2021

Expert-Generated and Auto-Generated Socratic Tutoring Systems For Code Comprehension

Zeyad Alshaikh

Recommended Citation

Alshaikh, Zeyad, "Expert-Generated and Auto-Generated Socratic Tutoring Systems For Code Comprehension" (2021). *Electronic Theses and Dissertations*. 2420.
https://digitalcommons.memphis.edu/etd/2420

EXPERT-GENERATED AND AUTO-GENERATED SOCRATIC TUTORING SYSTEMS FOR CODE COMPREHENSION

by

Zeyad Alshaikh

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Major: Computer Science

The University of Memphis

August 2021

DEDICATION

For my parents and grandparents.

ACKNOWLEDGMENTS

I would like to recognize my advisor and committee chair, Dr. Vasile Rus, for mentoring me through the years and for always having time for thoughtful discussion and review. His endless support and encouragement greatly enriched my time as a Ph.D. student. I am also deeply grateful to the rest of my committee members, Dr. Scott D. Fleming, Dr. Deepak Venugopal, and Dr. Amy Cook, for their insightful feedback and suggestions.

Over the years, I have had wonderful conversations and relationships with friends and colleagues, including Dr. Nabin Maharjan, Lasang Tamang, and Nisrine Ait Khyi, just to name a few. I thank them all for their help and collaboration with my research.

Special thanks go to my parents, my wife, and my children for their tremendous support and patience. I also remember the many friends, relatives, teachers, mentors, and colleagues who have directly or indirectly helped me throughout this journey.

ABSTRACT

Alshaikh, Zeyad, Ph.D. The University of Memphis. June, 2021.
Expert-Generated and Auto-Generated Socratic Tutorial Dialogue to Increase Programming Comprehension. Major Professor: Vasile Rus, Ph.D.

Programming skills are a vital part of many disciplines but can be challenging to teach and learn. Thus, the programming courses are considered difficult and a major stumbling block. To overcome these challenges, students could benefit from extensive individual support such as tutoring, but there are simply not enough qualified tutors available to meet rising demands.

A potential solution is the development of intelligent tutoring systems (ITSs), which offer individualized, one-on-one instruction. Such systems can offer the support to make programming instruction more effective, scalable and reduce existing teachers' workloads.

This dissertation demonstrates how conversational ITSs and the Socratic method of teaching can improve a novice's understanding of programming concepts and, in particular, the scaffolding of code comprehension processes. Furthermore, this work provides a novel method to automatically author a Socratic dialogue-based ITS. Indeed, two major outcomes of this work are a Socratic dialogue-based ITS and an automated dialogue authoring tool, which generates full Socratic dialogue from Java source code.

The key objectives of this dissertation were, first, to determine whether the Socratic method would be effective at eliciting learners to engage in self-explanations with the help of the Socratic Tutor ITS and, second, to assess the quality of Socratic Author's auto-generated tutorial dialogue. Thus, the work presented here sought to answer two main research questions: (1) can a Socratic ITS lead to improved code comprehension? and (2) to what extent can Socratic dialogue be generated automatically?

In sum, this research helps establish a relationship between code comprehension and the use of the Socratic method in learning computer programming. Furthermore, the work introduces a novel approach for generating Socratic dialogue from source code with examples for the Java programming language. The auto-authoring tool could help teachers and ITS developers create tutorial dialogues automatically from Java code without requiring non–domain knowledge. To the best of our knowledge, no such auto-generation of tutorial dialogues from source code has been done before and thus constituting a premiere.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer programming has become an increasingly integral part of many disciplines, such as math, science, and engineering. Despite this growing importance, introductory programming courses routinely suffer from high rates (e.g., 30-40%) of dropout and failure (Beaubouef & Mason, 2005; Bennedsen & Caspersen, 2007, 2019; Mcgettrick et al., 2005; Petersen et al., 2016; Robins et al., 2003). Such courses are often cited as being difficult, frustrating, and a stumbling block for students (Lane & VanLehn, 2004; Piteira & Costa, 2013; Proulx, 2000). A major reason computer programming is a challenging subject to teach or learn is its complexity and the need for extensive practice (Medeiros et al., 2018).

The process of computer programming can be broken down into three main phases (problem-solving and design, implementation, and testing and debugging), with each phase requiring different skills. The literature has shown that students struggle in each phase (Bain & Barnes, 2014; Bosse & Gerosa, 2017; Lane & VanLehn, 2003; Truong et al., 2005).

The problem-solving and design phase requires logic, abstract thinking, and domain knowledge. However, novice programmers often fail to engage in meaningful planning and instead jump straight to the implementation phase (Lane & VanLehn, 2004). As a result, (C.-K. Chang, 2014) and (Tom, 2015) found that many of the bugs in students' programs were not related to the programming language but were instead the result of conceptual understanding of problem-solving strategies. Moreover, a one-year analysis of students' assignments revealed that 22% of programming bugs were related to problem-solving skills (Bryce et al., 2010). Furthermore, a study by Bain and Barnes (2014) showed that 50% of students had no problem-solving strategies beyond looking up individual problems online as they arose.

The implementation phase requires knowledge of syntax and semantics involving memory and compression processes (Renumol et al., 2009). Writing a computer program (i.e., implementation) requires knowledge of programming concepts, e.g., variables, conditions, and loops, as well as knowledge of the syntax and semantics of the target language. In this phase, students implement and test out what they have learned in the classroom. This phase is also problematic; e.g., (Lane & VanLehn, 2003) found that students struggled to write a fully functional program without help, and (Bryce et al., 2010) showed that 78% of students' programming bugs were due to a combination of logic and syntax problems.

The final phase of the programming process is testing and debugging, where students focus on validating their program's outputs and fixing any bugs. Debugging a program requires a solid knowledge of the program's operation and the programming language as well as problem solving skills (Bryce et al., 2010). Therefore, novice programmers are often wasting hours trying to fix simple bugs and understanding error messages. As a result, debugging has become known to be difficult for novice programmers and to be a cause of frustration and attrition for many students (Alqadi & Maletic, 2017).

As noted above, computer programming cannot be effectively learned without extensive practice (Núñez et al., 2008), with students needing to write many programs and solve many exercises before they can be considered proficient. During this learning process, students need ongoing individual advice and prompt feedback from experts (Hattori & Ishii, 1999) that is tailored to their needs based on constructivist theories of learning. However, with more and more students taking programming courses, few qualified teachers (Shaffer, 2005), and limited school budgets, individual attention for all students is difficult to achieve using traditional means (A. T. Corbett et al., 2001; Xu & Sarrafzadeh, 2004). In addition, students often practice at home or take online or distance learning courses, which provide limited teacher interaction and assistance. Therefore, there is a need for education technology that offers individualized instruction whenever and wherever the student needs it (Wang, 2011; Weber & Brusilovsky, 2001).

Tutoring is one of the most effective forms of instruction. It has been demonstrated to produce mean learning gains ranging from 0.4–0.9 (with non-expert, cross-aged, or peer tutors) to 0.8–2.3 standard deviations (with expert tutors) over students only receiving traditional classroom instruction (Graesser et al., 2009; Person et al., 2007; VanLehn et al., 2007). Encouraged by the effectiveness of one-on-one human tutoring, computer programs that mimic human tutors have been built to try to ensure every student has access to tutoring. Developing such programs is important because it is logistically impossible to provide all students who need individualized help with a human tutor because there are millions of students all over the world attending introductory programming courses (Bennedsen & Caspersen, 2007), and there are not enough human tutors available for everyone (Nguyen et al., 2014). Teachers serving large programming classes also cannot afford to offer individualized instruction to all struggling students.

Studies have shown that intelligent tutoring systems (ITSs) can provide all students with individualized, one-on-one tutoring, improve the quality and effectiveness of computer programming instruction, and reduce teacher workloads (Truong et al., 2005; Pillay, 2003). As a result, many ITSs were developed as early as 1975 to aid students with different programming phases. For instance, Automatic Tutor (Danielson & Nievergelt, 1975) and xTEx-Sys for programming (Dadic et al., 2006) were meant to help students with planning and design, whereas systems such as PROUST (Johnson & Soloway, 1985) were designed to help students find logic and syntax errors. In addition, some ITSs have targeted one or more programming concepts; e.g., Meno-ii (E. M. Soloway et al., 1981) and RAPTIS (Woods & Warren,

1995) focus on the loop concept, while ProPL (Lane & VanLehn, 2004) targets conditions and loops.

Despite these advantages, creating content for ITSs is a challenge and requires expertise in cognitive science and programming. Furthermore, constructing such content is tedious, error-prone, and time intensive (Heffernan et al., 2006). Therefore, many authoring systems have been developed in different domains, such as physics, mathematics, and public policy, to increase the accessibility and affordability of authoring ITSs (Heffernan et al., 2006). One successful example of an authoring system was developed by (Jordan et al., 2001) that reduced the development time to three months. The system used a graphical interface for teachers to construct tutoring dialogue about physics. Another successful example developed by (V. Aleven et al., 2009) uses a graphical user interface to speed up the development of instructional components, such as hints and just-in-time messages.

Research Goals

This work aims to explore an application of natural language technology in an effort to help novice programmers develop a deeper understanding of programming concepts. The findings demonstrate how natural language technology and guided self-explanation through the Socratic method of teaching can improve understanding of programming concepts. The study has produced a Socratic dialogue-based ITS called “Socratic Tutor” and an automated dialogue authoring tool called “Socratic Author” that generates full Socratic dialogue from Java source code.

Research Questions

The research questions that guided this study were (1) can a Socratic ITS lead to improved code comprehension? and (2) to what extent can Socratic dialogue be developed automatically? Therefore, the detailed questions presented in Table 1 focus on the Socratic method’s effectiveness in teaching programming by using an

Table 1: Research questions for the ITS (Socratic Tutor)

Question
How much do students improve their learning of basic programming concepts when using Socratic Tutor?
How much can Socratic Tutor help students with low prior programming knowledge improve their mastery of basic programming concepts?
How much do students learn about each targeted programming concept when using Socratic Tutor?
What is the relationship between Socratic Tutor's scaffolding questions and learning gains?
How does the type of scaffolding questions affect students' learning processes?
How much do students explain the behavior of Java code?
What is the relationship between the self-explanation volume of Java code and learning gains?
What is the relationship between the self-explanation volume of Java code and quality?
How does self-efficacy affect students' learning gains?
How much does Socratic Tutor affect students' self-confidence?

ITS (Socratic Tutor), while Table 2 shows detailed questions to assess the effectiveness of the authoring tool (Socratic Author).

Initial Contributions and Future Work

This study developed a novel Socratic-based ITS to assist learners with understanding code. Therefore, the findings could help establish a relationship between code comprehension and the use of the Socratic method in computer science as well as a relationship between learning gains and student responses in open-ended question dialogue. Furthermore, the work introduces a novel approach for generating Socratic dialogue from Java source code. This approach could help teachers and ITS developers automatically create dialogue that does not require non–domain knowledge and can be archived by entering Java code and clicking a generation button.

Chapter 2 describes the related research on ITSs and authoring tools, examines previous efforts in the computer programming field, and illustrates the need for ITSs and authoring tools that focus on source code comprehension.

Table 2: Research questions for the automated authoring tool (Socratic Author)

Question
Can Socratic Author generate syntactically correct questions, , scaffolding questions, and answers?
Can Socratic Author generate semantically correct questions, scaffolding questions, and answers?
Can Socratic Author generate coherent and consistent dialogue?
Can Socratic Author generate dialogue that covers all important programming concepts presented in the code?
How much do students learn when using auto-generated dialogue compared to expert-generated dialogue?
How effectively does the auto-generated dialogue help students understand code examples?
How effectively does the auto-generated dialogue help students understand core programming concepts?

Chapter 3 presents the novel Socratic Tutor ITS for source code comprehension and describes the general architecture of the system as well as the implementation details of each model and how they interact with each other. It also presents preliminary findings from an experiment on introductory computer programming students. Chapter 4 illustrates the findings from a controlled experiment on introductory computer programming students, while Chapter 5 describes the architecture and design of the novel authoring tool and plans for future work. It also presents preliminary findings from an experiment on introductory computer programming students. Chapter 6 evaluates the authoring tool, and Chapter 7 concludes the study with suggestions for future directions.

Chapter 2

Related Research

Intelligent tutoring systems (ITSs) can provide individualized, one-on-one instruction for all students, improve the quality of programming education, and reduce teacher workloads (Anderson & Skwarecki, 1986; Pillay, 2003; Truong et al., 2005). The literature shows many successful examples of ITSs for computer programming, such as OOPS (Gálvez et al., 2009) and ITEM/IP (Brusilovsky, 1992), as well as for other domains, such as science and physics (Rus et al., 2013).

This chapter provides a literature review surveying 91 programming ITSs, focusing on 44 that contain the four main modules they typically use: domain expert, student, pedagogy, and interface. First, the chapter presents an overview of the architecture of an ITS and the main modules used by most programming ITSs. It then discusses each module in detail, highlighting the most commonly used techniques. After that, it illustrates the domain coverage of the most popular ITSs and reviews the types of feedback provided by programming ITSs. Finally, it summarizes the effectiveness and performance results of ITSs in terms of learning gains and other metrics.

Intelligent Tutoring Systems for Programming

An ITS is a computer-based system that delivers instruction in a flexible, personalized way by simulating one-on-one human tutoring (Butz et al., 2006). The ITS relies on ideas, theories, techniques, and technologies from such domains as artificial intelligence, computational linguistics, cognitive science, and education (Sleeman & Brown, 1982; Rus et al., 2013). A general architecture for an ITS was proposed by Wenger (1986), as shown in Figure 1, which was followed here because it has been widely used. The Wenger architecture consists of four main components: a domain expert module, a student module, a pedagogical module, and an interface module. Other architectures have been proposed, such as the one by Pillay (2003),

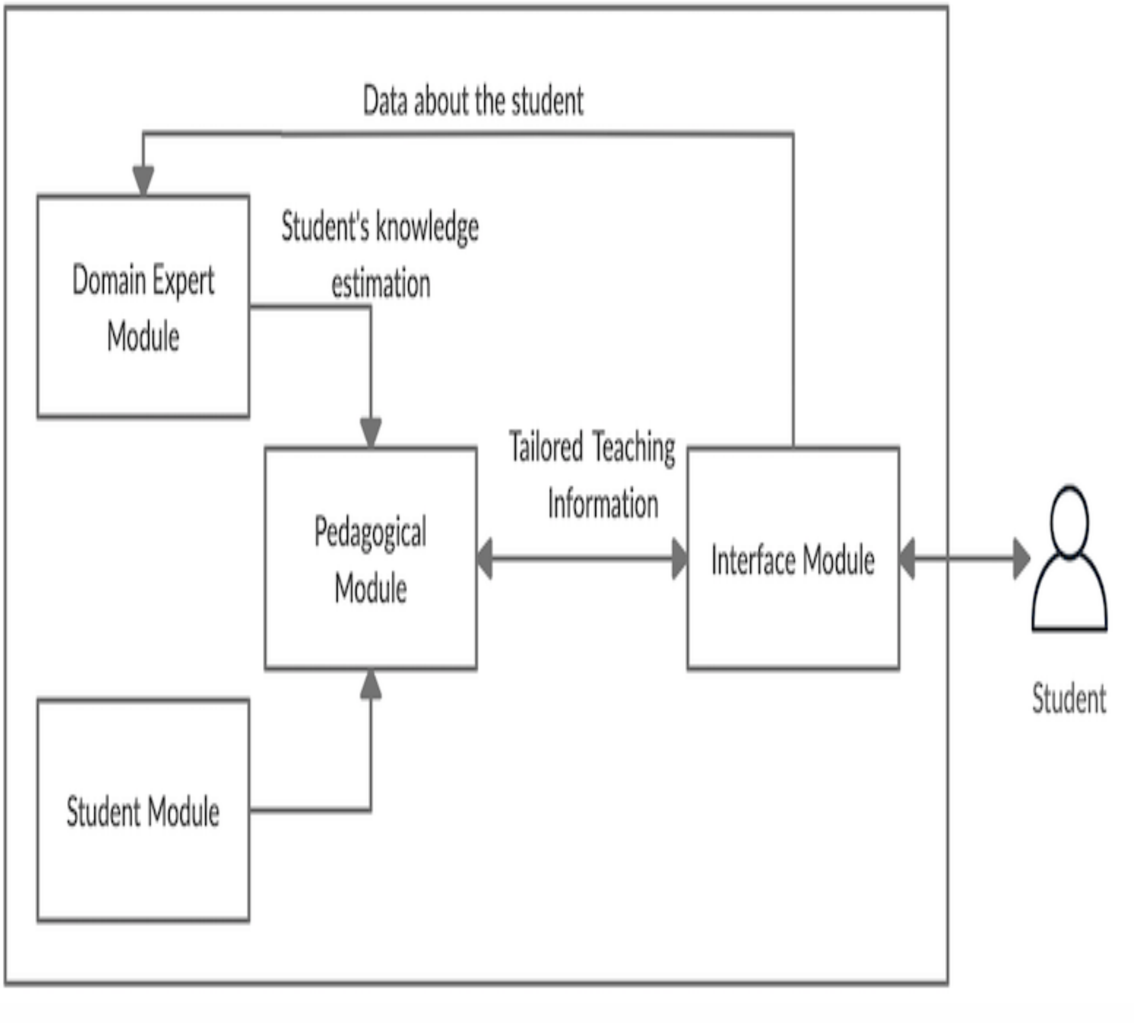


Figure 1. Intelligent tutoring system architecture

which was explicitly designed for a programming ITS and includes four more modules: code specification, explanations, problems, and learning/experience.

