mean learning gains (mean difference = .02, 95% CI: -.34 to .39) is very small (eta squared = .006). Similarly, from Table 4.2, we can see that there is also no significant difference in mean learning gains for the participants in the Guided Self-Explanation group and that the magnitude of the difference in mean learning gains (mean difference = -.24, 95% CI: -.53 to .03) is large (eta squared = .10). Thus, we found no significant evidence of a discrepancy between low versus high prior knowledge participants for the two strategies (free Self-Explanation and Guided Self-Explanation), respectively. The two methods work equally well for students of all prior knowledge levels.

**Mental Model Analysis**

To understand the mental model that students constructed while comprehending the code examples, we performed an in-depth analysis based on a

<table>
<thead>
<tr>
<th>Group</th>
<th>N</th>
<th>Mean</th>
<th>SD</th>
<th>t-val</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Prior Knowledge</td>
<td>13</td>
<td>0.31</td>
<td>0.44</td>
<td>0.67</td>
<td>0.88</td>
</tr>
<tr>
<td>High Prior Knowledge</td>
<td>16</td>
<td>0.28</td>
<td>0.42</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1 Independent Sample t-test result for learning gain between Low and High prior knowledge group in Self-Explanations
<table>
<thead>
<tr>
<th>Group</th>
<th>N</th>
<th>Mean</th>
<th>SD</th>
<th>t-val</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Prior Knowledge</td>
<td>14</td>
<td>0.45</td>
<td>0.31</td>
<td>0.67</td>
<td>0.87</td>
</tr>
<tr>
<td>High Prior Knowledge</td>
<td>17</td>
<td>0.70</td>
<td>0.43</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2 Independent Sample t-test result for learning gain between High and Low prior knowledge group in Socratic Method

A qualitative inspection of the self-explanations, which helps to reveal the learner’s comprehension at a more refined grain level. This analysis was conducted only for the free Self-Explanations students as the Guided Self-Explanations led to shorter, less revealing responses due to the guiding questions, which asked for specific responses to the key aspect of the code. We randomly selected five low and five high prior-knowledge students from the Self-Explanation group. Then, one of the authors of this experiment analyzed students’ self-explanations along with a number of factors that accounted for the quality of the self-explanations and mental models. These factors were based on self-explanation and code comprehension theories, e.g., the distinction between the program model, the domain model, and the situation model (Pennington, 1987; Schulte et al., 2010). The following factors were used for this qualitative analysis.

**Prior reference:** Do they make any references to prior knowledge? Quality self-explanations entail the integration of new information and prior knowledge.

**Inferences:** Do they make any inferences such as bridging inferences? For instance, when students explain one part of a line of code, do they refer to prior lines or code or prior elements of the code (bridging inferences)?

**Monitor:** Do they monitor and reflect on their understanding? Self-monitoring of one’s comprehension is also a key aspect of quality self-explanations.

**Control flow:** Can students correctly identify the control flow of the program? Do they identify the order in function calls, instructions and statement execution when the program runs?
Data flow: Can students correctly identify the data flow and the state of all data at each moment of the code execution, i.e., how data structures are created, updated, or transformed?

Program model: Do students show an understanding of the major structural components of the code, i.e., the program model?

Domain model: Do students show an understanding of the domain model?

Integrated model: Do students show an understanding of the integrated model of the code by referring to links between the domain model and program model?

Mental model: This factor combines all the above into a holistic score: prior reference + inferences + monitor + control flow + data flow + program model + domain model + integrated model

The first three factors above are measured in terms of the average number of times (across all six main tasks) that learners make references to prior knowledge, the number of inferences, and how many times they self-monitor their understanding, respectively. The rest of the factors (except the mental model) were measured using the 0-4 Likert scale (0 - Very Poor, 1 - Below Average, 2 - Average, 3 - Above Average, 4 - Excellent) for each task and the final score used is the average score across all six main tasks.

Descriptive statistics and the results of an independent sample t-test were obtained to compare the mean score for each of these factors between low and high prior-knowledge groups to understand any qualitative differences in the mental models those groups constructed.

Results

The results show that no student made references to prior knowledge or self-monitored their understanding. Only five students out of the 10 made 19 inferences for all six tasks altogether. Out of this, 18 inferences were made by four
students in the high prior knowledge group, and only one other student made one inference in the low prior knowledge group. *This suggests that high prior-knowledge students make more inferences whereas low prior knowledge students do not.* Thus, training students to make more inferences could be possibly applied to make them more competitive.

Albeit using a small sample, we found a significant difference in mental model score for low prior knowledge group (M=4.99, SD = 1.05) and high prior knowledge group (M=8.67, SD = 3.57; t(8) =-2.233, p = 0.05, two-tailed). The magnitude of the difference in the mean (mean difference =-3.88, 95 % CI: -7.7 to -0.03) was very large (eta squared = 0.38). Similarly, there is significant difference in control flow score between the low prior knowledge group (M=1.30, SD = 0.38) and the high prior knowledge group (M=2.37, SD = 0.99; t(8) =-2.237, p = 0.05, two-tailed). The magnitude of the difference in the mean score (mean difference =-1.07, 95 % CI: -2.16 to -0.03) was very large (eta squared = 0.38). There was no significant difference in mean score for inferences, data flow, program model, domain model, or integrated model. *These results indicate high prior knowledge students are better than low prior knowledge group in using control flow to build a better mental model.* Hence, the low prior knowledge group could be trained more on control flow aspects of code reading and understanding to help them build a better mental model, which helps them be more accurate.

**Conclusion**

We presented in this paper the results of a randomized control trial experiment that compared the effectiveness of two instructional strategies that scaffold learners’ code comprehension processes: eliciting free Self-Explanation and a Guided Self-Explanation using the Socratic method. The results showed pre-/post-test learning gains of 30% (M = 0.30, SD = 0.47) for the free Self-Explanation condition and learning gains of 59% (M = 0.59, SD = 0.39) for the
Guided Self-Explanation. There was no significant difference in mean learning gains for both self-explanations and the Guided Self-Explanation for low vs. high prior-knowledge students, i.e., no evidence of a discrepancy between these interventions to students based on their prior knowledge.

We also analyzed students’ comprehension using an in-depth analysis of their self-explanations and comprehension by assessing the quality of the self-explanations and the resulting mental models. The findings of this analysis suggest that students who make inferences and emphasize control flow are better comprehenders and learn more.

The Guided Self-Explanation using the Socratic method uses a sequence of guided questions emphasizing critical aspects of the target code, which could be its better performance than the free Self-Explanation, which does not provide any specific hints. While the latter seems to be more revealing in terms of learners’ comprehension processes and the resulting mental models, offering more support in the form of hints or guiding questions as in the Guided Self-Explanation proves to be more beneficial to learning.

One of the limitations of our study is the coverage of CS topics in CS1 and CS2 courses. While this experiment focusing on six programming concepts was a good start to investigating and comparing the effectiveness of free Self-Explanations and of the Socratic method, running a semester-long experiment covering all intro-to-programming topics would be more conclusive. Furthermore, our in-depth analysis of the mental model students constructed needs to be extended to all students in the free Self-Explanation group as opposed to just a 5+5 sub-sample. In future work, there is a need to run a semester-long study covering all topics in an intro-to-programming course and extend our analysis of mental models to all students in the free Self-Explanations condition.
Chapter 5

Automatic Question Generation Using Each Sentence in Code

Introduction

The use of Automatic Question Generation (AQG) (Das et al., 2021; Umardand & Gaikwad, 2017) for educational purposes (Kurdi et al., 2020; Le et al., 2014) has attracted a lot of interest from researchers in different disciplines. A recent review study by Kurdi et al. (2020) found only 3 AQG works targeting the domain of computer programming, which is the focus of our work. However, given the growing popularity of intro-to-programming courses, particularly for interdisciplinary domains such as data science, the need for adaptive instruction to accommodate learners of various backgrounds is only expected to grow exponentially. This work is part of our larger effort to develop Intelligent Tutoring Systems (ITS) to help novices master code comprehension. Such ITS relies on hints in the form of questions to provide the necessary support for learners. When done by human experts, authoring such questions is an expensive and tedious process, which is currently the norm. An automated QG process would address this challenge of authoring questions at scale for ITS. Thus, in this research, we apply state-of-art AQG to generate questions (short and gap-fill questions) using expert-provided code-block explanations intended to scaffold student self-explanations of the code during code comprehension tasks.