The following sections discuss the main modules of a generic ITS in detail and illustrate the most commonly used techniques.

Domain Expert Module

The domain expert module represents expertise in a domain or domain knowledge (Song et al., 1997). In a programming ITS, this module should facilitate the understanding of student responses about programs. Furthermore, the module should give a computational representation of the core knowledge a student is supposed to acquire, including basic programming concepts, e.g., loops, conditions, and the syntax of a specific programming language (Song et al., 1997). More than 16 techniques have been used to model the domain expert module for teaching programming. The sections below discuss the most commonly used techniques, as shown in Table 3.

Curriculum Information Network (CIN)

Curriculum information network (CIN) was introduced by (Wescourt et al., 1977) to store and represent the relationships between course materials. The learning material is composed of programming concepts and skills that students need to master, and a group of skills defines a programming technique. Each task in the network is linked to one or more skills that are required to solve the task.

The main advantage of the CIN is generalization, as the model can be used in other domains, such as physics, math, or statistics. Furthermore, the model was created by experts in the field with a clear understanding of basic concepts and their relationships. These factors make it easier to create a well-structured curriculum and learning path, which can increase student motivation (Blank et al., 2005). However, having an expert write all of this material can be more time-consuming than using automated techniques. By looking at systems that have implemented CIN, such as BIP (Barr et al., 1976) and CIMEL ITS (Blank et al., 2005), it is clear they have only included a small subset of programming concepts.

Constraint-Based Modeling (CBM)

Constraint-based modeling (CBM) was introduced by (Ohlsson, 1994) for domain modeling based on learning from errors theory. This approach helps students who lack declarative knowledge to identify their errors. Systems such as OOPs (Gálvez et al., 2009), J-LATTE (Holland et al., 2009), and SQL-Tutor (Mitrović, 1998) have used CBM in creating the domain expert module. The model is constructed by using constraints that illustrate features of correct solutions, where each constraint follows the form “IF *Cr* is satisfied, THEN *Cs* should also be satisfied; otherwise, a principle is being violated.” The two constraints *Cr* and *Cs* are the relevance condition and the satisfaction condition, where *Cr* represents the status of the problem and *Cs* represents the feature that the solution cannot violate.

The most notable advantage of CBM is the ability to handle different correct solutions through a set of constraints that each correct solution must adhere to (Holland et al., 2009). In addition, CBM is computationally efficient but needs to be manually constructed, and it is unclear whether CBM can provide the right abstraction for different domains (Mitrović, 1998).

Semantic Network

A semantic network is a graph structure representing knowledge, where vertices represent knowledge and edges represent semantic relationships (Liu et al., 2011; Sowa, 1992). According to Sowa (1992), there are many types of semantic networks, depending on how they are used, but they all share the same goal: supporting systems for automated reasoning. Out of the many types of semantic networks, Sowa (1992) listed the six most common: definitional net, assertional net, implicational net, executable net, learning net, and hybrid net.

A semantic network is simple to implement and understand, especially with the help of graphical representation tools. Thus, such a network can be built automatically and is easily translated to Prolog (Rashid, 2015). However, the network makes no distinction between individuals and classes, and the links in the graph only represent a binary relation with no standard definition of link name (Sowa, 1992).

Table 3: Intelligent Tutoring Systems for Programming

System	Year	Domain model	Student model	Assessment method	Interaction form	Feedback	Authoring of content
Automatic Tutor (Danielson & Nievergelt, 1975)	1975	AND-OR graph	-	Graph traversal	Dialogue	Hints & comments	Expert
BIP (Barr et al., 1976)	1976	CIN	Skill list	Output testing	Terminal-like	Hints, demo, model answers, flow charts, & error msg.	Expert
Meno-ii (E. M. Soloway et al., 1981)	1981	KN	-	Runs the program against PDL	Terminal-like	Error msg. with explanation	Expert
TURTLE Tutor (Miller et al., 1982)	1982	CS	Progress record	Path comparison	Game-like	Hints	Expert
LISP Tutor (Anderson & Skwarecki, 1986)	1986	Planning steps	-	Next step simulation	Dialogue	Error msg.	Expert
Bridge (Bonar & Cunningham, 1988)	1988	Plans + requirements	Plan list	Plans Evaluation	Menus + drag & drop	Hints & error correction	Expert
PROUST (Johnson, 1990)	1990	PRN	-	Intention-based analysis	terminal-like	Semantic & logical error msg.	Expert
ITS ADA (DeLooze, 1991)	1991	List of topics + exercises	Overlay model	Output testing	Terminal-like	-	Expert
INTELLITUTOR (Ueno, 1991)	1991	HPC	-	Compares input program with HPC	Terminal-like	Logical error msg. & reading material	Expert
ITEM/IP (Brusilovsky, 1992)	1992	CN	Overlay model	Output testing	IDE-like	Hints, plan, & visualization	Expert
MoleHill (Singley et al., 1993)	1993	Goals Tree	Goal list	Goal evaluation	Multi-function buttons	Color-coded goals	-
(Kay & Kummerfeld, 1994)	1994	CL	-	-	Learning material	-	-

System	Year	Domain model	Student model	Assessment method	Interaction form	Feedback	Authoring of content
Hyperex (Altamura & Roselli, 1995)	1995	HN	KN	MCQ	-	-	-
RAPTIS (Woods & Warren, 1995)	1995	ebook + learning path	progress record + learning path	-	Fill-in-the-blanks + MCQ	-	Expert
Duke Programming Tutor (Keim et al., 1997)	1997	SN	-	-	Dialogue	Hints	Expert
C-Tutor (Song et al., 1997)	1997	CIN	Overlay model	Reverse engineering (GOES) + didactical system (ExBug)	Terminal-like	Efficiency of input program	-
ELM-ART (Weber & Brusilovsky, 2001)	2001	CN	Overlay model	MCQ + syntax error	Fill-in-the-blanks + MCQ + text input	Hints, related material, & error msg.	Teacher
Pseudocode Tutor (Lane & VanLehn, 2003)	2003	-	-	MCQ + syntax error	Dialogue + draggable tiles	-	-
JITS (Sykes & Franek, 2003)	2003	-	-	-	Fill-in-the-blanks	Hints & comments on errors	-
ProPL (Lane & VanLehn, 2004)	2004	Knowledge-construction dialogue	-	Output testing	Dialogue	Hints, examples, & references to material	Teacher & semi-automatic
(Muansuwan et al., 2004)	2004	PS	BN	-	MCQ + fill-in-the-blanks	-	Teacher
CIMEL ITS (Blank et al., 2005)	2005	CIN	BN	Matching UML components	Eclipse plugin	Reading material	-
BITS (Butz et al., 2006)	2006	DAG	BN	MCQ	Reading material + MCQ	Correct or incorrect answer	Teacher
xTEx-Sys for programming							

System	Year	Domain model	Student model	Assessment method	Interaction form	Feedback	Authoring of content
(Dadic et al., 2006)	2006	SN	Progress record	Output testing and comparing with model solution	MCQ + drag & drop	-	-
(Sierra et al., 2007)	2007	CN	-	-	Reading material	-	Expert
CPP-Tutor (Naser, 2008a)	2008	-	-	Pattern match + edit distance	IDE-like	Correctness suggestions & view model solution	-
M-PLAT (Núñez et al., 2008)	2008	Online documentation	ER Database	White and black-box testing	Reading material + online compiler	Correct, syntax error, or result error	Teacher
JEE-Tutor (Naser, 2008b)	2008	-	-	Output testing	Fill-in-the-blanks	Hints and error msg.	-
(Dadic et al., 2008)	2008	SN	ER Database	Hartman's decomposition + constraint satisfaction + semantic analysis	IDE-like	Syntax and semantic error msg., solution, learning material, hints, and description of goals and plans	-
OOPS (Gálvez et al., 2009)	2009	CBM	CBM + score	JESS (Java Expert System Shell)	Drag & drop	Hints and comments	Teacher
J-LATTE (Holland et al., 2009)	2009	CBM	-	Constraint satisfaction + syntax and semantic correctness + code style analysis	drag & drop + writing code inside blocks	Hints and error msg.	-
MAG (Vesin et al., 2009)	2009	-	Progress record + misconception list	Output testing	MCQ + fill-in-the-blanks	Hints	Teacher
ProgTool (Goyal, 2011)	2011	-	ER Database	MCQ	Reading material + MCQ	Correct or incorrect answer	-
FLIP (Karkalas & Gutierrez-Santos, 2014)	2014	Rule list	-	Rule testing + alog. comparison	IDE-like	Hints & reading material	Expert

System	Year	Domain model	Student model	Assessment method	Interaction form	Feedback	Authoring of content
(Dobre, 2014)	2014	-	-	NLP word similarity	Learning material + text-based questions	-	-
Hendrix							
(Holmes et al., 2015)	2015	Ontology	Progress record	Multi-step natural language parsing	Question answering system	Model answer & reading material	-
FIT Java Tutor							
(Gross & Pinkwart, 2015)	2015	Resource space	SN	Output + syntax + semantic testing	IDE-like	Four indicator lamps feedback (syntax, executed, terminated on time, output is correct)	Teacher
(Kularbphettong et al., 2015)	2015	Rule list	Rule list	-	Learning material + MCQ	-	Teacher
Protus							
(Vesin et al., 2015)	2015	-	Progress record + learning style	Log of read materials	Reading material	Reading material	-
ADO-Tutor							
(Haddad & Naser, 2017)	2017	Topic list	ER Database	MCQ	Reading material + MCQ	Pass/fail test result	Teacher

* Curriculum Information Network (CIN) *Knowledge Network (KN) * Curriculum Structure (CS) *Program Requirements Network (PRN) *Hierarchical Procedure Graph (HPC)
*Concept Network (CN) * Concepts List (CL) * Hypertext Network (HN) *Program Specifications (PS) *Directed Acyclic Graph (DAG) *Constraint-Based Model (CBM) *Multiple-Choice Question (MCQ)

Table 4: Intelligent tutoring system extra information

System	Year	Prog. language	Learning env.	Macro-adaptivity	Micro-adaptivity	Experimental result	Domain independent
Automatic Tutor	1975	Lang. Independent	×	-	✓	-	✓
BIP	1976	BASIC	×	✓	✓	✓	✓
Meno-ii	1981	Pascal	×	-	-	-	-
TURTLE Tutor	1982	LOGO	✓	-	-	-	-
LISP Tutor	1986	Lisp	✓	-	-	✓	✓
Bridge	1988	Pascal	×	✓	-	✓	✓
PROUST	1990	Pascal	×	-	-	✓	✓
ITS ADA	1991	Ada	✓	✓	-	-	✓
INTELLITUTOR	1991	Pascal	×	-	-	-	-
ITEM/IP	1992	Mini-language Turingal	✓	✓	✓	✓	✓
MoleHill	1993	Smalltalk	×	✓	-	✓	✓
(Kay & Kummerfeld, 1994)	1994	C	✓	✓	-	-	✓
Hyperex	1995	Pascal	✓	✓	-	✓	-
RAPTIS	1995	Pascal	✓	✓	-	-	✓
Duke Programming Tutor	1997	Lang. Independent	×	-	✓	✓	-
C-Tutor	1997	C	✓	✓	-	✓	-
ELM-ART	2001	Lisp	✓	✓	-	✓	-
Pseudocode Tutor	2003	Lang. Independent	×	-	-	✓	-
JITS	2003	JAVA	×	-	-	-	-
ProPL	2004	Lang. Independent	×	-	✓	-	-
(Muansuwan et al., 2004)	2004	C	×	-	-	-	-
Haskell-Tutor	2004	Haskell	×	✓	-	-	-
Prolog Tutor	2004	Prolog	×	✓	✓	-	✓
(C. Lee & Baba, 2005)	2005	C++	✓	✓	-	✓	-
CIMEL ITS	2005	Java	✓	-	✓	-	✓
BITS	2006	C++	✓	✓	-	-	✓
xTEx-Sys for programming	2006	Lang. Independent	×	-	-	-	-
(Sierra et al., 2007)	2007	Java, C++, Perl	✓	-	-	-	✓
CPP-Tutor	2008	C++	×	-	-	-	-
M-PLAT	2008	Java, C, C++	✓	✓	-	✓	✓
JEE-Tutor	2008	Java	×	-	-	-	✓
(Dadic et al., 2008)	2008	Lang. Independent	✓	✓	-	✓	✓
OOPS	2009	Pseudo-language	×	✓	-	✓	✓
J-LATTE	2009	Java	×	-	-	✓	-

System	Year	Prog. language	Learning env.	Macro-adaptivity	Micro-adaptivity	Experimental result	Domain independent
MAG	2009	Java	✓	✓	-	-	-
ProgTool	2011	Lang. Independent	✓	✓	-	-	✓
FLIP	2014	JavaScript	×	-	-	-	-
(Dobre, 2014)	2014	C	✓	-	-	-	-
Hendrix	2015	Java	×	✓	✓	✓	✓
FIT Java Tutor	2015	Java	✓	✓	-	✓	✓
(Kularbphettong et al., 2015)	2015	Java	✓	✓	-	✓	-
Protus	2015	Java	✓	✓	-	-	-
ADO-Tutor	2017	ADO	✓	✓	-	✓	-

Student Module

The student module represents a student's knowledge status, learning style, and other relevant information about the student, such as their emotional state (Xu & Sarrafzadeh, 2004; Song et al., 1997; Rus et al., 2013).

In a programming ITS, the student module should represent the student's current knowledge of programming language concepts and point out any misconceptions based on the student's answers (Song et al., 1997). According to (Kass, 1989), building a successful tutoring system depends heavily on correct user modeling. The literature shows that many techniques are used to create student modules. The most common are detailed below.

Overlay Model

The most commonly used technique for modeling student knowledge is the Overlay model. This model was introduced by (Stansfield et al., 1976) based on the assumption that the student has partial knowledge of the domain. The model is a subset of the domain module with additional items, such as the degree of mastering concepts. The name Overlay comes from imagining the student module as a paper laid over the domain module, where any hole in the student module represents a misconception where the student lacks knowledge (VanLehn, 1988).

C-Tutor (Song et al., 1997) is an example of a programming ITS that uses the Overlay model, where the student module is a duplicated version of the domain module with an extra slot called "studied." This slot is marked when students finish studying the materials to indicate they have learned the concept. ELM-ART uses a more sophisticated multi-layer Overlay model that consists of Visited Status, Learned Status, Inferred Status, and Known Status. Visited Status indicates whether the student has visited the page, Learned Status shows whether the student has successfully solved the unit's tests and exercises, Inferred Status decides whether the unit can be marked as “known” from previously learned units, and Known Status determines whether the unit has been noted as “known.”

The Overlay model is a simple, efficient way of modeling student knowledge, and as a result, many programming ITSs have used it (Brusilovsky, 1992; Cabada et al., 2015; DeLooze, 1991; Song et al., 1997). However, this model is inadequate for sophisticated modules because it does not consider how the student makes inferences (Rivers, 1989) and does not integrate incorrect knowledge, student preferences, and different cognitive needs (Chrysafiadi & Virvou, 2013).

Bayesian Network

Bayesian Network is a well-known framework that is used to reason about uncertainty in student modules (Conati et al., 2002). A Bayesian network is a direct acyclic graph where nodes represent variables of interest and arcs represent dependency and relationships. Many programming ITSs in the literature have used this network in their student module. For instance, BITS (Butz et al., 2006) uses a Bayesian network to suggest learning goals and generate a suitable reading sequence for each student. Other examples are (C. Lee & Baba, 2005) and CIMEL ITS (Blank et al., 2005), where (C. Lee & Baba, 2005) is designed to teach students how to use C++ STL, and CIMEL ITS tutors students on the “objects-first” approach.

A Bayesian network is frequently employed to reason about uncertainty because of its high representative power, flexibility, and well-defined formalism (Desmarais & Baker, 2012). However, it also contains a large number of hidden nodes such as skills, concepts, and misconceptions (Desmarais & Baker, 2012) that are never directly observed (Hastie et al., 2001).

Initialization of Student Module

The primary purpose of the student module is to adapt to each student’s needs, and thus, when a student is new to the system, the student module uses an initialization phase to assess the student’s knowledge. The most commonly used method to set up the student module involves giving the student an initial test. For instance, (C. Lee & Baba, 2005; Weber & Brusilovsky, 2001) gave an initial test to set up the student model. However, systems such as BIP (Barr et al., 1976) and Haskell-Tutor (Xu & Sarrafzadeh, 2004) let students estimate their own knowledge level. Furthermore, some ITSs, such as ELM-ART (Weber & Brusilovsky, 2001), allow students to check the system estimation and modify it.

Interface Module

The interface module controls interactions and information exchange to facilitate communication between the student and the other modules (Pillay, 2003; Xu & Sarrafzadeh, 2004). The fundamental focus of this module is what information to present and how to present it to the student (Pillay, 2003). According to (Anderson & Skwarecki, 1986), the effectiveness of an ITS depends heavily on the design of the interface module. The literature on programming ITSs shows many types of interaction, ranging from simple command-line interaction to game-like interfaces. The main types of interaction are outlined below.

Menus and Buttons

Menus and buttons have been adopted by some programming ITSs, such as BRIDGE (Bonar & Cunningham, 1988), OOPs (Gálvez et al., 2009), and J-LATTE (Holland et al., 2009), where the learner practices and solves programming problems using a menu with buttons. For instance, BRIDGE allows the student to construct fully functional programs using a menu selection. The main advantage of this approach is to increase the student's focus by limiting the number of possible actions, while limiting the possible solutions makes automated assessment easier. Furthermore, this interface reduces the burden of learning programming language syntax for novice programmers and allows them to concentrate more on problem-solving (Gálvez et al., 2009). On the other hand, computer programming can only be learned through practice, and the menu and buttons approach can prevent the student from learning syntax. That is, students do not have the chance to make any syntax errors and learn from them. Thus, the menu and buttons approach is more suitable for learning and practicing problem-solving and program design.

Fill-in-the-Blanks

The fill-in-the-blanks approach uses a graphical interface that presents a code with missing statements and asks the student to complete the code. Examples of this type of system are JITS (Sykes & Franek, 2003) and J-LATTE (Holland et al., 2009), which allow the student to fill in a missing code or statement. This approach helps students by directing their attention to a specific location in the code (Sykes & Franek, 2003), reducing the number of possible solutions, making automated code assessment easier, and helping students develop a model for each type of programming problem (Odekirk-Hash & Zachary, 2001). However, the ultimate goal of learning programming is to produce efficient and bug-free code, and such an approach prevents the student from practicing constructing a fully functional program from scratch. This approach is more suitable for tutoring students on learning a specific concept, such as *if* conditions or loops, without distracting the student with the rest of the code.

Natural Language Dialogue

Human tutors often have a conversation with the learner and elicit self-explanations in addition to using other forms of communication and coordination, such as gestures and graphical elements. Eliciting self-explanations has been shown to lead to learning (Chi et al., 1994); therefore, the natural language interface offers an excellent option to interact with the learner and to promote learning by eliciting self-explanations and simulating real student-teacher interaction. Despite this advantage, the literature shows few programming ITSs, such as Automatic Tutor (Danielson & Nievergelt, 1975), Duke Programming Tutor

(Keim et al., 1997), ProPL (Lane & VanLehn, 2004), GENIUS (McCalla & Murtagh, 1985), and Program Enhancement Adviser (PEA) (Moore & Moore, 1995), have used natural language dialogue as part of the interface.

One of the earliest natural language dialogue systems was Duke Programming Tutor (DPT) (Keim et al., 1997). DPT is a multi-modal voice dialogue system that guides students through a standard programming lab to help them construct simple programs. The system uses a semantic network with a feature vector to model the domain and uses a temperature function to steer the dialogue. DPT takes students' input (i.e., a program) and uses simplistic minimum edit distance between their code and the goal program. Therefore, DPT offers task-oriented dialogue that leads a student to achieve a set of goals in the process of writing a simple Pascal program.

Another example of a dialogue-based ITS is ProPL, which focuses on programming design and problem-solving. ProPL uses coached program planning (CPP) (Lane & VanLehn, 2003) as a style of tutoring to elicit problem decomposition from students. Therefore, the tutor gives students a problem and asks them to identify: (1) goal(s), which are declared by the main question of the problem; (2) schema, the method to accomplish the goal(s); and (3) objects, i.e., the data required by the program. The final product from the interaction is pseudocode that can be translated into a program.

The goal of these systems is to help students understand and solve programming tasks with an emphasis on the planning and designing phase of computer programming.

On the other hand, systems such as GENIUS and PEA can be looked at as helping tools. GENIUS interacts with students to help them fix syntax errors by keeping them engaged in a dialogue in the hopes that they can resolve the errors. The interaction is limited to yes/no questions and "I don't know" responses. The PEA system, on the other hand, helps students improve their coding style by providing advice on how to make the code more readable and maintainable. The system can answer students' questions about its change suggestions.

The main advantage of a natural language dialogue approach is that it simulates real teacher-student interaction and offers students the freedom to express their ideas in a natural way. All ITSs in the literature that use this as their primary interaction form focus on problem-solving, design, or debugging.

In contrast, the novel ITS proposed in this work, Socratic Tutor, engages students in natural dialogue interaction similar to a human tutor. The main goal is to test the student's knowledge of programming concepts and clarify any misconceptions. To this end, Socratic Tutor is clearly different from other ITSs in computer programming.

Pedagogical Module

The pedagogical module contains knowledge about what and how to teach. The module decides what the next instructional step will be based on the student's current knowledge as maintained in the student module, their answers, and the instructional goals (Song et al., 1997). For example, the pedagogical model may infer that the student does not understand a target concept and could thus decide to provide more examples, hints, or a different teaching approach.

The most common pedagogical approach in programming ITSs is gradually introducing programming concepts in a specific programming language (Cazzola & Olivares, 2016; P. Lee & Phillips, 1998). An example of a programming ITS that uses this strategy is ELM-ART (Weber & Brusilovsky, 2001), which builds a learning path for every student based on the student's current knowledge and the relevance of the studied topics. Furthermore, ELM-ART offers the student the flexibility of choice by marking the learning units with colored bullets to show the learning status of each unit. For example, a green bullet means the topic is ready to be visited, whereas a red bullet means the student is not ready for that topic.

Another example of such a system is ITEM/IP (Brusilovsky, 1992), which allows the student to choose what to learn by selecting “What next” or “repeat” from the main menu.

Adaptivity

One of the most critical features of any ITS is providing each student with personalized tutoring (Butz et al., 2006) by adapting its instructional strategies based on student characteristics as captured by the student module. To achieve this goal, the system must utilize the student module and adapt according to the student’s performance as inferred from their answers and actions. Adaptivity during tutoring can be broadly divided into macro-adaptivity and micro-adaptivity. Macro-adaptivity indicates the system’s ability to choose the most suitable task for the student to work with (Rus et al., 2014). For instance, when a student does not answer an exercise correctly, ELM-ART randomly selects another task with lower difficulty, and in the case of a correct answer, the system chooses a task with greater difficulty.

Another example proposed by Dadic et al. (2008) allows students to start from the middle of a difficulty-sorted list of exercises. If the student fails to solve a given problem, the system selects another problem from the middle of the easier problem subset. If the student’s answer is correct, the system chooses a problem from the middle of the harder problem subset. Haskell-Tutor (Xu & Sarrafzadeh, 2004) offers macro-adaptivity by offering the student three options: (1) following a predefined sequence, (2) choosing from a sorted list of problems, or (3) letting the system choose based on the student’s knowledge level.

On the other hand, micro-adaptivity indicates the system’s ability to adapt its support features while the student is working on a specific task (Rus et al., 2014). An example of a system using micro-adaptivity is BIP (Barr et al., 1976), where each problem contains subtasks. The subtasks are isolated parts of the main problem that can be solved individually. Therefore, when students are struggling with a problem, the system interferes to help by letting them solve the subtasks until they can solve the complete problem. Another example is ER-Prolog-Tutor (Tchetagni et al., 2005), where the tutor breaks down the main problem into multiple sub-problems. Each sub-problem can be solved independently to achieve the final solution. Table 4 shows a complete list of the ITSs and their adaptivity level.

Feedback

The effectiveness of an ITS depends heavily on the timing and type of feedback given (Naser, 2008a). Timing in this context refers to when the ITS should present feedback, while the type of feedback might include suggested reading materials, vague or specific hints, error messages, and showing the correct answer, as shown in Table 3.

Direct and immediate feedback, i.e., immediately after a misconception has been detected, is generally acceptable to have the best impact. However, immediate feedback can also be a distraction and cause the student not to pay attention (Blank et al., 2005). Another issue with immediate feedback in programming is that the tutor might not have enough context to explain why the code is wrong (Anderson & Skwarecki, 1986).

The timing of feedback can be further classified into continued and on-demand feedback. An example of continued feedback is MoleHill (1993), where the feedback works in the background and labels each goal with a green or red color. Green means the student achieved the goal, while red means the program is buggy or incorrect. Another example proposed by (Gross & Pinkwart, 2015) uses four indicator lamps: (1) the code contains a syntax error, (2) the code can be executed, (3) the program terminates in a predefined time, and (4) the code produces a correct result.

Some other systems provide feedback only in a specific situation, such as (Muansuwan et al., 2004) and Haskel-Tutor (Xu & Sarrafzadeh, 2004), where the system only offers feedback in the following situations: (1) if the student's answer is not correct, (2) if the student asks a question, or (3) if the student requests an explanation. Table 3 shows each system and the type of feedback it provides.

Authoring of Content

The next step after creating a domain expert module is feeding it with content. This process, called authoring of content, usually requires teachers or experts to provide a problem statement and model solution.

Authoring content is a challenge in ITS development and requires expertise in cognitive science and programming. Furthermore, constructing such content is tedious, error-prone, and time intensive (Blessing, 1997; Heffernan et al., 2006). For instance, (Murray, 2003; V. Aleven et al., 2009) estimated that 100 development hours resulted in only 1 hour of tutoring.

Therefore, many authoring systems have been developed in different domains, such as physics, mathematics, and public policy, to increase the accessibility and affordability of authoring ITSs (Heffernan et al., 2006). For instance, an authoring system developed by Jordan et al. (2001) reduced development time to three months. The system used a graphical interface for teachers to construct tutoring dialogue for physics. As another example, Aleven et al. (2009) developed a graphical user interface to speed up the development of instructional components, such as hints and just-in-time messages. However, these systems still require teachers or experts to use a graphical interface to create the content themselves.

In the domain of programming ITSs, there are some exceptions to this tendency, such as JITS (Sykes & Franek, 2003) and C-Tutor (Song et al., 1997). JITS only requires the teacher to provide a problem statement and desired output, while C-Tutor only requires a sample program. C-Tutor uses a goal extraction system (Hahn & Kim, 1997) to extract a problem description automatically.

Domain Coverage

As noted earlier, the programming process consists of multiple phases, including planning, designing, implementing, testing, and debugging code. Novice programmers struggle at every step, and their performance worsens when trying to do multiple steps at once (Du Boulay, 1986). On top of that, programming courses cover a wide range of skills and concepts, so many programming ITSs focus on only one of these phases.

The planning and design phase plays a significant role in solving programming problems. However, novice programmers often underestimate the importance of this phase or jump straight to implementation (Pintrich et al., 1987; Lane & VanLehn, 2004). As a result, many tutoring systems have attempted to address this shortcoming. For instance, Pseudocode Tutor (Lane & VanLehn, 2003) focuses on the problem-solving stage, where a student can use natural language to interact with the tutor. The result of the tutoring session is a set of pseudocode blocks that the student can rearrange.

Some programming ITSs focus on the implementation phase and try to teach syntax and programming concepts in a specific programming language. For example, Meno-ii (E. M. Soloway et al., 1981) and RAPTIS (Woods & Warren, 1995) teach the loop concept in Pascal.

Program testing and debugging is the final phase where the students judge their program. However, about 85% of novice programmers' code contains bugs, which they often have difficulty fixing even when the model answer is presented (Hattori & Ishii, 1999). Therefore, systems such as PROUST (Johnson & Soloway, 1985) try to help students find semantic and logical bugs. Table 3 shows a complete list of the domain coverage for each system.

Programming ITS Effectiveness

The literature contains many successful examples of ITSs that increase the quality of programming education (Pillay, 2003) in terms of student knowledge. For instance, Anderson et al. (1986) showed that students who used Lisp Tutor scored one letter grade higher on their final exam. In addition, (Lane & VanLehn, 2003) found that students who used Pseudocode Tutor committed fewer mistakes and provided more comments in their programs. Another example is ITEM/IP (Brusilovsky, 1992), which reduced learning time from 4-5 weeks to 2-3 weeks and increased student interest in learning programming.

Programming ITS performance has been reported in terms of assessment capabilities, a critical aspect of any fully automated ITS. For instance, C-Tutor was evaluated on 240 programs containing 532 bugs. The system was able to run 93% of the programs and identify 81% of the total bugs. Similarly, ITEM/IP was able to detect 80% of bugs in student programs.