As already noted, there is some prior work in the area of AQG for programming (Alshaikh, Tamang, & Rus, 2021; Brusilovsky & Sosnovsky, 2005; Dancik & Kumar, 2003; Hsiao, Brusilovsky, & Sosnovsky, 2008; Thomas, Stopera, Frank-Bolton, & Simha, 2019; Zavala & Mendoza, 2018). Most of these works (Brusilovsky & Sosnovsky, 2005; Dancik & Kumar, 2003; Hsiao et al., 2008; Thomas et al., 2019; Zavala & Mendoza, 2018) create clones of programming exercises or
code-snippets for targeted concepts or topics in order to provide students an opportunity to practice more and master the topic(s). However, these exercises are more like opportunities to practice as opposed to questions meant to scaffold students’ learning while working on those exercises. AQG work meant to generate scaffolding questions was done by Thomas et al. (2019) for program-tracing (short, multiple-choice mental execution questions). Similarly, Alshaikh et al. (2021) have worked on helping students comprehend particular code examples by asking sequences of automatically generated short questions from static analyses of code. Nevertheless, these efforts rely on a template-based QG approach, which requires significant expert time to design the templates and can only generate a few types of questions. Also, the template-based approach cannot generate questions that inquire about a deeper understanding of the code.

The two systems, Machine Noun QG and Machine Verb QG, are designed to take as inputs textual descriptions of the code, e.g., explanations generated by experts, and output questions of two types: short questions and gap-fill questions. The short questions are intended to help students focus on certain parts and aspects of the code to scaffold their comprehension processes and learning. In a tutoring system, for instance, that follows constructivist theory of learning (Ben-Ari, 2001) in which students are offered the right dosage of help when needed, not more, not less, the hints in the form of short questions may be followed by gap-fill questions or bottom out hints if the student is still struggling. That is, the gap-fill questions provide more information for the student.

Unlike past work, we do not use a template approach for question generation. Instead, we adopt a state-of-art pre-trained sequence-to-sequence model ProphetNet (Qi et al., 2020) together with fine-tuning for QG using the SQUAD dataset (Rajpurkar, Zhang, Lopyrev, & Liang, 2016). Using textual explanation of the code as input together with the ProphetNet model leads to a more computer...
language-independent approach to QG. Also, it has the advantage of producing a deeper and wider variety of questions (as opposed to just those captured by the expert-generated templates).

In sum, this paper answers the following research questions:

1. Is it possible to automatically generate short questions that are linguistically well-formed, pedagogically sound, and indistinguishable from human-generated questions?

2. Is it possible to automatically produce gap-fill questions useful in ITS?

3. How do machine-generated questions compare to human-generated questions?


In the rest of the paper, we first describe the design of our system in detail in section 5. Next, we provide detail on how we evaluate our systems in section 5. Then, we present the result of the evaluation in section 5. Finally, we also present the conclusion of this work in section 6

**System Design**

We designed our two systems to generate short questions and gap-fill questions for every sentence in explanations of programming examples. The self-explanations we use in our work are block-level explanations, i.e., the code is divided into logical blocks, and logical-level explanations are generated. Implementation details about how the logical steps are implemented in the code are provided as well. Since the explanations are expert-generated, we assume that each sentence describes an essential aspect of the code. Therefore, ideally, students must think, and explain if prompted, all these aspects present in the expert generated explanations if they correctly and entirely understood the code. Our main task is
thus to generate questions that are linguistically well-formed, pedagogically sound, and indistinguishable from human-generated questions. Other concerns such as how to dynamically trigger only needed short questions and when to follow up with gap-fill questions based on the student response are beyond the scope of this study. Furthermore, in other settings, e.g., where the explanations are not generated by experts, i.e., they may be noisy, using a sentence selection strategy to select the only sentences that are informative, important, or contain novel information (Chen, Yang, & Gasevic, 2019) may be needed. However, that is beyond the scope of this work.

We describe, below, the detailed implementations for each of the two systems: the Machine Noun QG and the Machine Verb QG.

**Machine Noun QG**

As the name indicates, Machine Noun QG considers noun chunk in a sentence as the main target to generate questions about. Targeting the noun chunks makes sense as they most likely refer to key concepts or steps in the solution to the problem the code is solving. There are some other linguistic, e.g., noun phrases are quite frequent in language, and technical reasons such as the fact that they are easier to identify automatically by syntactic parsers (full or shallow parsers) that noun phrases is the most common answer category and makes 31.8% of all answers in SQUAD dataset (Rajpurkar et al., 2016).

First, Machine Noun QG segments the explanations into sentences and then selects a noun chunk for every sentence - details follow. We divide the self-explanation into constituent sentences using a library called pySBD, a pipeline extension in spaCy 2 (Honnibal & Montani, 2017), which applies the Golden Rule Set for sentence boundary detection as described in Sadvilkar and Neumann (2020). For each sentence, we extract noun chunks, also using spaCy 2. When a sentence has multiple noun chunks, the first step is to discard any noun chunk with more
than 4 words; Chau, Labutov, Thaker, He, and Brusilovsky (2021) define “single
words or short phrases of two to four words” as domain concepts (i.e., ideally what
we would like to target with our questions). Then, we select the longest noun chunk
from the remaining noun chunks under the assumption that longer inputs are
beneficial for the question generator. If two noun chunks have the same length, we
select the noun chunk that has appeared first in the sentence, assuming that an
important keyphrase comes first. Next, the Machine Noun QG generates short
questions and gap-fill questions. To generate short questions, we used ProphetNet
(Qi et al., 2020), a pre-trained sequence-to-sequence model, fine-tuned for question
generation tasks using the SQUAD (Rajpurkar et al., 2016) dataset. We pass a pair
of <sentence, selected noun chunk for the sentence> to the model, and the model
outputs the short question. The gap-fill question is created by masking the
sentence’s noun chunk. In this way, we can obtain short questions and gap-fill
questions for all sentences in the expert-generate explanations of the code.

**Machine Verb QG**

Machine Verb QG targets verb phrases in the input sentences as the main
target for generating questions.

Machine Verb QG segments the self-explanations into constituents sentences
using the same process as in Machine Noun QG. Then, we obtain verb phrases in a
sentence by extracting a sequence of tokens that matches the pattern = ['POS':
'VERB', 'OP': '?', 'POS': 'ADV', 'OP': '*', 'POS': 'AUX', 'OP': '*', 'POS': 'VERB',
'OP': '+']. To match the pattern, we use Matcher in the spacy 2 library. Next, we
select the single verb phrase similar to how noun chunk is selected in Machine Noun.

Finally, Machine Verb QG generates the short question and gap-fill questions
similarly to the process the the Machine Noun QG does.
Evaluation

To evaluate our systems, we generated questions using both the Machine Noun QG and Machine verb QG systems and a set of code examples annotated with explanations by experts that followed annotation guidelines rooted in program comprehension theories and theories of human learning (see section 5). Sample short questions and gap-fill questions generated by the machines are shown in Table 5.1 and Table 5.2, respectively. A set of evaluation criteria (see section 5) were used to evaluate the automatically generated questions as described in section 5.

Dataset

```java
/* SAMPLE CODE EXAMPLE in our Dataset*/

/* TOPIC: Arrays, TASK: Calculate Average of numbers */
public class AverageOfNumbers {
    public static void main(String[] args) {

        /* Code-Block 1, Expert-Explanation, short and gap-fill questions*/
        double[] numArray = {8,6,11,7};
        double sum = 0.0; double average;

        /* Code-Block 2
        The sum of numbers is calculated using a for loop that iterates over each number in the numArray array and adds each number to the sum. When the for loop completes execution, the value of the sum is 32.
        */
        for (int i = 0; i < numArray.length; i++) {
            sum += numArray[i];
        }

        /* Code-Block 3, Expert-Explanation, short and gap-fill questions*/
        average = sum / numArray.length;
    }
}
```
Our dataset consists of 10 code examples, one each for the following topics typically found in intro-to-programming courses: logical operators, if-else, while loops, for loops, nested loops, arrays, two-dimensional arrays, methods, and class & objects. Each code example has a well-defined goal, well-separated code blocks, expert-generated explanations for each code block, and short and gap-fill questions to help elicit each sentence in the explanation; see the sample code example above. These code examples were prepared by a group of subject matter experts who refined the explanations through several iterations. Note that expert-explanation, short and gap-fill questions are intentionally not provided for code-block 1 and 3 in the sample code example above to save space.

**Evaluation Criteria**

We evaluated short questions using the following criteria. The first two criteria measure linguistic well-formedness, whereas the other criteria (3,4,5) check the pedagogical soundness. The last criterion determines how likely a human judge would find the question indistinguishable from a human-generated question.

1. **Grammaticality**: Is the question grammatically correct, i.e., free of language errors? Scale: 1 (Very Poor) to 5 (Very Good).

2. **Semantic Correctness**: Is the question semantically correct, i.e., meaning-wise? Scale: 1 (Very Poor) to 5 (Very Good).

3. **Domain Relevancy**: Is the question relevant to the target domain, i.e., does it target a programming concept? Scale: (Yes/No)

4. **Answerability**: Does the question have a clear answer in the input text? Scale: (Yes/No)
5. **Helpfulness**: Is the question likely to help the student think about the target concept and produce an answer close to the expert-provided explanation? Scale: 1 (Not Likely) to 5 (Very Likely).

6. **Recognizability**: How likely is it that a human generated the question? Scale: 1 (Not Likely) to 5 (Very Likely).

Each gap-fill question is labeled to one of the following categories.

1. **Good**: It asks about key concepts from the input sentence and would be reasonably difficult to answer.

2. **OK**: It asks about a) key concepts but might be difficult to answer (the answer is too lengthy, the answer is ambiguous, etc.) or b) likely key concept (weak concept).

3. **Bad**: It asks about 1) an unimportant aspect of the sentence or 2) has an uninteresting answer that can be figured out from the context of the sentence.

4. **Acceptable**: A question that is OK or Good is also automatically labeled as acceptable for evaluation purposes as explained later (the annotator does not have to do this).

**Annotation**

Two independent human experts annotated a total of 450 questions. It consisted of 225 short questions and 225 gap-fill questions; Human, Machine Noun QG, and Machine Verb QG generated 75 questions in each question category. The human experts were two PhD students studying computer science with 5+ years of professional programming experience; who had worked as teaching assistants for intro-to-programming courses. The judges were first trained on the annotation protocol and asked to annotate 25 short questions, after which they got a chance to ask any clarification questions and revise their annotation score if necessary before
proceeding with the rest of the questions. The goal of the first 25 questions was to develop the same understanding of the evaluation criteria and for more consistency. Then, they were presented with the remaining 200 short questions to judge based on the criteria just discussed in section 5. Similarly, they judged the 225 gap-fill questions. In both cases, the questions were randomized before being presented to each annotator. Annotators were also provided the source sentence from which the question was generated. As measured by Cohen’s Kappa, the final inter-annotator agreement was 0.30, 0.39, 0.71, 0.93, 0.37, 0.37, and 0.91 for grammaticality, semantic correctness, domain relevancy answerability, helpfulness, recognizability, and gap-fill questions, respectively.

<table>
<thead>
<tr>
<th>Self-Explanation</th>
<th>Human</th>
<th>Machine Noun QG</th>
<th>Machine Verb QG</th>
</tr>
</thead>
</table>
| The sum of numbers is calculated using a for loop that iterates over each number in the numArray array and adds each number to the sum. When the for loop completes execution, the value of the sum is 32. | 1. How is a sum of numbers calculated?  
2. What is the value of the sum when the for loop completes execution? | 1. How is the sum of numbers calculated?  
2. When does the for loop finish execution? |

Table 5.1 Sample Short Questions generated by Human, Machine Noun QG and Machine Verb QG

**Results**

The performance overview of the two systems on short questions and gap-fill questions is shown in Table 5.3 and Table 5.4, respectively. To compare the performance of Machine Noun QG, Machine Verb QG, and human experts in terms of grammatically, semantic correctness, helpfulness, and recognizability, we applied independent-samples t-tests which indicate whether the difference in mean (M) between any two generators is significant. Similarly, we applied a chi-square test of
The sum of numbers is calculated using a for loop that iterates over each number in the numArray array and adds each number to the sum. When the for loop completes execution, the value of the sum is 32.

<table>
<thead>
<tr>
<th>Self-Explanation</th>
<th>Human</th>
<th>Machine Noun</th>
<th>Machine Verb</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The sum of numbers is calculated using a for loop that iterates over each number in the numArray array and adds each number to the sum. [for loop, Good]</td>
<td>1. The sum of numbers is calculated using a for loop that iterates over each number in the numArray array and adds each number to the sum. [for loop, Good]</td>
<td>1. The sum of numbers is calculated using a for loop that iterates over each number in the numArray array and adds each number to the sum. [for loop, Good]</td>
<td>1. The sum of numbers is calculated using a for loop that iterates over each number in the numArray array and adds each number to the sum. [for loop, Good]</td>
</tr>
<tr>
<td>2. When the for loop completes execution, the value of the sum is 32. [Okay]</td>
<td>2. When the for loop completes execution, the value of the sum is 32. [for loop, Okay]</td>
<td>2. When the for loop completes execution, the value of the sum is 32. [for loop, Okay]</td>
<td>2. When the for loop completes execution, the value of the sum is 32. [for loop, Okay]</td>
</tr>
</tbody>
</table>

Table 5.2 Sample Gap-Fill Questions generated by Human, Machine Noun QG and Machine Verb QG. The first text inside [] is answer and the second text represents the quality of gap-fill questions given by human annotator.

independence (with Yates’ continuity correction) to check whether the difference in the proportion of answerability, domain relevancy, and acceptability of the gap-fill question was significant. Note that all significant test results are significant for a p-value less than or equal to 0.05. We report results in detail only when the difference is significant. Otherwise, for space reasons, we just indicate that the result is not significant.

The results of our analysis are summarized below; see sections 5 and 5.

Results for Short Questions

Is it possible to automatically generate short questions that are linguistically well-formed? (Grammaticality and Semantic Correctness): Both Machine Noun QG and Machine Verb QG generated grammatically very good questions with the mean grammaticality scores of 4.51 and 4.64, respectively. Likewise, these machines also
Table 5.3 Performance of Machine Noun QG, Machine Verb QG, and Human on Short-questions. SD= Standard Deviation.

generated semantically very good questions where the mean semantic scores for the Machine Noun QG are 4.76 and 4.49 for Machine Verb QG.