Self-Explanation

Self-explanation is when students explain the target material to themselves while learning, which can make them better learners, i.e., learn more deeply and show higher learning gains. Self-explanation's effectiveness is attributed to its constructive nature, as it activates several cognitive processes, such as generating inferences to fill in missing information and integrating new information with prior knowledge; it also benefits from its meaningfulness to the learner; i.e., self-explanation is self-directed and self-generated, making learning more personally meaningful than when explaining the target content to others (Roy & Chi, 2005). The positive effects of self-explanation on learning have been demonstrated in different science domains, such as biology (Chi et al., 1994), physics (Conati & Vanlehn, 2000), math (V. A. Aleven & Koedinger, 2002), and programming (Bielaczyc et al., 1995).

A series of studies (Bielaczyc et al., 1995; Pirolli & Recker, 1994; Recker & Pirolli, 1990) found that self-explanation helped learning Lisp programming concepts, and 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 were more successful at a program construction task (Visual Basic) compared to those who did not apply the strategy. The effectiveness of self-explanation in programming has also been studied in relation to SQL (Yuasa, 1994), JavaScript (Kwon & Jonassen, 2011), HTML (Kwon et al., 2011), and assembly language (Hung, 2012). (Bielaczyc et al., 1995) showed that university students who underwent explicit training on self-explanation and self-regulation strategies outperformed students in a control group in problem-solving performance.

Code Comprehension

Code comprehension, i.e., understanding computer programs, is a critical skill for 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 from a textbook or provided by the instructor). Furthermore, it has been estimated that software professionals spend at least half of their time analyzing software artifacts in an attempt to comprehend computer source code (O'brien, 2003). Reading code is the most time-consuming activity during software maintenance, consuming 70% of the total lifecycle cost of a software product (Rugaber, 2000; Basili & Boehm, 2001; Buse & Weimer, 2008). (O'brien, 2003) noted 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 will have lasting positive effects on their academic success and future careers.

According to (Schulte et al., 2010), “Comprehension is usually conceptualized as a process in which an individual constructs his or her own mental representation of the program.” Mental (or situation) models are a key component of source code comprehension theories (Shneiderman & Mayer, 1979; Brooks, 1983; Pennington, 1987b; Shaft, 1992; Good, 1999; Burkhardt et al., 2002). It is well accepted that there is a modality-independent higher-level skill of comprehension that involves constructing situation models (Zwaan & Radvansky, 1998). This assertion is based on research showing that participants arrive at similar (although not identical) understandings when presented with textual, visual, and audio descriptions of a situation.

It is also well documented that novice programmers struggle with constructing accurate mental models during key learning activities, such as source code comprehension (E. Soloway & Ehrlich, 1984; Pennington, 1987b; Ramalingam et al., 2004; Margulieux et al., 2012).

This challenge is not surprising given that constructing mental representations is a higher-level comprehension skill, typically engendering a high cognitive load (Kintsch & Walter Kintsch, 1998; Zwaan & Radvansky, 1998; Snow, 2002; Graesser et al., 2011).

According to code comprehension theories, code comprehension involves four major components: (1) an external representation, external views or aids assisting the programmer in comprehending the code; (2) a knowledge base, the programmers’ knowledge; (3) a situation/mental model, i.e., the programmer’s current understanding of the code, which is constantly updated through the assimilation process; and (4) an assimilation process through which the situation model is updated based on the knowledge base, external representation, and current situation model (Shneiderman & Mayer, 1979; Brooks, 1983; Letovsky & Soloway, 1986; Pennington, 1987b; Spohrer & Soloway, 1989; Shaft, 1992; Von Mayrhauser &

Vans, 1995; Good, 1999; Burkhardt et al., 2002; Robins et al., 2003; Schulte, 2008; Schulte et al., 2010). A major problem with traditional programming comprehension models is that they were the result of analyzing expert programmers' comprehension processes as opposed to those of novices, i.e., individuals with no or almost no relevant knowledge. More recently, some researchers have sought to address this issue, such as Schulte et al. (Schulte et al., 2010), who proposed an education comprehension model.

As opposed to professional programmers, for the majority of learners in introductory programming courses, their knowledge base is modest or even non-existent. For such novices, support from external representations, such as the source code itself; related documents, such as textbooks or materials provided by the instructor; and advice from more knowledgeable classmates/peers, experts, or intelligent tutors plays a vital role in mediating the construction of mental models. In the absence of adequate support, students in introductory programming courses struggle, with a negative impact on their self-efficacy, learning, and retention. Many feel overwhelmed and enter a state of "shock" from which they never recover (Du Boulay, 1986; E. Soloway & Spohrer, 2013), resulting in quitting the field, as indicated by high attrition rates. Instructors, peers, and experts are not available to help most of the time. However, ITSs can be available 24/7/365 and can be accessed from anywhere using any device with an Internet connection and a browser.

Given the importance of code comprehension among programming students and the fact that they struggle to build accurate mental models during learning activities, there is an enormous need for individualized support in the form of an ITS. The work described here contributes to this goal of developing ITSs to model and scaffold learners' code comprehension processes and skills in introductory programming courses. Therefore, it should improve their learning and increase retention and graduation rates in computer science programs. In particular, this work focuses on developing a Socratic ITS and automating the authoring process for code comprehension.

Socratic Method of Teaching

The Greek philosopher Socrates used a series of questions to guide students in their process of examining a topic or concept. Socrates believed guided dialogues would help his students better understand a given topic, identify incomplete understanding of key concepts, clarify misunderstandings, and correct misconceptions, resulting in a deeper understanding. Graesser et al. (1995) argued that the remedial aspect of tutor interaction, in which tutor and learner collaboratively improve an initial answer to a problem, is the primary advantage of tutoring over classroom instruction. Chi et al. (2004) advanced a related hypothesis: tutoring enhances students' capacity to reflect iteratively and actively on domain knowledge.

The main characteristic of the Socratic method is to observe and guide without providing answers by presenting questions and lets the student reconsider his or her thinking. The Socratic method consists of five stages wonder (posing question), hypothesis (an answer to the wonder), elenchus or refutation (counterexample), accept/reject and finally act accordingly (Boghossian, 2012).

The Socratic method can be categories as classic and modern version (Delić & Bećirović, 2016). The classic method refers to the early version of Socratic dialogues where the tutor claims not to know the most fundamental principles such as justice. The goal is to get a short answer and achieve an understanding of basic principles. On the other hand, the modern method does not rely entirely on student's answers. Alternatively, it guides the students step by step by a set of set of pre-designed questions to lead the students to a specific idea. The aim of the modern method goes deeper than naming or identifying essential topics by helping students develop specific knowledge about those topics

The Socratic method has been used as a teaching tool in many disciplines such as legal education (Sheppard, 2007), mathematics (K.-E. Chang et al., 2003), Physics (Warnakulasooriya et al., 2006). Jackson (2006) stated that the Socratic method has been the dominant teaching tool since the dawn of modern law school. In mathematics, teachers use the Socratic method to teach topics such as subtraction (K.-E. Chang et al., 1998) and geometry (Goldin et al., 2017).

The Socratic method can be an excellent tool for teaching or a weapon that causes perplexity, humiliation and shame (Boghossian, 2012; Delić & Bećirović, 2016). For instance, professor Kingsfield calling a student who does a bad job answering a question down to the podium, giving him a dime, and stating, "Go call your mother, and tell her you'll never be a lawyer" (Jackson, 2006). To disapprove these claims, Delić and Bećirović (2016) stated that the cause of humiliation, shame and all other negative feelings are due to the lack of knowledge of the Socratic method.

One-on-one instruction has the advantage of engaging most students' attention and interest more than other forms of instruction, such as lecturing or monologues in which the student might not choose to pay attention (VanLehn et al., 2007). Therefore, the Socratic method relies on a so-called direct line of reasoning (K.-E. Chang et al., 2003) that emphasizes directing students' attention to key parts of a learning task, triggering reasoning and explanation processes from students, which have been shown to be extremely beneficial in deep understanding tasks.

ITSs with conversational dialogue represent a special category of educational technologies. These conversational ITSs are founded on explanation-based constructivist theories of learning and the collaborative constructive activities that occur during human tutoring. They have been shown to promote student learning gains up to an impressive effect of 1.64 sigma when compared to students learning the same content in canned text remediation (VanLehn et al., 2007). (Brown &

Kane, 1988; Pressley et al., 1992) reported that students learned more effectively when given the ability to explore and explain their own knowledge. Therefore, Socratic tutoring has been employed by human as well as computer tutors as early as 1977 in Stevens and Collins' WHY system. AutoTutor (Graesser et al., 2004) is another example of such a system. A type of conversational ITS implements a Socratic tutoring style that relies on prompting students to think and provide information in the form of answers to questions. The questions are designed to follow a directed, predefined line of reasoning (Rosé et al., 2001).

Discussion and Conclusion

There is a large and growing need for advanced, personalized technology, such as ITSs, for programming education. The ITS holds the promise of providing personalized education to every learner 24/7. This chapter summarized the research on ITSs for programming. While successes in limited contexts have been produced, there are still many challenges left in terms of developing and deploying ITSs at scale across many types of learners and topics. The current work contributes to this area of research by developing a dialogue-based ITS for programming and an auto-authoring tool.

Chapter 3

Socratic Intelligent Tutoring System

Socratic Tutor is an online dialogue-based intelligent tutoring system (ITS) that helps novices develop a deeper understanding of programming concepts. The main type of instructional task is source code comprehension, where the system shows a Java code example and asks the student to self-explain the code and predict its output. After that, the system guides the student by asking a sequence of questions to elicit self-explanations while the student is reading and trying to understand the major programming concepts presented in the code. This system uses the Socratic method of teaching to provide scaffolding using a three-level feedback loop. On the first level, the system briefly explains the programming concept and asks the student to answer the original question. If the student fails, the system provides help in the form of a fill-in-the-blank question on the second level. This type of question limits the student's answers and draws more attention to the key part of the solution. If the first and second levels fail to elicit a correct response, third-level scaffolding is given in the form of a multiple-choice question. Finally, if the student cannot give the correct answer, the system presents the solution and moves on to the next question.

Architecture

The architecture of Socratic Tutor consists of a student module, a domain module, a pedagogical module, an interface module, and a natural language understanding (NLU) engine (Banjade et al., 2015), as shown in Figure 2. The NLU engine is used to automatically evaluate students' answers, making it possible to tailor the teaching strategy to each learner. For instance, if a learner provides a correct answer, the system presents positive feedback, such as "great job" or "excellent answer." In the case of an incomplete or incorrect answer, the system provides scaffolding in the form of Socratic questions.

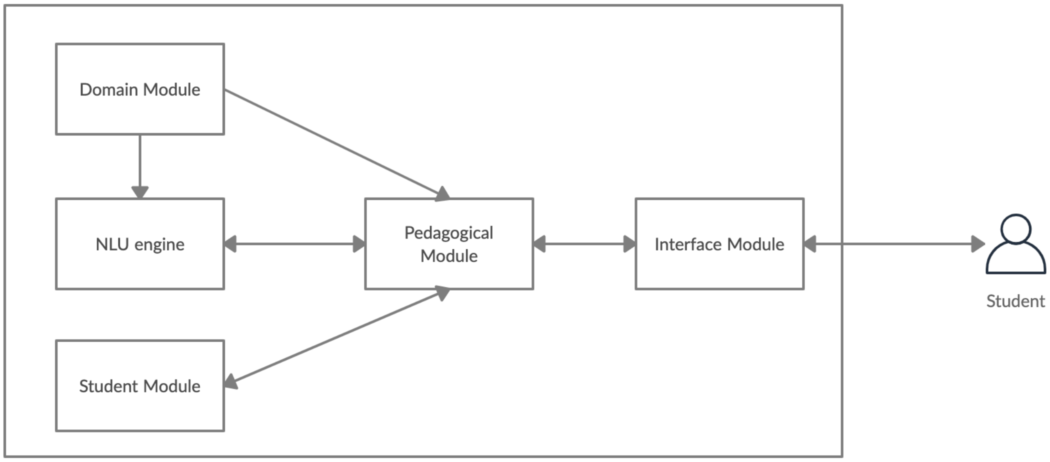


Figure 2. General architecture of Socratic Tutor

Student Module

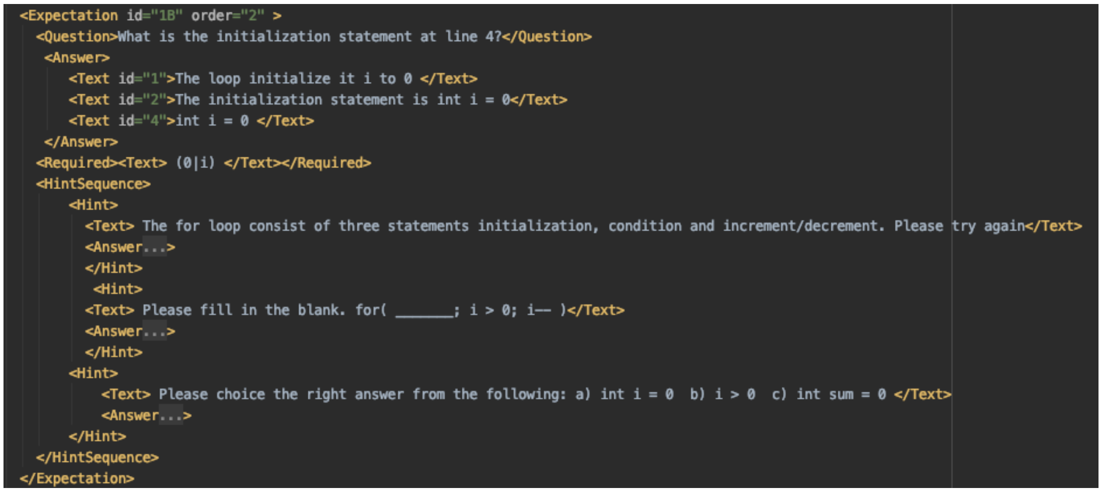
The student module stores the student's information and level of mastery of programming concepts. The module uses an ER database to store and retrieve this information as well as two sets: to-be-learned and completed. When a student logs on for the first time, the module generates a new record and a copy of the to-be-learned set from the domain module. Each time the student completes a task, the concept is moved from the to-be-learned set to the completed set.

Domain Module

The domain module consists of a list of topics in a predefined order, where each programming concept is associated with certain tasks. The tasks are written by experts in XML format, and each task includes Java code, questions, model answers, misconceptions, and feedback. Figure 3 shows a snapshot of a domain task.

Pedagogical Module

The pedagogical module is defined by rules in XML format that inform the tutor about the next step at each point in the dialogue. Therefore, based on a student's answers and the results from the NLU engine, the module decides to (1) start Socratic scaffolding questions, (2) move to the next question, or (3) move to



```
<Expectation id="1B" order="2" >
  <Question>What is the initialization statement at line 4?</Question>
   <Answer>
      <Text id="1">The loop initialize it i to 0 </Text>
      <Text id="2">The initialization statement is int i = 0</Text>
      <Text id="4">int i = 0 </Text>
   </Answer>
  <Required><Text> (0|i) </Text></Required>
  <HintSequence>
      <Hint>
        <Text> The for loop consist of three statements initialization, condition and increment/decrement. Please try again</Text>
        <Answer...>
        </Hint>
         <Hint>
        <Text> Please fill in the blank. for( _______; i > 0; i-- )</Text>
        <Answer...>
        </Hint>
      <Hint>
          <Text> Please choice the right answer from the following: a) int i = 0  b) i > 0  c) int sum = 0 </Text>
          <Answer...>
      </Hint>
  </HintSequence>
</Expectation>
```

Figure 3. Snapshot of a domain task

the next task. The rules can be easily customized for each tutoring session. Figure 4 shows a snapshot of the pedagogical module rules.