Comparison: The mean grammaticality score for human expert generated questions was 4.67 whereas the mean for semantic correctness was 4.84.

1. The grammatical quality of questions generated by Machine Noun QG, Machine Verb QG, and human experts are equivalents; there is no significant difference in mean grammaticality scores between any two of them.

2. The semantic correctness of human generated questions (M=4.84, SD=0.57) is not significantly different from the semantic correctness of questions generated by the Machine Noun QG system (M=4.76, SD=0.46). The Machine Verb QG system (M=4.49, SD=0.80) significantly underperformed relative to the human generated questions for the semantic correctness criterion, t(134.25) = -3.07, p=0.003. Finally, questions generated by the Machine Noun QG system are significantly superior when it comes to semantic correctness compare to the Machine Verb QG system, t(118.62)=2.51, p=0.01.

Is it possible to automatically generate short questions that are pedagogically sound? (Domain Relevancy, Answerability, and Helpfulness). The automated
systems generated questions relevant to the domain of program comprehension in an
impressive proportion: 92% by the Machine Noun QG system and 89.3% by the
Machine Verb QG system. Only the Machine Noun QG system produced a vast
majority of answerable questions (93% of the questions were rated
Answerability=Yes), whereas the Machine Verb QG system produced a bit more
than half of the questions answerable (54.7%). Similarly, the average helpfulness
score of Machine Noun QG questions is 4.27 and therefore likely to help students
think of and articulate the expected answer. On the other hand, the Machine Verb
QG’s average helpfulness score is only 3.44.

Comparison: The human-generated questions are 93% domain related, 97.3%
times answerable, and have an average helpfulness score of 4.31 (likely to help).

1. The proportion of the domain-relevant questions generated by humans,
   Machine Noun QG, and Machine Verb QG are similar; there is no significant
difference between any two approaches.

2. There is no significant difference in the proportions of answerable questions
   generated by Machine Noun QG compared to humans or Machine Verb QG.
   However, the proportion of answerable questions generated by Machine Verb
   QG is significantly lower compared to the questions generated by humans,
   \( \chi^2(1, n=150) = 35.12, p = 0.00 \).

3. For helpfulness, there is no significant difference between human generated
   questions (M=4.31, SD=0.77) and the questions generated by the Machine
   Noun QG system (M=4.27, SD=0.96). However, the questions generated by
   the Machine Verb QG system (M=3.44, SD=0.96) are significantly less
   helpful, \( t(141.27) = -6.09, p=0.00 \) compared to human. The Machine Noun
   QG significantly performs better than the Machine Verb QG system for the
   helpfulness criterion, \( t(148) = 5.26, p=0.00 \).
Is it possible to automatically generate short questions that are indistinguishable from human-generated questions? (recognizability): The Machine Noun QG generated questions are indistinguishable from human generated questions; the mean recognizability score for Machine Noun QG is 3.49 (likely generated by humans) whereas the mean recognizability score for questions generated by the Machine Verb QG system is 2.76 (hard time to say who actually generated the questions). That is, raters think the questions have a good chance of being created by humans.

Comparison: The mean recognizability score for human-generated questions is 4.49. The Machine Noun QG significantly underperformed relative to humans 4.49 (SD=0.91); t(130.56)=-5.38, p=0.00. Similarly, the Machine Verb QG also significantly underperformed humans, t(139.16)=-10.13, p=0.00. The Machine Noun QG system significantly outperformed the Machine Verb QG system, t(139.16)=-10.13, p=0.00.

Results for Gap-Fill Questions

Is it possible to automatically produce gap-fill questions that are useful to use in adaptive instructional systems?: Table 5.4 shows the proportions of bad, okay, good, and acceptable (okay+good) gap-fill-questions generated by the automated systems and humans. The automated approaches generated a majority of acceptable gap-fill-questions, i.e., 84% of the gap-fill-questions generated by Machine Noun QG are acceptable (73.3% okay + 10.7% good) and 80% of the gap-fill-questions produced by Machine Verb QG are acceptable (38.7% okay + 41.3% good).

<table>
<thead>
<tr>
<th></th>
<th>Bad %</th>
<th>Okay %</th>
<th>Good %</th>
<th>Acceptable%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Noun QG</td>
<td>16</td>
<td>73.3</td>
<td>10.7</td>
<td>84</td>
</tr>
<tr>
<td>Machine Verb QG</td>
<td>20</td>
<td>38.7</td>
<td>41.3</td>
<td>80</td>
</tr>
<tr>
<td>Human</td>
<td>2.7</td>
<td>53.3</td>
<td>44</td>
<td>97.3</td>
</tr>
</tbody>
</table>

Table 5.4 Performance of Machine Noun QG, Machine Verb QG, and Human on Gap-Fill Questions.
Comparison: Humans generated 97.3% acceptable (53.3% okay, 44% good) gap-fill-questions.

1. Both Machine Noun QG and Machine Verb QG significantly underperformed compared to humans in generating gap-fill questions that are of acceptable quality. The proportions of the acceptable gap-fill questions generated by Machine Noun QG is significantly lower than the proportion of acceptable gap-fill questions generated by humans, $\chi^2(1, n=150)=6.38$, $p=0.012$. Similarly, the proportions of acceptable gap-fill questions generated by Machine Verb QG is also significantly less compared to humans, $\chi^2(1, n=150)=9.55$, $p=0.002$.

2. Comparing the automated systems using the proportion of acceptable gap-fill questions, they performed more or less similar. There is no significant difference in the proportions of acceptable gap-fill questions generated by Machine Noun QG compared to Machine Verb QG, $\chi^2(1, n=150)=0.19$, $p=0.67$. However, if we look closer, using a finer-grained quality criterion (i.e., proportions of okay and good gap-fill questions too), we found that Machine Verb QG performed better compared to Machine Noun QG. This is because Machine Verb QG yielded more or less equally okay and good quality gap-fill questions, whereas the Machine Noun QG produced a majority of okay and very few good gap-fill questions.

Conclusion

The need to automate and scale the generation of questions to scaffold students' comprehension and learning is critical as the current practice of manual authoring is too expensive and tedious. To address this need, we developed and evaluated in this paper two automated systems, Machine Noun QG and Machine
Verb QG, which automatically generate short-questions and gap-fill-questions for scaffolding students during program comprehension.

Our experiments showed that machines can generate short questions which are linguistically well-formed, pedagogically sound, and likely indistinguishable from human-generated questions. We also found that most gap-fill questions generated by machines (Machine Noun QG: 84%, Machine Verb QG: 80%) are of acceptable quality to be used by adaptive instructional systems. Compared to human experts, Machine Noun QG performed comparable for short questions but underperformed for gap-fill questions in almost all criteria. However, Machine Verb QG performed significantly lower for short (except grammaticality) and gap-fill questions compared to humans. Between the automated systems, Machine Noun QG performed better; it performed better for semantic correctness, helpfulness, and recognizability and similarly in all other criteria.

Since current auto-evaluation methods cannot be relied on Novikova, Dušek, Curry, and Rieser (2017), we opted for human evaluation of questions which in a way limited our dataset to 10 code examples. Furthermore, we did not study the effectiveness of auto-generated questions in terms of student experience and learning gains when used in adaptive instructional technologies, which is necessary and falls under our plan. Instead, we used a proxy: the helpfulness criteria. While our approach is expected to minimize the authoring process significantly, given that there is no guarantee that each question is of high quality along all criteria, in actual learning system development, a human expert must be involved in the loop to check the quality of the questions before being used in an actual learning system. However, this should reduce the time significantly compared to the current approach when human experts have to also generate the questions not only check them. In our current approach, humans also have to generate the comments for the code from which our automated systems generate the questions. One future research direction...
for us is to automate the generation of code explanations using code examples and the surrounding text in programming textbooks. This should automate the authoring process even more leading to more cost-effective and scalable approaches for learning technology development.
Chapter 6

Automatic Question Generation Using Target Concepts in Code Explanation

Introduction

This dissertation aims to develop Self-Explanation-based Intelligent Tutoring Systems (ITS) to help students master code comprehension. Questions are critical components for building such ITS, for they use questions as hints to scaffold students to articulate code explanation (i.e., self-explain what the code does). The current norm is that such questions are authored manually by experts, thus, making them time-consuming and costly (for expert hours). This work is one of our efforts to automatically generate such questions using target concepts in code explanation and thus reduce the cost of using expert hours. Figure 6.1 illustrates what we are trying to achieve: for the given code with explanations, we want to target critical concepts and then generate a question to inquire about it.

Various research has been done in Automatic Question Generation (AQG) for code comprehension (Alshaikh et al., 2021; Brusilovsky & Sosnovsky, 2005; Dancik & Kumar, 2003; Hsiao et al., 2008; Thomas et al., 2019; Zavala & Mendoza, 2018). Most of these work requires an expert to provide the template that combines the information in the code structure to generate the questions. Thus, they are costly, and the resultant questions are not about program comprehension, i.e., what does the code do (e.g., Why is variable result defined?); they are about the code execution (e.g., What is the value of variable result after four loop execution?) or syntax of code (what is the type of variable result?). In our recent effort 5, we train the model on SQuAD (Rajpurkar et al., 2016) to generate questions for code comprehension for each sentence in the code explanation. But, the model generates too many questions for lengthy code explanations and does not handle well the programming jargon in code explanations. The SQuAD, a general reading
In this work, we first develop a large dataset for code comprehension called CodeQG, specific to the programming domain, containing 83141 instances. Then, we train the model in the dataset using Transformer Encoder Decoder (Vaswani et al., 2017) to generate the question based on target concepts in code explanations. Hence, the model generates questions limited to the concept targeted in code explanations instead of each sentence in our previous work. Also, using code explanations in our work instead of the code widely used in template-based approaches, the resultant questions are more about code comprehension.

This study aims to answer the following question:

1. How well does the model in our approach perform in generating questions for code comprehension?

2. Does the model performance increases if the programming domain-specific
dataset is used instead of the general reading comprehension dataset during training?

3. How does the model’s performance differ if the model input sequence (target concept + code explanation) is altered by adding a code (target concept + code explanation + code) or replacing the code explanations with the code (target concept + code)?

In the rest of the paper, we first discuss the related work in section 6 followed by section 6, where we describe two datasets used, CodeQG and SQuAD, including detail on how we formed them. Then, we detail our approach in section 6, providing details of the task, model architecture, and the different model types. Next, in section 5, we present the result of our study. Finally, we end with the conclusion of this work in section 6.

**Related Work**

Although there has been much research on Automatic Question Generation (AQG) for educational purposes (Das et al., 2021; Kurdi et al., 2020; Le et al., 2014; Umardand & Gaikwad, 2017), the recent survey (Kurdi et al., 2020) indicates very few (only three) works in the domain of computer programming education; our studies has found additional few more.

We find that most studies (Alshaikh et al., 2021; Brusilovsky & Sosnovsky, 2005; Dancik & Kumar, 2003; Hsiao et al., 2008; Thomas et al., 2019; Zavala & Mendoza, 2018) use information in the code itself in combination with expert-crafted templates to generate questions about the code. In this approach, code is used either by building Abstract Syntax Tree (AST) to find the different target concepts in the code or by dynamically simulating the code at various stages of execution to predict the code output. Then, the target concepts or the predicted output value is combined with the appropriate template provided by an expert to generate the question for the code. Firstly, expert involvement makes it costly.
Secondly, the generated question focuses only on the syntax or code execution but fails to check if the student truly comprehended the code. Questions for code comprehension should further inquire what the syntax or output of the code means. For example, for the statement “int sum = 0;” in a code, rather than asking what is the type of variable sum, inquiring why the variable sum is defined is more helpful to inquire students during code comprehension.

To address the problem discussed beforehand, our previous work 5 uses code explanation and generates questions for each sentence in the code explanation. For this purpose, we used ProphetNet (Qi et al., 2020) and fine-tuned it for question generation tasks using the SQUAD (Rajpurkar et al., 2016) dataset. By using the code explanation instead of the code, the generated question now inquires about code comprehension. Unfortunately, this approach generates too many questions while the code explanation is lengthy. Also, the SQuAD (Rajpurkar et al., 2016) dataset used to train the model might not be well suited since it does not cover programming jargon used in code explanation. Thus, there is an opportunity to improve the model by using programming-specific datasets and upgrading to more recent ones QG algorithms. However, there is a lack of such programming-specific datasets as well.

In this work, we first form a dataset called CodeQG that is specific to the programming domain. Then, we train the Transformer encoder-decoder model, a deep learning algorithm producing SOTA in many NLP tasks, to generate questions for code comprehension given the code summary (code explanations) and the target concepts. Thus, we generate questions limited to the number of target concepts we use.

**Dataset**

In this section, first, we describe how we constructed CodeQG, a programming-specific dataset. Then, we also describe the SQuAD dataset 1.1, a
general-purpose reading comprehension dataset, and some preprocessing we applied to the dataset for our purpose.

**CodeQG**

CodeQG is a programming domain-specific dataset we specifically constructed for this work of automatically generating questions, based on target concepts, for source code comprehension or understanding. The dataset contains a total of 83,141 instances. Each dataset instance includes the java code, code summary, answer, and question.

The java code is java methods, and the code summary is Javadoc comments describing Java methods’ functionality. The answer is one of the target concepts— a word or phrase in the code summary. The target concepts in our dataset are noun subject, direct object, open clausal complement, temporal, locative, manner, cause, and purpose in the code summary. The questions inquire about target concepts and are generated using the template (see Liu and Wan (2021) for details).

**Statistics**

The general statistics of the codeQG dataset are summarized in Table 6.1, 6.2 and 6.3. Table 6.1 shows the number of instances and unique codes in train, dev, and test split as well in total. Table 6.2 presents the min, max, and average token count for code, code summary, answers, and questions. We use RobertaTokinzer for tokenization. Table 6.3 shows the distribution of different question types in our CodeQG dataset.

<table>
<thead>
<tr>
<th></th>
<th>Instance Count</th>
<th>Unique Code count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train</td>
<td>66515</td>
<td>34843</td>
</tr>
<tr>
<td>Dev</td>
<td>8314</td>
<td>4338</td>
</tr>
<tr>
<td>Test</td>
<td>8313</td>
<td>4242</td>
</tr>
<tr>
<td>Total</td>
<td>83141</td>
<td>43423</td>
</tr>
</tbody>
</table>

Table 6.1 The number of instances and unique code count in a train, dev, test, and total for the CodeQG dataset.

69
<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>5</td>
<td>393</td>
<td>105.42</td>
</tr>
<tr>
<td>Code Summary</td>
<td>4</td>
<td>342</td>
<td>30.43</td>
</tr>
<tr>
<td>Answer</td>
<td>1</td>
<td>25</td>
<td>5.30</td>
</tr>
<tr>
<td>Question</td>
<td>4</td>
<td>50</td>
<td>9.53</td>
</tr>
</tbody>
</table>

Table 6.2 The average, minimum, and maximum count of tokens in the CodeQG dataset.