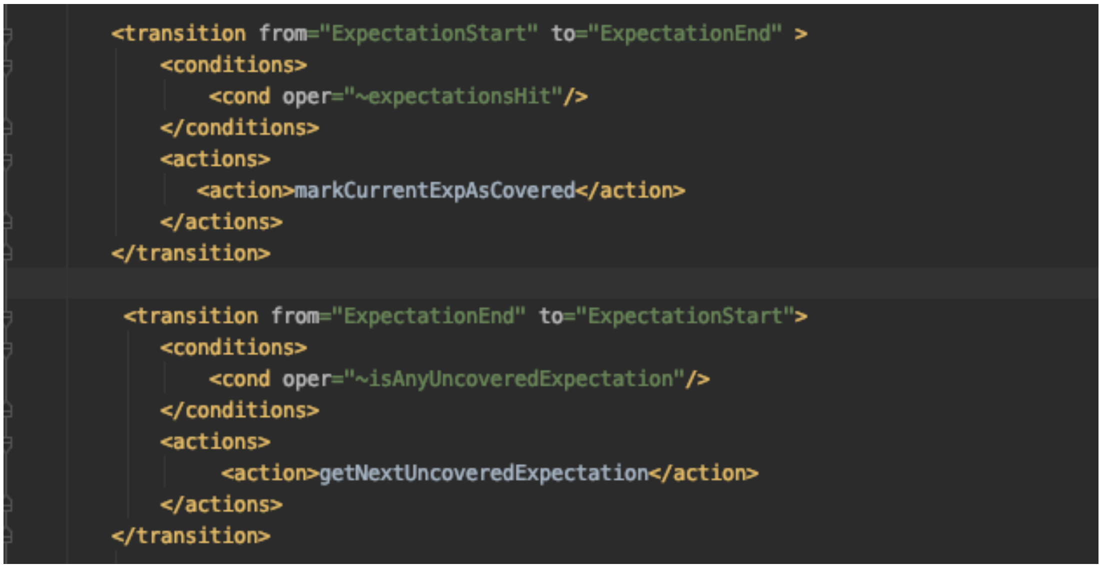
Interface

The interface of Socratic Tutor consists of three main areas, as shown in Figure 5:

1. The code area, where the source code example is shown.
2. The dialogue area, where the dialogue history is shown.
3. The student response area, where students type their input, could show greetings, answers, or questions.

Preliminary Assessment of Socratic Tutor

An experiment was conducted to evaluate the effectiveness of Socratic Tutor and understand its impact on students' programming knowledge as well as the relationship between their learning gains and self-efficacy and the tutor's scaffolding questions. The experiment was conducted to answer the following research questions: (1) How much do students improve their learning of basic programming concepts when using Socratic Tutor? (2) How much can Socratic Tutor help



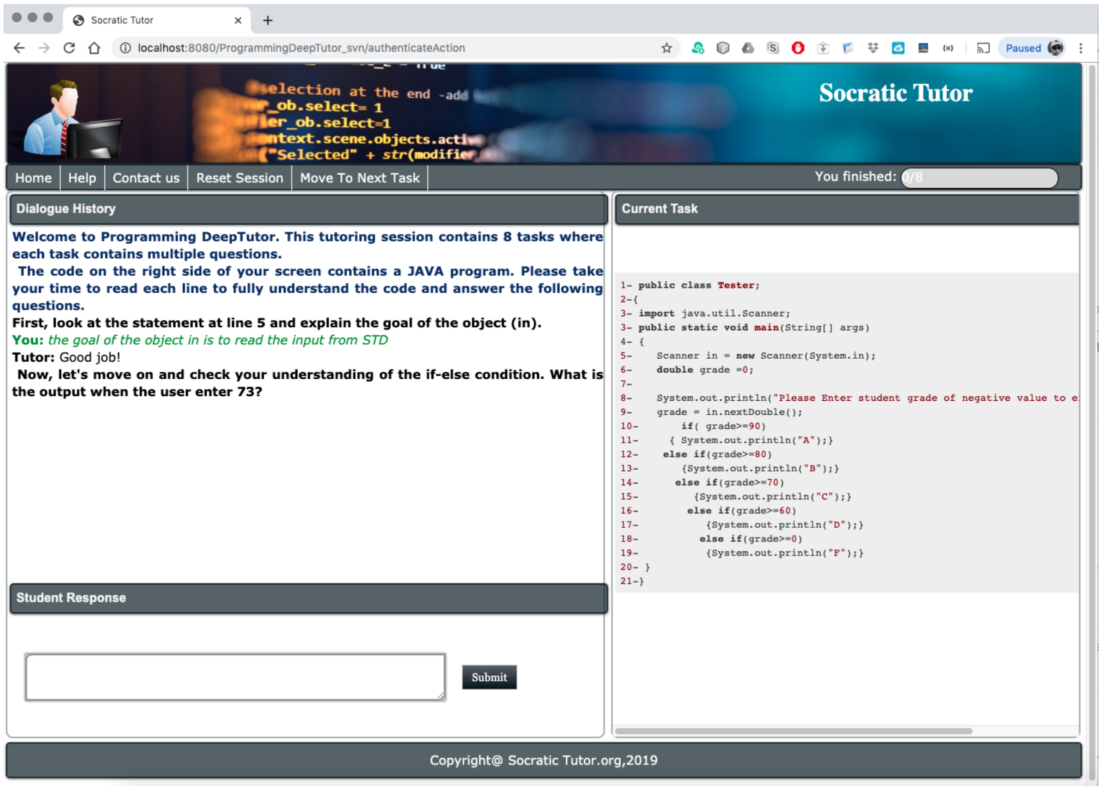
```
<transition from="ExpectationStart" to="ExpectationEnd" >
    <conditions>
        <cond oper="~expectationsHit"/>
    </conditions>
    <actions>
       <action>markCurrentExpAsCovered</action>
    </actions>
</transition>

 <transition from="ExpectationEnd" to="ExpectationStart">
    <conditions>
        <cond oper="~isAnyUncoveredExpectation"/>
    </conditions>
    <actions>
         <action>getNextUncoveredExpectation</action>
    </actions>
</transition>
```

Figure 4. Snapshot of pedagogical rules

students with low prior programming knowledge improve their mastery of basic programming concepts? (3) What is the relationship between Socratic Tutor's scaffolding questions and learning gains? (4) How does the type of scaffolding questions affect students' learning processes? and (5) How does self-efficacy affect students' learning gains? The scope of the experiment was limited to simple input-output operations, using math and string classes, *if* and *if-else* conditions, *while* loop, *for* loop, array, function overload, recursion function, and finally *class*.

Method

Participants consisted of 34 undergraduate students enrolled in an introductory computer science course. The participants interacted with Socratic Tutor where the tutor asked them to explain the code while trying to understand it and to predict the output. After that, it asked students to self-explain the programming statements used in the code. If a participant's answer was incorrect or incomplete, the system initiated the three-level Socratic scaffolding questions.



Welcome to Programming DeepTutor. This tutoring session contains 8 tasks where each task contains multiple questions.

The code on the right side of your screen contains a JAVA program. Please take your time to read each line to fully understand the code and answer the following questions.

First, look at the statement at line 5 and explain the goal of the object (in).

You: *the goal of the object in is to read the input from STD*

Tutor: Good job!

Now, let's move on and check your understanding of the if-else condition. What is the output when the user enter 73?

```
1- public class Tester;
2-{
3- import java.util.Scanner;
3- public static void main(String[] args)
4- {
5-    Scanner in = new Scanner(System.in);
6-    double grade =0;
7-
8-    System.out.println("Please Enter student grade of negative value to e
9-    grade = in.nextDouble();
10-       if( grade>=90)
11-     { System.out.println("A");}
12-    else if(grade>=80)
13-       {System.out.println("B");}
14-      else if(grade>=70)
15-         {System.out.println("C");}
16-        else if(grade>=60)
17-          {System.out.println("D");}
18-          else if(grade>=0)
19-          {System.out.println("F");}
20- }
21-}
```

Figure 5. Socratic Tutor interface

Materials

The materials for this experiment consisted of a background questionnaire, a self-efficacy survey (see Table 9), and a pre/post-test. The background questionnaire contained 16 questions measuring student programming knowledge on a 5-point Likert scale, where 5 was "strongly agree" and 1 was "Strongly disagree." The self-efficacy survey contained 11 questions on a 5-point Likert scale, with each question related to a programming concept that participants would encounter during the tutoring session or general programming skills. The pre-test and post-test had a similar level of difficulty and contained nine Java programs, where participants had to predict the output of each program.

Procedure

The experiment was conducted in a computer lab under the supervision of experimenters. First, participants were briefed about the purpose of the experiment

Table 5: Pre- and post-test means, SD, improvement, and learning gain

		Pre-test		Pos-test				
Section	N	M	SD	M	SD	Potential	Improvement	Learning gain
High	17	83.99	7.58	91.83	7.89	16.01	7.84	48%
Low	17	67.65	8.84	84.97	10.35	32.35	17.32	53%
All	34	75.82	11.6	88.4	9.71	24.18	12.58	52%

* Final scores out of 100%

and asked to read and sign a consent form. Those who consented took a background questionnaire, a self-efficacy survey, and a pre-test. After these initial assessments, they underwent an hour-long tutoring session using Socratic Tutor while working on nine Java code examples. Finally, they took a post-test with a format and difficulty level similar to the pre-test.

Assessment

Each question in the pre- and post-tests were scored with a 1 when the answer provided by the student was correct and 0 otherwise. Based on this rubric and student responses, a learning gain (LG) score was completed as follows (Marx & Cummings, 2007).

$$LG = \begin{cases} \frac{\textit{post-test}-\textit{pre-test}}{100-\textit{pre-test}} & \textit{post-test} > \textit{pre-test} \\ \frac{\textit{post-test}-\textit{pre-test}}{\textit{pre-test}} & \textit{post-test} < \textit{pre-test} \\ \textit{drop} & \textit{pre-test} = \textit{post-test} = 100 \textit{ or } 0 \\ 0 & \textit{post-test} = \textit{pretest} \end{cases} \tag{3.1}$$

Results

Learning Gains

To measure Socratic Tutor's overall effectiveness, the difference between pre- and post-test scores is displayed in Table 5. The difference in this case is a dependent variable that represents the change in learning or knowledge due to the tutoring session. It is calculated by subtracting the number of correct answers in the pre-test from the number of correct answers in the post-test. There are two possible outcomes for the difference: (1) a positive value representing an increase in student knowledge or (2) a zero or negative value representing no improvement or a decline.

Learning gain is another method to represent changes in student knowledge. It is calculated as shown in equation 3.1, where a positive value represents a gain, and zero or a negative value represents no change or a decline.

The results showed an overall pre-test mean of 75.82 and post-test mean of 88.4, for a mean difference of 9.71 points ($p < 0.01$) and a learning gain of 52.03%. Therefore, the overall increase in student knowledge was promising, validating the effectiveness of Socratic Tutor.

We further analyzed the results to answer the research question *How much can Socratic Tutor help students with low prior programming knowledge improve their mastery of basic programming concepts?* For this purpose, participants were divided into two groups, high and low prior programming knowledge, based on average pre-test scores.

The high knowledge group had a mean pre-test score of 83.99 and post-test mean of 91.83, resulting in a post-test vs. pre-test difference of 7.84 points and a learning gain of 48.97%. On the other hand, the low knowledge group had an average pre-test score of 67.65 and average post-test score of 84.97, resulting in a 17.32-point difference between tests and a 53% learning gain. Therefore, students with lower pre-test scores outperformed the high knowledge group by 9.94 points in terms of pre-/post-test differences and 4.57% in terms of learning gains.

The results from a two-tailed test showed a statistically significant difference between the two groups regarding pre- and post-test differences and learning gains ($p < 0.01$).

Table 6: Socratic scaffolding questions per session grouped by average pre-test scores

		Socratic scaffolding questions		
Section	N	Mean	Median	SD
High	17	8.82	6.00	6.96
Low	17	12.88	12.00	7.36
All	34	10.85	10.50	7.34

Socratic Scaffolding Questions

One of Socratic Tutor's main features is its ability to provide tailored support to students and correct misconceptions. It uses positive, neutral, and negative feedback messages and Socratic scaffolding questions that vary in the degree of information provided, as shown in Table 8. On average, the system provides 10 scaffolding questions per tutoring session.

To explore the Socratic scaffolding questions' effectiveness, students were divided into two groups based on their average pre-test scores. Table 6 shows that, on average, students received 10.85 scaffolding questions during a tutoring session. However, this number increased by 15.8% for students in the low programming knowledge group and dropped by 18.8% for those in the high programming knowledge group. These results are cross-referenced with Table 16 to better understand the effects of Socratic scaffolding questions on learning gain scores. The results suggested that more help led to higher knowledge gains, with ($p < 0.01$).

To further understand the relationship between support received and each programming concept, the instances of support for each programming concept were grouped and analyzed, as shown in Table 7. This analysis revealed the degree of difficulty of each programming concept by using help received as an indicator. Students struggled most with the $function$ related concepts, receiving help an average of 2.53 times for $function$ overload and 2.06 times for $recursive function$. On the other hand, the $if-else$ condition, $Math$ and $String$ class, $while$ loop, and $array$ showed the lowest rates of help received. For these concepts, students received support less than once on average, meaning they often got the correct answer without any help. Finally, the *loops* and *class* concepts were in the middle, with feedback given 1.6 times on average.

Table 7: Socratic scaffolding questions per programming concept

Concept	Total	Mean	Median	SD
If-else condition	33.00	0.97	1.00	1.06
Math & String class	17.00	0.50	0.00	0.71
While loop	21.00	0.62	0.00	0.85
For loop	54.00	1.62	1.00	2.24
Array	31.00	0.91	0.00	1.56
Function overload	85.00	2.53	2.00	2.09
Recursion function	70.00	2.06	1.00	2.00
Class	56.00	1.65	1.00	2.01

Analyzing the effectiveness of various scaffolding questions revealed that the most useful question type was fill-in-the-blank questions with a success rate of about 76%. Multiple-choice questions were the second most helpful type with a success rate of about 69%. The least helpful were concept explanation questions with a success rate of about 62%. Furthermore, Socratic Tutor provides three levels of scaffolding, as shown in Table 8. On average, students were able to provide the correct answer 62% of the time after receiving the first level of help. This success rate increased by 14% in Level 2 but fell by 7% in the third level.

Table 8: Effectiveness of Socratic scaffolding questions grouped by type

Level	Type	Succeeded	Failed
1	Concept explanation	62.02%	37.98%
2	Fill-in-the-blank	76.47%	32.53%
3	Multiple-choice	69.05%	30.95%

Self-Efficacy

To understand the relationship between self-efficacy and learning gains, participants were divided into two groups based on average self-efficacy scores, as shown in Table 10. Students with higher self-efficacy scores showed on average

Table 9: Self-efficacy means and SD

Question	Mean	SD
I believe I will receive an excellent grade in the Java programming class.	4.27	1.85
I will certainly master the skills taught in the Java programming class.	3.69	1.76
I can write syntactically correct Java statements.	4.04	1.68
I can understand the language structure of Java and the usage of the reserved words.	4.04	1.87
I can write logically correct blocks of code using Java.	3.85	1.78
I can write a small Java program given a small problem that is familiar to me.	4.50	1.63
I can write Java programs if given a well-specified set of requirements.	4.31	1.67
I can understand how to declare and use variables in Java such as int, double, String, etc.	4.88	1.58
I can understand a conditional expression in Java such as if ... else ...	4.96	1.59
I can declare and use functions in Java.	4.38	1.77
I can declare and use recursive functions in Java.	4.31	1.74
I can understand a loop in Java, such as for and while loops.	4.77	1.58
I can mentally trace well-defined iterative statements in Java, like for and while loops.	4.35	1.77
I can understand the object-oriented paradigm.	4.31	2.05
I can understand and make use of a Java class that is already defined, given a clearly labeled declaration of the class.	4.15	1.93
I can identify the objects in the problem domain and can declare, define, and use them.	4.15	1.89

Table 10: Self-efficacy and leaning gain means and SD

		Self-efficacy		Learning gain	
Section	N	M	SD	M	SD
High	13	80.77	11.72	53.85	44.83
Low	21	42.38	11.08	51.01	23.52

2.84% higher learning gains than those with low self-efficacy. However, a two-tailed test found no statistically significant difference between the two groups ($p > 0.01$).