<table>
<thead>
<tr>
<th>Question Type</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>What</td>
<td>67.78%</td>
</tr>
<tr>
<td>How</td>
<td>10.47%</td>
</tr>
<tr>
<td>Where</td>
<td>5.53%</td>
</tr>
<tr>
<td>Why</td>
<td>8.69%</td>
</tr>
<tr>
<td>When</td>
<td>0.69%</td>
</tr>
<tr>
<td>For What purpose</td>
<td>4.17%</td>
</tr>
<tr>
<td>Other</td>
<td>2.67%</td>
</tr>
</tbody>
</table>

Table 6.3 Distribution of question types in CodeQG dataset.

**Data Source**

We constructed the CodeQG dataset using two data sources: Java code-comment pairs (Hu et al., 2018) and CodeQA (Liu & Wan, 2021).

Java code-comment is a parallel corpus of over seventy thousand Java code-comment pairs from Github. The code is a java method, and the comment is the corresponding Javadoc comment describing the functionality of the Java methods and is taken as a code summary.

CodeQA is a free-form question-answering dataset for source code comprehension, which includes a Java dataset with 119,778 question-answer pairs and a Python dataset with 70,085 question-answer pairs; we only use the java dataset. Each instance in this dataset consists of a code and the question-answer pair for the code.

**Dataset Construction**

To construct the CodeQG dataset, first of all, for every 119,778 codes in the java dataset in CodeQA, we retrieve the corresponding code summary (Javadoc...
comment) from Java code-comment pairs. As a result, the newly formed CodeQG also contains 119,778 instances, where each instance has a code and the code summary, answer, and question corresponding to the code. Note that CodeQA is constructed using Java code-comment pairs. Therefore, searching the code summary for each code in codeQA in the Java code-comment dataset is possible. We then remove any duplicates and apply some filtering, resulting final 83,141 instances in the CodeQG dataset.

In further discussion, we refer to the instance part consisting of code, code summary, and answer as the source, for they are the input to the QG system. Similarly, we refer to the question as the target, as the goal of the QG system is to output it.

We outlined detailed steps that we followed for data construction below:

1. First, to form data instances for CodeQG, for every code in codeQA, we retrieve the corresponding code summary from the Java code-comment pairs dataset. Therefore, an instance in CodeQG has a source (i.e., code and code summary, answer) and target (questions).

2. Then, we remove the instances containing ‘yes’ as the answer. It is difficult even for humans to look for only answers as ‘yes’ and generate the question; thus, such instances are unsuitable for training question generation systems.

3. Next, we remove any duplicate instances. We consider instance duplicates if both the source and target or only the source are the same.

4. We discard instances whose target’s(questions) tokens count is greater than 50 or the answer’s count is greater than 25. We used RobertaTokenizer(ref) to tokenize.

5. Similarly, we eliminate cases in which the source tokens count (total tokens count for code, code summary, and answer) exceeds 400.
6. Then, we shuffled the data instances but grouped them by code; we shuffled the instances but kept the instances having the same code together.

7. Finally, we split the data instances into train, dev, and test in 8:1:1 ratio.

SQuAD 1.1

Stanford Question Answering Dataset, version 1.1, SQuAD 1.1 (Rajpurkar et al., 2016), is a general-purpose reading comprehension dataset consisting of 100,000+ questions posed by crowd-workers on a set of Wikipedia articles, where the answer to each question is a segment of text from the corresponding reading passage. We use SQuAD to train one of the models to compare its performance with the model trained on CodeQG (programming domain-specific dataset).

Note that the latest version of the dataset, SQuAD 2.0 (Rajpurkar, Jia, & Liang, 2018), contains additional questions whose answers are not necessarily segment of text or span from the reading passage. However, for our purpose, we wanted an answer to be a segment of text from the reading passage (context). Thus, we used SQuAD 1.1 rather than the latest SQuAD 2.0.

The SQuAD is originally designed to train a question-answering system where the source is the question and the passage, and the target is the answer. Our goal is to train the Question Generation (QG) system; thus, we used SQuAD differently; our source input to the QG system is the answer and the passage, whereas the target is to output the question. We only take the training split of the SQuAD dataset. Then we apply some filtering resulting total of 83,593 data instances, which we use to train one of the models. The resultant SQuAD dataset’s relevant statistics, such as minimum, maximum, and average count of tokens for passages, answers, and questions, are summarized in Table 6.4.

The detailed filtering steps that we followed are outlined below:

1. We remove instances with the answer’s tokens count greater than 25.
2. We eliminate instances with tokens count greater than 50 for questions.

3. We discard instances with a question’s tokens counts smaller than 3.

4. We drop cases in which the combined tokens count for the answer and the passage (i.e., source tokens count) exceeds 400.

We only considered using the training split of the SQuAd dataset because we wanted to train one of the models using this general-purpose reading comprehension dataset and compare the model performance with those trained with the programming-specific CodeQG dataset. However, for all models, we opted to use the dev set for CodeQG to select the parameters, tune them and choose the best model of a training algorithm. Similarly, we evaluate the performance for all models using a test split from CodeQG. This is because the dev and test split of the codeQG best represents the target we look to hit or what we want to achieve.

Approach

This section details our approach to developing a model for automatic question generation using target concepts in code explanation. We first define the task, then explain the model architecture, and finally, describe the different models type we developed along with their purpose.

Task

Given the input as code explanation and the target concepts in the explanation, our task is to automatically generate output as the question.

Ideally, code explanation should explain what a code does in detail. The

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passage</td>
<td>26</td>
<td>398</td>
<td>155.18</td>
</tr>
<tr>
<td>Answers</td>
<td>1</td>
<td>25</td>
<td>4.51</td>
</tr>
<tr>
<td>Question</td>
<td>4</td>
<td>46</td>
<td>12.37</td>
</tr>
</tbody>
</table>

Table 6.4 Statistics showing the average, minimum, and maximum count of tokens for the SQUaAD dataset after preprocessing for our training purpose.
target concepts can be keyword(s) or phrases in the code explanation that we expect students to know if they truly comprehend the program well. By generating the questions based on target concepts in the code explanation, the idea is to pose limited concept-based questions instead of asking questions for each sentence in our previous methods.

In the CodeQG dataset, the code summary is equivalent to the code explanation, and answers are the target concepts for the code summary.

**Architecture**

The design of our AQG model utilizes Transformer (Vaswani et al., 2017) architecture. We take the transformer encoder-decoder architecture but initialize the encoder with CodeBERT (Feng et al., 2020). Note that CodeBERT is a bimodal model pre-trained with natural language and programming languages (Java, Python, etc.). Thus, by using it as an encoder, we look to encode better code explanation that consists of both natural language and programming languages vocabulary.

We train our primary model to predict the question (i.e., target) given the input sequence (i.e., source) of various combinations of the answer (i.e., target concept), code summary (i.e., code explanation), and code. We set the maximum source and target token lengths to 404 and 52, respectively. We train for 20 epochs and select the model that performs best on dev split; we use the BLEU (Papineni, Roukos, Ward, & Zhu, 2002) score as an optimization metric to choose the best model.

We evaluate the performance of the best-performing model on the test split and report it. We use automatic evaluation metrics for evaluation.

**Model Types**

We trained five different models using the same architecture discussed in 6 but with a different purpose. Table 6.5 shows different model types, their input source sequence, and the name of the dataset used for training, development, and
testing purposes for that particular model type. Note that all these model uses
CodeQG for training, development, and testing, except model AP, which uses
SQuAD for training.

Below, we present different models based on their input design and purposes.

**ACs**

The source input sequence to this model is \(<s>Answer</s> Code
Summary</s>\). We proposed this model to automatically generate questions, used
by ITS, for code comprehension using target concepts in code explanation. Here,
the answer is the target concept, and the code summary is the code explanation.

**A**

The source input sequence for this model is \(<s>Answer</s>\). This model
serves as a baseline model when trained on the CodeQG dataset and when only the
answer is provided as information in the input sequence.

**AP**

The source input sequence to this model is \(<s>Answer</s>Passage</s>\). Unlike all other models trained using the CodeQG training set, we train this model
using SQuAD 1.1 train split.

It also serves as another baseline. This model is trained on the
general-purpose dataset SQuAD 1.1, and its performance in the test split of the
CodeQG dataset gives us an idea of how the model performs when trained on the
general-purpose dataset and used for question generation for code comprehension.
By comparing this model AP with model ACs, the primary model we proposed and
trained using CodeQG, we know if CodeQG adds value to the question generation
tasks for code comprehension. We hypothesize that our CodeQG dataset should
increase the model’s performance for QG tasks specific to the programming domain.
ACsCd

The source input sequence to this model is \(<s>Answer</s> Code Summary</s>Code</s>. This model aims to experiment performance effect of adding code in the input sequence to our proposed model ACs.

ACd

The source input sequence to this model is \(<s>Answer</s> Code</s>. The purpose of developing this model is to see how the performance of our primary model ACs differs if the code is used instead of the code summary in its input sequence. To answer it, we can compare the performance of this model, ACd, with ACs.

Results

We evaluate the performance of the different models that we developed using several automatic metrics: BLEU (Papineni et al., 2002), ROUGE-L (Lin, 2004), METEOR (Banerjee & Lavie, 2005), Exact Match (EM), and F1. Table 6.6 summarizes the performances of these models in automatically generating questions for code comprehension.

From the result in table 6.6, we can infer that ACs, the model we proposed, is impressive in automatically generating questions using target concepts for code explanations. It performed best among all models and roughly double compared to the baseline model A. Looking at the model standalone, the score of 9.12, 94.20, 61.62, 77.05, and 94.64 for BLEU, ROUGE, METEOR, EM, and F1, respectively, is

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Input Source Sequence</th>
<th>Train, Dev &amp; Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACs</td>
<td>Answer + Code Summary</td>
<td>CodeQG, CodeQG, CodeQ</td>
</tr>
<tr>
<td>A</td>
<td>Answer</td>
<td>CodeQG, CodeQG, CodeQ</td>
</tr>
<tr>
<td>AP</td>
<td>Answer + Passage</td>
<td>SQuAD, CodeQG, CodeQ</td>
</tr>
<tr>
<td>ACsCd</td>
<td>Answer + Code Summary + Code</td>
<td>CodeQG, CodeQG, CodeQ</td>
</tr>
<tr>
<td>ACd</td>
<td>Answer + Code</td>
<td>CodeQG, CodeQG, CodeQ</td>
</tr>
</tbody>
</table>

Table 6.5 Model Type with their input source sequence and name of source dataset used for train, dev, and test purposes.
very good; in particular, EM signifies that this model did produce more than one-third question exactly matching. These high scores also mean this model successfully captured the ability to generate various question types (what, how, where, when, why, and for what purpose) as in CodeQG and the expert’s knowledge (embedded in the question template they provided). To compare with the historical performance, we compare our model with BART (Tang et al., 2022), which performs best for question generation task in the SQuAD 1.1, the most widely used dataset for QG, with BLEU, METEOR, and ROUGE scores of 25.08, 26.73, and 52.55, respectively. We can see that ACs perform far better than BART in all metrics; Although the direct comparison of ACs and BART is not all fair since they are trained and evaluated on a different dataset, we can at least deduce that our model performs well in the historical context.

In addition, we see that ACs (the model trained on CodeQG) perform roughly three times better than AP (the model trained on general-purpose dataset SQuAD) for automatically generating questions for program comprehension; see Table 6.6. Thus, we validate our hypothesis that our CodeQG dataset increases the model’s performance for QG tasks specific to the programming domain.

By comparing the performance scores of ACs and ACsCd, we find that adding code in addition to code summary in the source input sequence does not improve the performance of our proposed model, ACs. Look at Table 6.6, they perform almost similarly, or ACsCd slightly underperforms ACs. Likewise, we also find that altering our proposed model ACs input sequence by replacing the code

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU</th>
<th>ROUGE</th>
<th>METEOR</th>
<th>EM</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACs</td>
<td>89.12</td>
<td>94.20</td>
<td>61.62</td>
<td>77.05</td>
<td>94.64</td>
</tr>
<tr>
<td>A</td>
<td>39.86</td>
<td>54.58</td>
<td>23.37</td>
<td>19.61</td>
<td>55.37</td>
</tr>
<tr>
<td>AP</td>
<td>22.67</td>
<td>34.21</td>
<td>23.47</td>
<td>0.0</td>
<td>36.86</td>
</tr>
<tr>
<td>ACsCd</td>
<td>88.89</td>
<td>94.06</td>
<td>61.42</td>
<td>76.86</td>
<td>94.99</td>
</tr>
<tr>
<td>ACd</td>
<td>45.90</td>
<td>60.83</td>
<td>27.28</td>
<td>26.63</td>
<td>61.84</td>
</tr>
</tbody>
</table>

Table 6.6 Performance of Different Model Types.
explanations with the code (target concept + code) drops the performance by almost half; see Table 6.6, ACd performs almost half compared to ACs. The fact that ACs and CsCd perform similarly and ACsCd is only slightly better than A suggests that the code contains little information for model training in our CodeQG dataset. This is not surprising given that questions in our dataset were generated using the template that used information only in the code summary and did not use any code information.

Conclusions

As the need for ITS for code comprehension, which uses questions to scaffold students, is growing, it is crucial to develop cost-efficient approaches that automatically generate a balanced number of questions for code comprehension. Similarly, forming a programming domain-specific dataset to train such approaches is urgent.

In this work, we constructed a large dataset, CodeQG, specific to code comprehension. Then, we trained Transformer using the dataset to automatically generate questions using target concepts in code explanations. Our finding shows that our model not only generated a wide variety of impressive questions (BLEU:89, ROUGE: 94, F1:94.64), but the model's performance improved almost triple by training on CodeQG compared to using SQuAD. We also found that using code explanations compared to code is critical in generating questions for code comprehension.

While our approach automatically generated questions using targeted concepts in code summary, the study is required to evaluate the effectiveness of generated questions in scaffolding students for code comprehension while used in ITS. Furthermore, the CodeQG dataset can be extended by human-generated instances, using humans to provide actual code explanations, target concepts, and
questions. Also, it is essential to conduct further research to determine the critical concepts in code explanations that are worth targeting and approach to do so.
Chapter 7

Conclusion and Future Work

As part of our larger effort towards developing a Self-Explanation Based Intelligent Tutoring System for Code Comprehension, the work in this dissertation focuses on and aims to 1) investigate the effectiveness of self-explanation in ITS as a learning/teaching strategy for code comprehension and 2) develop approaches for automatically generating question to be used by ITS to scaffold student for code comprehension.

The main research questions this dissertation aimed to address were as follows:

- **RQ1**: How effective is Free Self-Explanation (i.e., the most basic form of Self-Explanation) as a learning strategy for code comprehension?

- **RQ2**: How effective is the learning/teaching strategy Guided Self-Explanation (i.e., a form of self-explanation used in ITS) compared to Free Self-Explanation for code comprehension?

- **RQ3**: How to automatically generate questions for code comprehension using each sentence in code explanations?

- **RQ4**: How to automatically generate questions for code comprehension using targeted concepts in code explanations?