Conclusion

Promoting a deeper understanding of programming concepts is the primary focus of Socrates Tutor, a dialogue-based ITS that uses NLP techniques to guide dialogue. Socratic Tutor simulates a more realistic student-tutor setting by using natural language, allowing the system to ask any type of question a human tutor might ask and to evaluate natural language answers. Additionally, the system is language independent and can be used to tutor any programming language. Many dialogue-based ITSs have been developed for computer programming, such as DPT, ProPL, GENIUS, and PEA. However, Socrates Tutor is a novel dialogue-based ITS that uses the Socratic method of teaching to tutor students on computer programming.

The system was evaluated by conducting an experiment on 34 urban university students taking an introduction to computer programming course. The results showed that learning gains increased by 52% or 12.58 points from the pre-test to the post-test. Students with lower knowledge at the beginning benefited more from using the system, with a 4.57% increase in learning gains or 9.49 points on the post-test. In addition, students with less knowledge at the beginning received help four more Socratic scaffolding questions on average.

The findings indicated that type of support had a definite effect on novice programmers. Socrates Tutor provides three levels of scaffolding, as shown in Table 8. On average, students were able to provide the correct answer after the first level

62% of the time. This rate increased to 67% in the second level but dropped in the third level by 7%. The most effective type of scaffolding questions were fill-in-the-blank, with a success rate of 76%. Multiple-choice was the second most helpful, with a success rate of 69%. The least helpful was concept explanation, with a success rate of 62%.

Chapter 4

Socratic Method vs. Traditional Output Method for Source Code Comprehension

To understand the impact of the Socratic method on students' programming knowledge and other characteristics, such as confidence, an experiment was conducted to answer the following research questions: (1) How much do students improve their learning of basic programming concepts when using Socratic Tutor? (2) How much can Socratic Tutor help students with low prior programming knowledge improve their mastery of basic programming concepts? (3) How much do students learn about each targeted programming concept when using Socratic Tutor? (4) What is the relationship between the self-explanation volume of Java code and learning gains? (5) How much do students explain the behavior of Java code? (6) What is the relationship between the self-explanation volume of Java code and quality? (7) How much does Socratic Tutor affect students' self-confidence? and (8) What is the relationship between Socratic Tutor's scaffolding questions and learning gains?

This chapter presents a comparative study regarding the traditional output method of learning Java programming and the Socratic method. The participants were university students, and the scope was limited to arithmetic operations, nested $if-else$, $while$ loops, for loops, arrays, and class concepts.

Method

Participants consisted of 70 undergraduate students enrolled in an introductory computer science course at a major 4-year Asian university. Half of the students were randomly assigned to a control group, which used a scaled-down version of the Socratic Tutor ITS that only presented Java code examples and asked participants to predict the output without providing any feedback or Socratic tutoring. The other half of the participants were assigned to a treatment group that used the full version of Socratic Tutor. Socratic Tutor asked them to self-explain the code while trying to understand it and to predict the output. After that, it asked the students to self-explain the programming statements used in the code. If a participant's answer was incorrect or incomplete, the system initiated the three-level Socratic scaffolding.

Materials

Materials for this experiment included a self-confidence survey taken before and after the experiment and a pre-test and post-test measuring participants' knowledge about key programming concepts. The self-confidence survey contained six questions, each related to one programming concept students would encounter during the tutoring session. For each question, participants needed to select a confidence level about the concept on a 7-point Likert scale, where 7 was "highly confident" and 1 was "not confident at all." The pre-test and post-test had similar levels of difficulty and contained six Java programs linked to the six survey questions. For each program in the pre-test and post-test, participants were asked to predict the output of the code.

Procedure

The experiment was conducted in a computer lab under supervision. First, participants were informed about the purpose of the experiment and given a consent form. Those who consented took a self-confidence survey and the pre-test. Once they finished the pre-test, they participated in an hour-long tutoring session. Finally, they took the post-test and another self-confidence survey.

Assessment

The test questions were scored 1 if the answer was correct and 0 otherwise. The learning gain score (LG) was calculated for each participant as follows (Marx & Cummings, 2007):

Table 11: Pre- and post-test means, SD, potential, improvement, and learning gain

		Pre-test		Post-test				
Section	N	Mean	SD	Mean	SD	Potential	Improvement	LG
Control group	29	57.67	35	61.33	33.3	42.3	3.66	8%
Treatment group	32	57.83	30	80	21.67	42.17	22.17	52%

* Final scores out of 100%

$$LG = \begin{cases} \frac{\textit{post-test} - \textit{pre-test}}{100 - \textit{pre-test}} & \textit{post-test} > \textit{pre-test} \\ \frac{\textit{post-test} - \textit{pre-test}}{\textit{pre-test}} & \textit{post-test} < \textit{pre-test} \\ \textit{drop} & \textit{pre-test} = \textit{post-test} = 100 \textit{ or } 0 \\ 0 & \textit{post-test} = \textit{pre-test} \end{cases} \tag{4.1}$$

Their confidence level was computed as indicated below:

$$ConfidenceLevel = post\text{-}Confident - pre\text{-}Confident \tag{4.2}$$

Improvement was calculated as follows:

$$Improvement = Post\text{-}test - Pre\text{-}test \tag{4.3}$$

Results

Out of 70 participants, three were excluded from the treatment group and two from the control group because they had a perfect score on both the pre-test and post-test. Another four were excluded from the control group because they did not complete the experiment, resulting in an adjusted total of 61.

Learning Gains

To answer the questions “How much do students improve their learning of basic programming concepts when using Socratic Tutor?” and “How much can Socratic Tutor help students with low prior programming knowledge improve their mastery of basic programming concepts?”, the results from both groups were

Table 12: Pre- and post-test means, SD, potential, improvement, and learning gains (high vs. low)

		Pre-test		Post-test				
Section	N	Mean	SD	Mean	SD	Potential	Improvement	LG
Control group								
High	14	86.67	6.67	88.3	11.67	13.33	1.63	12%
Low	15	26.76	21.67	38.3	28.33	73.24	11.54	15%
Treatment group								
High	15	71	5	78	6.67	29	7	24%
Low	17	43	21.67	70	21.67	57	27	47%

* Final scores out of 100%

analyzed in terms of average pre-test, post-test, pre-post improvement, and learning gains, as shown in Table 11. The treatment group showed learning gains 45% (18.5 points) higher on the post-test than the control group. A two-tailed t-test revealed a statistically significant difference between groups regarding their improvement *(t=2.8, df=54, p<0.05)* and learning gains *(t=3.6, df=51, p<0.05)*.

Participants were divided into high and low groups based on their average pre-test scores to look for an impact from low or high prior programming knowledge, as shown in Table 12. Those with low prior knowledge of the targeted programming concepts in the control group showed a 11% post-test improvement, although the difference between pre- and post-test was not significant ($p>0.01$). On the other hand, participants with lower prior programming knowledge in the treatment group showed a significant 27% increase in post-test improvement *(t=3.6, df=21, p<0.05)*. Comparing the low knowledge participants from the control and treatment groups, an independent-samples t-test showed a significant difference in post-test improvement *(t=2, df=24, p<0.05)* and learning gains *(t=2.4, df=19, p<0.05)*. Further analysis showed a statistically significant difference between low and high prior programming knowledge participants in the treatment group *(t=2.3, df=21, p<0.05)*.

To address the question “How much do students learn about each targeted

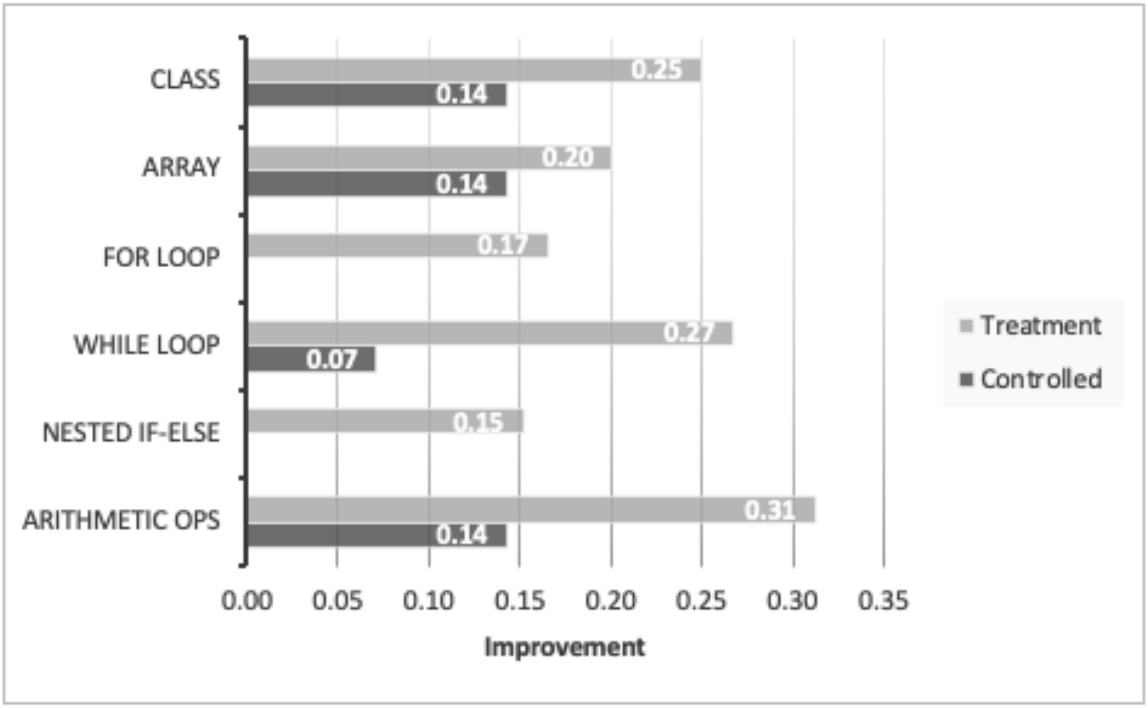


Figure 6. Improvement in students' knowledge per programming concept

Table 13: Means and SD of words, sentences, and content-word per tutoring session

		Word count		Sentence count		Content-word count	
Section	N	Mean	SD	Mean	SD	Mean	SD
All	32	384	46.24	65.2	34.20	168.2	24.36
Grouped by average learning gains							
High	15	410.71	52	69.7	24.18	180.76	28.23
Low	17	353.65	61.3	63.5	32.12	155.24	19.19

programming concept when using Socratic Tutor?", their test improvement was analyzed and is shown in Figure 6. The improvement in the treatment group was higher with a minimum of 10% and maximum of 33%. To further understand these figures, an independent-samples t-test between groups revealed a significant difference in nested if-else *(t=-2.04, df=56, $p<0.5$)* and loop *(t=-1.97, df=54, $p<0.5$)* concepts.

Analysis of Students' Natural Language Responses

According to self-explanation theories of learning, encouraging students to explain target concepts, core programming concepts in this case, should have a

Table 14: Mean of Self-explanation volume and quality factors per task

Section		Volume	Inferences	Control flow	Data flow	Program model
High	13	49.4	2.80	3.80	3.60	4
Low	19	12.93	0.5	0.5	0.33	1

positive impact on their learning. Using the Socratic method, which involves asking students questions and encouraging them to self-explain concepts, was expected to have a similar effect. The researcher quantified students' responses to answer the questions "What is the relationship between the self-explanation volume of Java code and learning gains?" and "How much do students explain the behavior of Java code?" The results of this analysis are shown in Table 13 in terms of word, sentence, and content-word count. The content words were nouns, verbs, adjectives, and adverbs. On average, responses contained 384 words, 65.2 sentences, and 168.2 content words per tutoring session. To further understand how students' knowledge affected their interaction with Socratic Tutor, they were divided into two more groups (high and low) based on their average learning gains. The high programming knowledge group's responses contained more words, sentences, and content words. On average, there was a difference of 57.06 words, 6.2 sentences, and 25.52 content words, but a two-tailed t-test found no significant difference between the high and low groups ($p>0.01$).

To answer the question "What is the relationship between the self-explanation volume of Java code and quality?", self-explanation responses were divided into two groups (high and low) based on average length and were analyzed using inferences, control flow, data flow, and program model in keeping with self-explanation and code comprehension theories (Pennington, 1987a; Schulte et al., 2010). For each factor in each task, self-explanations were measured on a 5-point Likert scale (0 = Very Poor, 1 = Below Average, 2 = Average, 3 = Above Average,

Table 15: pre- and post-confidence scores and average improvement

		Pre-confidence		Post-confidence		Improvement	
Section	N	Mean	SD	Mean	SD	Mean	SD
Control group	29	4.5	2.3	4.4	2	-0.1	2.2
Treatment group	32	4.3	1.6	5.1	1.5	0.8	1.3

4 = Excellent). After that, the final score was measured by averaging the total scores from all tasks, as shown in Table 14.

Descriptive statistics were calculated and an independent-samples t-test was run to compare the mean score for each factor between high and low volume to understand the relationship between the volume and quality of self-explanations. The results showed a significant difference in self-explanation quality between high and low volume responses ($p<0.01$). An independent-samples t-test for each factor showed a significant difference between the low and high groups ($p<0.01$). These results suggested that the volume of explanation had an impact on the quality of self-explanation.

Self-Confidence

To address the question "How much does Socratic Tutor affect students' self-confidence?", students' level of self-confidence was compared before and after the treatment. The confidence level of the treatment group improved by 15% on average, while the control group's confidence went down by -3%, as shown in Table 15. According to an independent-samples t-test, this difference was significant ($t=-3.1$, $df=58$, $p<0.05$).

Socratic Scaffolding Questions

Finally, the last question was answered ("What is the relationship between Socratic Tutor's scaffolding questions and learning gains?"). Students who used Socratic Tutor received an average of 15.4 Socratic scaffolding questions per tutoring session (SD=7.1). The scatter plot in Figure 7 illustrates a positive relationship between the number of scaffolding questions and learning gains. The

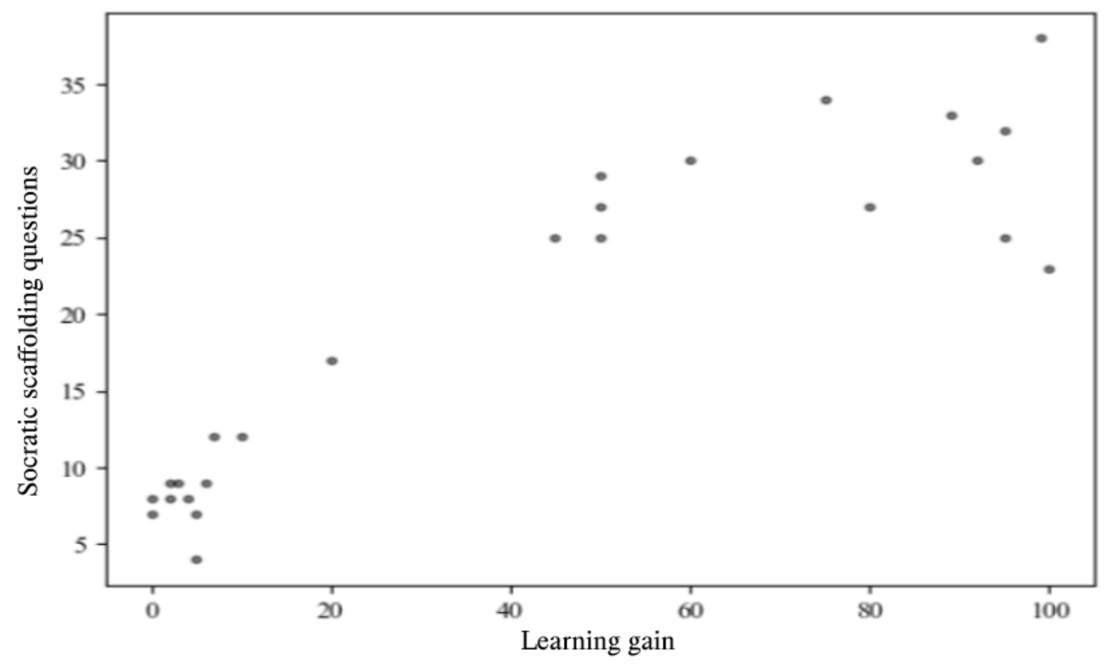


Figure 7. Scatter plot of the relationship between Socratic scaffolding questions and learning gains

relationship was further investigated using a Pearson product-moment correlation coefficient, which showed a strong positive correlation between number of scaffolding questions and learning gains (r=0.68, N=32, p<0.05). In other words, a higher number of Socratic scaffolding questions was associated with higher learning gains.

Conclusion

An experiment was conducted with 70 undergraduate students enrolled in an introductory computer science course to understand the effectiveness of Socratic Tutor. The study sought to answer the following research questions: (1) How much do students learn when using Socratic Tutor? (2) How much does Socratic Tutor affect students with low programming knowledge? (3) How much do students learn about each programming concept? (4) What is the relationship between the quality and volume of self-explanation? (5) How much does Socratic Tutor affect students' self-confidence? and (6) What is the relationship between Socratic scaffolding questions and learning gains?

The participants were divided into a control group, which had to read a code and predict the output without any feedback, and a treatment group, which interacted with Socratic Tutor. The learning gains for the treatment group were 45% higher and their post-test improvement was 24% higher than the control group. Furthermore, students with lower prior programming knowledge in the treatment group benefited significantly more than the control group in terms of their pre-/post-test scores. The analysis for each programming concept showed the improvement in the treatment group was higher, with a minimum of 10% and maximum of 33%. However, only *if-else* and *for* loop showed a statistically significant difference in this regard. Furthermore, Socratic Tutor's effect on students' self-confidence was calculated. The treatment group's confidence level rose by 15%, while it went down by -3% in the control group. Finally, the relation between feedback and learning gains showed a positive correlation *(r=0.68).*

Chapter 5

Socratic Author: Auto-Generated Socratic Dialogue for Programming Comprehension

The potential ability of intelligent tutoring systems (ITSs) to provide individualized, one-on-one instruction for all students has been studied for decades, with the first ITS systems developed as early as 1975 (Danielson & Nievergelt, 1975). ITSs were developed to aid students in the different phases of programming (Bain & Barnes, 2014; Bosse & Gerosa, 2017; Lane & VanLehn, 2003; Truong et al., 2005) and have been shown to improve the quality and effectiveness of programming instruction (Pillay, 2003), as indicated by students' learning gains (A. Corbett et al., 1999; Chassignol et al., 2018; Freedman et al., 2000). However, a major challenge in developing ITSs for computer programming or any target domain are the development or authoring costs. These costs are high because authoring content and other needed elements for ITSs is tedious, error-prone, and time consuming and usually involves domain experts, pedagogical experts, cognitive scientists, linguistic experts in the case of dialogue-based ITSs, and software developers, to name a few (V. Aleven et al., 2009; Blessing, 1997; Heffernan et al., 2006; Murray, 2003).

One way to address this challenge is to automate the entire authoring process, or as many parts of it as possible. To this end, automated authoring systems have been developed in various domains, such as physics, mathematics, and public policy, to increase accessibility and affordability (Heffernan et al., 2006). For instance, (Jordan et al., 2001) built knowledge sources for a dialogue system in only three months. The system used a graphical interface for teachers to construct tutoring dialogue for physics. As another example, (V. Aleven et al., 2009) developed a graphical interface to speed up the development of instructional components, such as hints and other messages.

The work presented in this chapter represents another step toward fully automating the authoring process. More specifically, the chapter offers a method to automate the generation of content for a conversational (dialogue-based) ITS that implements a Socratic method of tutoring to help students in introductory programming courses improve their code comprehension skills.

This automated method relies on static analysis and dynamic simulations of code examples, question generation, and automated assessment of student responses. The method is explained in detail, illustrated with 15 Java code examples and a qualitative analysis of the generated dialogue. The proposed method is fully automated, thus enabling development at scale of ITSs across different Java programming concepts.

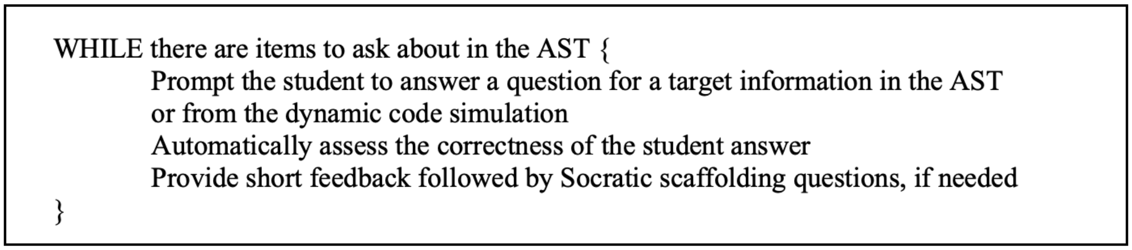
The Socratic Tutoring Framework

Tutoring is one of the most effective forms of instruction. Students in human tutoring conditions show mean learning gains of 0.4–0.9 (non-expert) to 0.8–2.3 standard deviations (expert tutors) compared to students in traditional classroom instruction (Bloom, 1984; Cohen et al., 1982; Graesser et al., 2009; Person et al., 2007; VanLehn et al., 2007). Results have suggested there is something critical about having a one-on-one connection, regardless of whether the student communicates with a human or a computer. (Graesser et al., 1995) argued that the remedial element of tutorial interaction, in which tutor and learner collaboratively improve an initial answer to a problem, is the primary advantage of tutoring over classroom instruction. (Chi et al., 2004) similarly hypothesized that tutoring increases a learner's capacity to reflect iteratively and actively on domain knowledge. Furthermore, one-on-one tutoring has the advantage of engaging most students' attention and interest more than other forms of instruction, such as lecturing, in which the student might choose not to pay attention (VanLehn et al., 2007).

ITSs with conversational dialogue represent a special category of educational technology based on constructivist theories of learning and the collaborative constructive activities that occur during human tutoring. They have been demonstrated to promote learning gains up to an impressive effect of 1.64 sigma, higher than students learning the same content in canned text remediation (VanLehn et al., 2007).

This study has focused on a particular type of conversational ITS that implements a Socratic tutoring style based on prompting students to think and provide information in the form of answers to questions. The questions are designed to follow a directed, predefined line of reasoning (Rosé et al., 2001). Based on static analysis and dynamic simulations of code examples that learners are prompted to understand, this study proposes an automated method to generate a Socratic line of reasoning and corresponding questions necessary to implement a Socratic tutorial dialogue for code comprehension. To be able to automatically assess student responses during interactive dialogue and provide brief feedback, benchmark responses are created from abstract syntax trees. For each target node in the syntax trees, the system generates a question whose correct or benchmark answer is the target information. As an example, the static analysis of the statement *int num = 10;* results in the following benchmark answer *declaring an integer variable num and initializing it to 10* . When a student is prompted to answer the question *What does the statement in Line 1 do?* the student response is automatically compared to the corresponding benchmark response. If the two texts match, positive feedback is provided, e.g., *Great job!* Otherwise, students receive negative feedback followed by Socratic scaffolding questions. The student may also receive neutral feedback, depending on how semantically close their answer is to the benchmark answer. The Socratic tutoring framework as shown in Figure 8 was adopted to automatically generate content for a Socratic ITS.

The Socratic tutoring framework can be implemented adaptively, as not all students will receive all the prompts/questions. For example, those who show



```
WHILE there are items to ask about in the AST {
        Prompt the student to answer a question for a target information in the AST
        or from the dynamic code simulation
        Automatically assess the correctness of the student answer
        Provide short feedback followed by Socratic scaffolding questions, if needed
}
```

Figure 8. The Socratic ITS Framework

mastery of certain concepts, such as loops, will be asked fewer questions about loops than students who have yet to master that concept. This adaptive behavior thus tailors the instruction to each learner, which should result in better learning outcomes.

It should be noted that the above framework covers the inner loop or within-task interaction of an ITS. Thus, it assumes that an instructional task has been selected for the learner to work on, e.g., a particular Java code example. Therefore, the goal is to automate the interaction with the learner within the task. The outer loop, which is responsible for selecting an appropriate instructional task for a given learner, is not described here (see VanLehn's two-loop ITS framework (Vanlehn, 2006)).

In addition, the Socratic line of reasoning for a target code example only targets the program level or program model aspects of comprehension. Comprehension theories make a distinction between the program model, domain model, and situation model (Pennington, 1987b; Schulte et al., 2010). The program model is some representation of the control-flow of the program or a direct mental equivalent of the code. On the other hand, the domain model captures the function or goals of the program from a target domain perspective. Thus, the domain model describes the problem and solution being implemented. Finally, the situation model captures an integrated view of both the program and domain models with an emphasis on cross-references between the two models. Therefore, the situation model contains information not captured by the individual program and domain models. In fact, there is evidence that the best code readers can build such an integrated situation model by seeking to understand and infer cross-references between the program and domain models.

Socratic Author Design

Socratic Author was designed and developed as a stand-alone tool that can be used by ITS developers. The authoring tool requires only source code examples as input to produce a full dialogue that can simply be played by a dialogue manager. The current implementation was developed for the Java language to help students understand Java code examples. The tool was developed in Python and the output dialogue is in JSON format (JavaScript Object Notation, or JSON, is a lightweight data-interchange format that is easy for machines to parse and generate). When porting to a new target language, e.g., Python, the only components that need to be changed are the static code analyzer and the underlying dynamic simulator of the code, which are typically available as off-the-shelf components.

The architecture of the authoring tool consists of five major components: question generation, benchmark answer generation, scaffolding question generation, run time information, and dialogue generation (see Figure 9). The dialogue generation module takes as input the output of the other four components or modules (question generation, answer generation, feedback generation, run time information) to generate a complete segment of dialogue. As already noted, the generated dialogue can be adaptively played by conversational ITSs. Therefore, all learners do not need to be asked all questions; questions can be chosen adaptively depending on learners' knowledge and other characteristics, such as their emotional state.

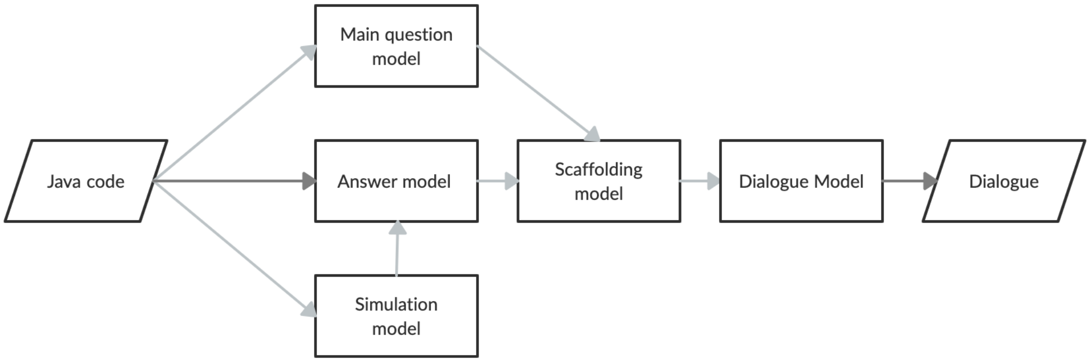
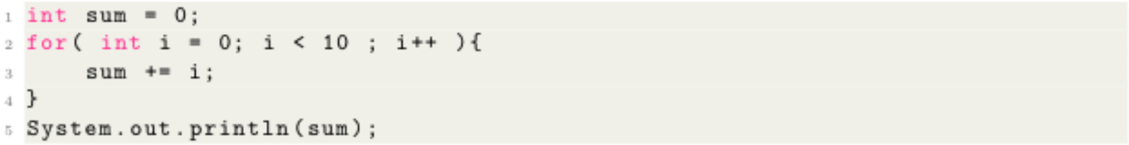


Figure 9. Architecture of Socratic Author

The Question Generation Module

The question generation module uses a Java parser to identify programming concepts and sub-concepts in an AST and program run time information from the dynamic program simulation model. This allows the module to generate questions based on the dynamic behavior of the code, e.g., *how many times a loop is executed and the values of the counter i during the execution*. Furthermore, the AST allows the module to generate questions based on the static code analysis, such as *What is the name of the integer variable declared in Line 1?*. The previous questions were generated from the code shown in Figure 10.

Based on the static analysis of the code, the tool generates three types of scaffolding questions. The first are definition questions to check students' knowledge of basic programming concepts, e.g.,*What is the int keyword used for?*. The second type targets the syntactic understanding of programming concepts, e.g., *Can you indicate the loop's stop condition?*. The last are a sequence of related questions targeting all aspects of programming statements. For instance, for an array of integer declaration and initializing statement, a related question, e.g., *What are the values of the array arr after executing the statement in Line 5?*. Therefore, the sequence question targets the declaration and initialization aspects of the statement. Another example would be *for* loop sequence questions, where each question in the



```
int sum = 0;
for( int i = 0; i < 10 ; i++ ){
    sum += i;
}
System.out.println(sum);
```

Figure 10. Java for loop

sequence targets a critical element: the declaration and initialization of the loop counter, the stop condition, and the increment/decrement statement (see Figure 11).