Two studies were performed to investigate the effectiveness of Self-Explanation in ITS as a teaching/learning strategy for code comprehension. Based on the findings of our first study, it is an effective strategy for the purpose. Our second study indicates that self-explanation, while used in ITS, called Guided Self-explanation, induces more learning gain in learning code comprehension than using the strategy standalone. Furthermore, we developed various approaches for
automatically generating code comprehension questions using each sentence and
target concepts in code explanations. The evaluation result shows that the generated
questions are impressive and can be used in ITS with minimal human review.

Our first study was carried out to investigate the effectiveness of
self-explanation in standalone as a learning/teaching strategy for code
comprehension. We hypothesized that self-explaining the code should help in code
comprehension for the positive effect of the self-explanation strategy is well
documented in the science learning { (Best et al., 2005; Chi et al., 1994; Ramalingam
et al., 2004). For this purpose, we conducted a randomized trial experiment where
the Self-Explanation First (experimental) group was asked to self-explain the code
first and then predict its output. In contrast, the other Prediction First (control)
group was asked to predict the output and then self-explain the code; note that
although this group self-explain too, they predicted the output of the code before
self-explaining. Then, we did an independent t-test between the two groups’ scores
for correctly predicting the output to see the effect of self-explaining the code.
There was a significant difference in the mean score between the Self-Explanation
First (M=2.54) and Prediction First group (M=1.3), p=0.042. The result confirms
our hypothesis that self-explanation is an effective learning/teaching strategy for
code comprehension. We also found a strong positive correlation (r =0.62, n = 23, p
= 0.001) between the volume of self-explanation and learning gain; the more
students self-explain, the better they will comprehend the code.

To compare Free self-Explaantion with Guided Self-Explanation, we carried
out the second randomized trial experiment with 105 participants where they were
randomly and equally assigned to one of the groups: Predict Only, Self-Explanation,
and Guided Self-Explanation. All groups were shown the same code and an equal
number of code examples and comprehended them using the strategy based on
group assignment; the Predict Only group only predicted the output, the
Self-Explanation group explained what the code does, and Guided Self-Explanation used ITS where students were scaffolded to explain the code using series of guiding questions related to critical aspects of the code example. They also did pre-test and post-test, and the learning gain was calculated using them. Then, ANOVA was done to see the difference between the groups. The result showed that the Mean learning gain of prediction only, self-explanation, and guided self-explanation groups is 0.047, 0.30, and 0.59, respectively, and the difference in mean score between each group is significant. While the findings not only reconfirm our hypothesis that self-explanation is an effective strategy for code comprehension, it also brings to light that they can be used more effectively combined with ITS to scaffold students during code comprehension. In addition, we also proposed a model for the mental model analysis of students during code comprehension using their self-explanation. We used it to perform an analysis of the self-explanation we collected. Our analysis showed that students with high prior knowledge focus more on talking about the program’s control flow and reference their prior knowledge than the low prior knowledge group. Thus, training students with low prior knowledge can help them better comprehend the code.

After confirming self-explanation as an effective strategy to use in ITS for code comprehension, we set out to our third work to develop approaches for automatically generating questions for code comprehension using each sentence in code explanations. It was done to automate the generation of questions to be used by ITS to scaffold student self-explanations. We developed two systems, Machine Noun QG and Machine Verb QG, that automatically generate questions for each sentence in the code explanations. While both systems use ProphetNet (Qi et al., 2020) trained on SQuAD, the Machine Noun QG targets noun phrase, and Machine Verb QG targets verb phrases in the sentence. We manually evaluated the generated questions using criteria: grammatical, semantic correctness, helpfulness,
recognizability, answerability, and domain relevancy. We compared the performance of our two systems with humans. The result showed that our systems generated linguistically well-formed questions (Grammaticality and Semantic Correctness), pedagogically sound (Domain Relevancy, Answerability, and Helpfulness), and indistinguishable from human-generated questions (Recognizability). Both systems performed equally well compared to humans in all evaluation criteria. Among the machine, Machine Noun QG performed better.

In our final work, we develop approaches for automatically generating questions using targeted concepts in code explanations instead of using each sentence as in our previous work. Using targeted concepts in code explanations instead of each sentence, we limited the number of questions to the number of targeted concepts, which would otherwise generate too many questions for lengthy code explanations. Unlike previous work, which uses a general reading comprehension dataset SQuAD, we form our programming domain-specific dataset CodeQG and train the Transformer to predict the question for a given sequence of targeted concepts and code explanations. The automatic evaluation results show that this approach not only generated a wide variety of impressive questions (BLEU:89, ROUGE: 94, F1:94.64), but the model’s performance improved almost triple by training on CodeQG compared to using SQuAD. We also found that using code explanations compared to code is critical in generating questions for code comprehension.

Our experiments to examine the effectiveness of self-explanation as a learning strategy for code comprehension were conducted in a laboratory environment where students used the strategy for only 1-2 hours to comprehend 6 java code examples. These six code examples covered only basic topics: operator precedence, nested if-else, for loops, while loops, arrays, and creating objects. To better understand the strategy’s implications, we recommend that further studies
examine the effectiveness of self-explanation for a semester-long to comprehend all
topics covered in the intro-to-programming course. These studies can also be
extended to other programming languages like python for more insights. Likewise,
we developed approaches that automatically generated questions for code
comprehension, which were linguistically well-formed, pedagogically sound, and
indistinguishable from human-generated questions. Further user studies should be
done to evaluate how effective these questions are when used in ITS to scaffold
students during code comprehension.

In this dissertation, we investigated the effectiveness of using the
Self-Explanation strategy in the code comprehension domain, a work not found in
the literature. We found the strategy to be an effective strategy for learning code
too. We also developed approaches for automatically generating questions specific
to code comprehension; only a few such works were done in the past. Our
approaches were able to generate questions acceptable to ITS. Finally, we developed
a large dataset CodeQG specific to the programming domain; this can be used to
push further research in question generation for code comprehension.
References


programming. In *Proceedings of the 18th koli calling international conference on computing education research* (pp. 1–9).


NSF. (n.d). *Nsf award search: Award # 1822816 - collaborative research: Csedpad: Investigating and scaffolding students’ mental models during computer programming tasks to improve learning, engagement, and retention.*


((Accessed on 03/19/2023))


Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., ... Paterson, J. (2007). A survey of literature on the teaching of introductory programming. In Working group reports on iticse on innovation and technology in computer science education (pp. 204–223).


Educational Psychology Review, 21(1), 55–66.

information processing systems, 30.


revisited. In Proceedings of the 2014 conference on innovation & technology in
computer science education (pp. 39–44).

The failure to teach k-12 computer science in the digital age. New York, NY:
The Association for Computing Machinery and the Computer Science
Teachers Association.

The failure to teach k-12 computer science in the digital age. Association for
Computing Machinery, 26.

system. Proceedings of 12th Australian Society for Computers in Learning in

feasibility study of using automated program repair for introductory
programming assignments. In Proceedings of the 2017 11th joint meeting on
foundations of software engineering (pp. 740–751).

query writing procedures.

automatically generate programming exercises. In Proceedings of the 49th acm
technical symposium on computer science education (pp. 14–19).

programming to is students: The impact of teaching approaches on learning

comprehension and memory. Psychological bulletin, 123(2), 162.

Constructing multidimensional situation models during reading. Scientific
studies of reading, 2(3), 199–220.

Zweben, S. (2008). Undergraduate enrollment in computer science trends higher;
doctoral production continues at peak levels. computing degree and enrollment
trends from the 2007-2008 cra taulbee survey (Tech. Rep.). Technical report,