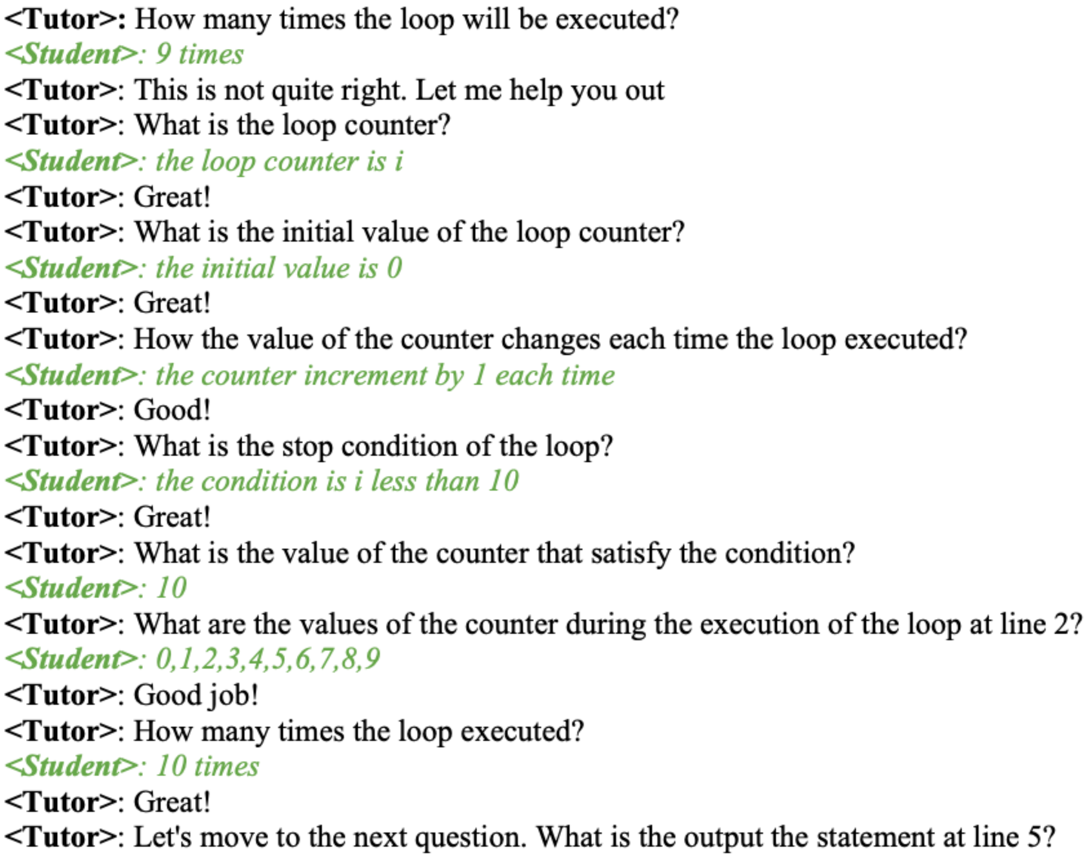
Finally, the module can group a set of questions for a code block, i.e., a group of statements between balanced braces. The purpose is to ask the learner to summarize the goal of the block, e.g., functions, loops, and if-else. For instance, a block question generated for the code in Figure 10 is *What does the code on the block from Line 24 do?*

There is one challenge with these block-level questions. Generating a higher-level benchmark response summarizing the function of the block in order to automatically assess student responses was beyond the scope of the current method, which focused on the program model as opposed to the domain model. Furthermore, automatically generating functional benchmark responses for a given block of code is a challenging task the researcher plans to tackle in the future. For this reason, the current solution to generate the benchmark responses for a given block is to concatenate the benchmark responses of the individual statements in the block, as detailed later.

The Benchmark Answer Generation Module

To generate a meaningful answer, the answer generation model starts with the abstract syntax tree of the input Java program obtained from Java parser.

The model generates block- and statement-level answers by traversing the syntax tree. A complete pass of the tree statement nodes generates a complete sentence, where each type of node is associated with a predefined template. That is, this study followed a template-based text generation approach, which is widely used



<Tutor>: How many times the loop will be executed?

<Student>: 9 times

<Tutor>: This is not quite right. Let me help you out

<Tutor>: What is the loop counter?

<Student>: the loop counter is i

<Tutor>: Great!

<Tutor>: What is the initial value of the loop counter?

<Student>: the initial value is 0

<Tutor>: Great!

<Tutor>: How the value of the counter changes each time the loop executed?

<Student>: the counter increment by 1 each time

<Tutor>: Good!

<Tutor>: What is the stop condition of the loop?

<Student>: the condition is i less than 10

<Tutor>: Great!

<Tutor>: What is the value of the counter that satisfy the condition?

<Student>: 10

<Tutor>: What are the values of the counter during the execution of the loop at line 2?

<Student>: 0,1,2,3,4,5,6,7,8,9

<Tutor>: Good job!

<Tutor>: How many times the loop executed?

<Student>: 10 times

<Tutor>: Great!

<Tutor>: Let's move to the next question. What is the output the statement at line 5?

Figure 11. Snapshot of auto-generated questions for Figure 10

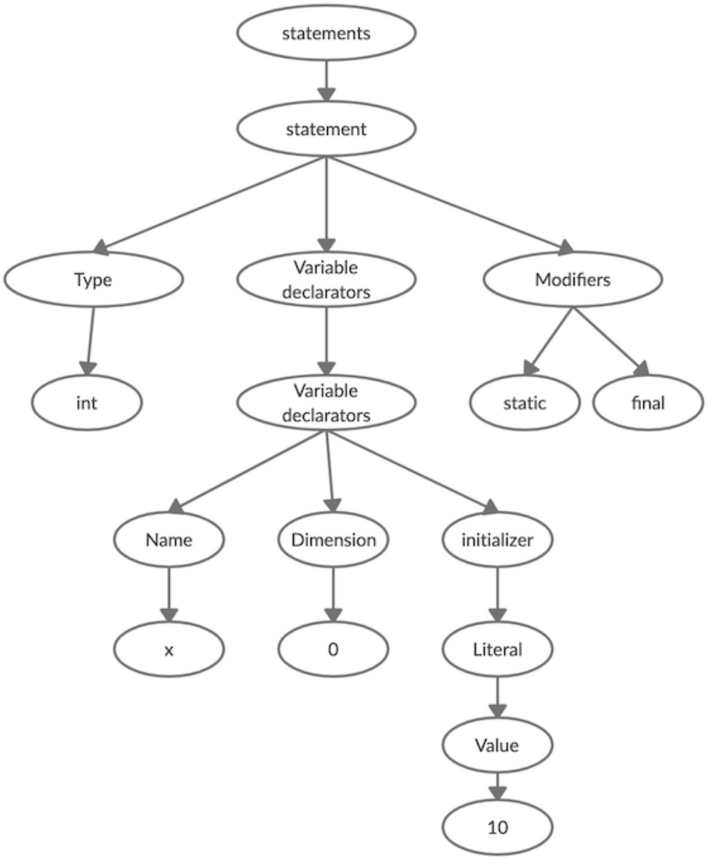


Figure 12. Abstract syntax tree for *static final int x = 10*

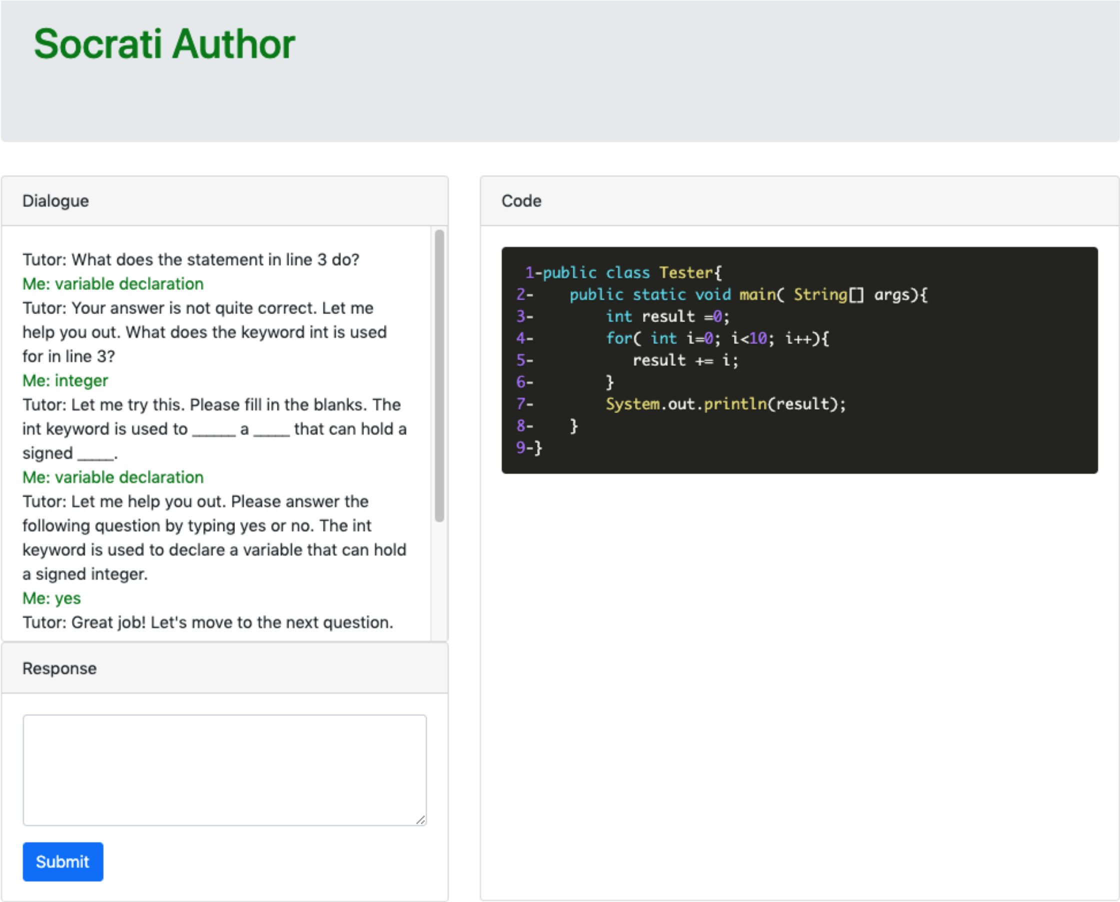
in natural language generation (Jurafsky & Martin, 2008). For instance, a complete pass over the tree of the statement *final static int = 10* produces the following benchmark answer *Declare a final static integer x and initialize it to 10* (see Figure 12. For block-level answers, the answer generation module first identifies the block by matching the node type with a predefined list of types, e.g., *function, loop, and if-else*, then analyzes the sub-tree to generate an answer for each statement in the block. Next, the module combines the answers for each individual statement in the block into a single paragraph. For example, the block-level answer for the *for* loop presented in Figure 10 is “The for loop in line 2 iterates over the counter i from 0 to 9, increasing the counter by 1 in each iteration. In each iteration, the value of sum is incremented to its current value plus the value of i.”

The Code Simulation Module

Generating answers from abstract syntax trees is not enough because the trees do not represent any run-time information. This is critical for code examples that require user input, in which case the behavior of the code will change depending on the user input. Therefore, a Java Debugger Interface (JDI) was used to simulate the execution of Java programs and record variable values. This information answers questions such as *what is the value of the variable sum in Line 3 when i is 4 or what are the values of the counter i during the execution of the for loop*. Therefore, the simulation module offers the ability to trace the execution of Java code and generate questions based on the results of the dynamic execution of the code.

The Socratic Scaffolding Module

The Socratic scaffolding module uses the information from the answer generation module and a set of short predefined phrases expressing positive (*Good job*), neutral (*Good try*), or negative (*Not quite right*) feedback to generate a more complete utterance. The module responses consist of short feedback (e.g., *Good job*) followed by an assertion of the correct answer. For incorrect of incomplete answers, the module generates three levels of Socratic scaffolding questions, as shown in Figure 13. At the first level, a definition question about the targeted programming concept is generated, e.g., *What is the int keyword used for in Line 1?* For Level 2, a concept completion question is generated in the form of a fill-in-the-blank question, e.g., *The int keyword is used to ______ a ______ that can hold a signed* _______. The expected keywords are *declare*, *variable*, and *integer*. Finally, at Level 3, a verification question in the form of a yes/no question is generated, e.g., *Please answer the following question by typing yes or no. Is the int keyword used to declare a variable that can hold a signed integer?*. Using yes/no questions allows the system to verify common misconceptions students may have and correct them. No



Tutor: What does the statement in line 3 do?
Me: variable declaration
Tutor: Your answer is not quite correct. Let me help you out. What does the keyword int is used for in line 3?
Me: integer
Tutor: Let me try this. Please fill in the blanks. The int keyword is used to _____ a ____ that can hold a signed ____.
Me: variable declaration
Tutor: Let me help you out. Please answer the following question by typing yes or no. The int keyword is used to declare a variable that can hold a signed integer.
Me: yes
Tutor: Great job! Let's move to the next question.

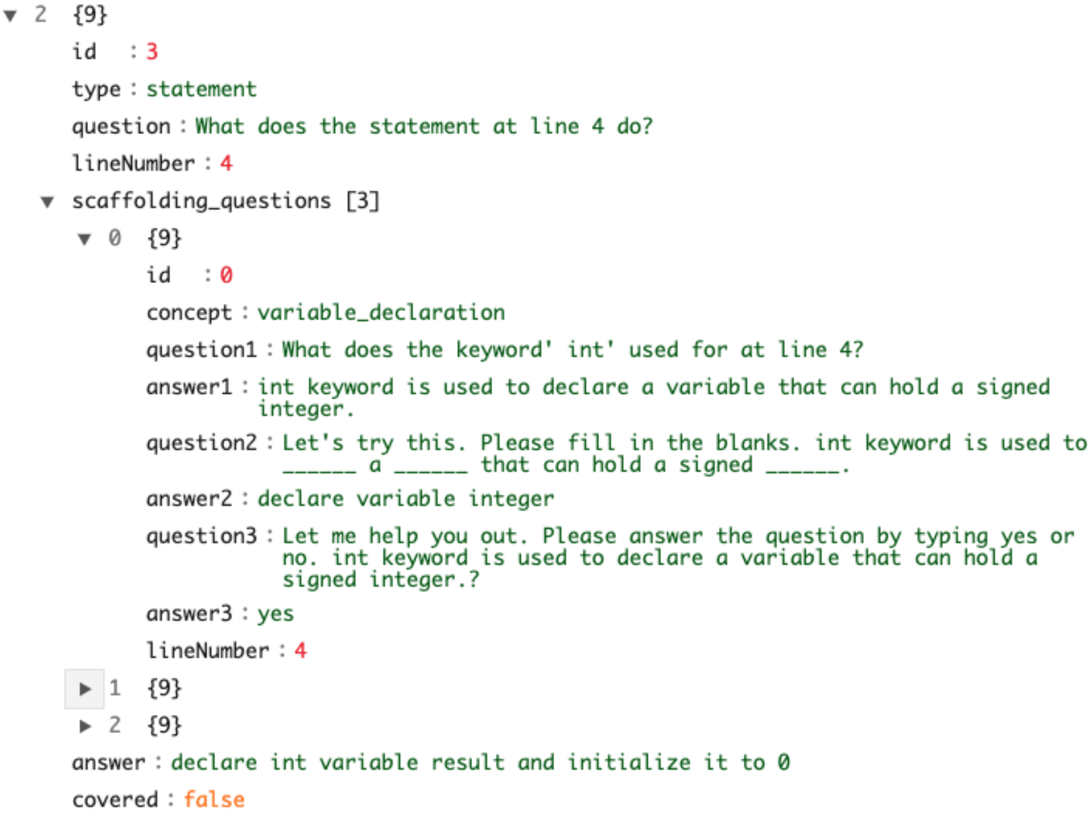
```
 1-public class Tester{
2-    public static void main( String[] args){
3-        int result =0;
4-        for( int i=0; i<10; i++){
5-            result += i;
6-        }
7-        System.out.println(result);
8-    }
9-}
```

Figure 13. Snapshot of tutoring session

misconception information has been incorporated into the method to account for and correct misconceptions, but this feature will be added in the future.

The Dialogue Module

Once the questions, answers, scaffolding questions, and run-time simulations are produced, the dialogue module generates a complete segment of dialogue in the form of a sequence of questions and corresponding expected benchmark answers, which are used to automatically assess whether student responses are correct using a semantic similarity approach (Khayi & Rus, 2019). The dialogue is specified as a JSON object. Figure 14 shows part of the auto-generated Socratic dialogue for the program shown in Figure 10.



```
▼ 2  {9}
     id   : 3
     type : statement
     question : What does the statement at line 4 do?
     lineNumber : 4
   ▼ scaffolding_questions [3]
     ▼ 0  {9}
          id   : 0
          concept : variable_declaration
          question1 : What does the keyword' int' used for at line 4?
          answer1 : int keyword is used to declare a variable that can hold a signed
                    integer.
          question2 : Let's try this. Please fill in the blanks. int keyword is used to
                      ______ a ______ that can hold a signed ______.
          answer2 : declare variable integer
          question3 : Let me help you out. Please answer the question by typing yes or
                      no. int keyword is used to declare a variable that can hold a
                      signed integer.?
          answer3 : yes
          lineNumber : 4
     ▶ 1  {9}
     ▶ 2  {9}
     answer : declare int variable result and initialize it to 0
     covered : false
```

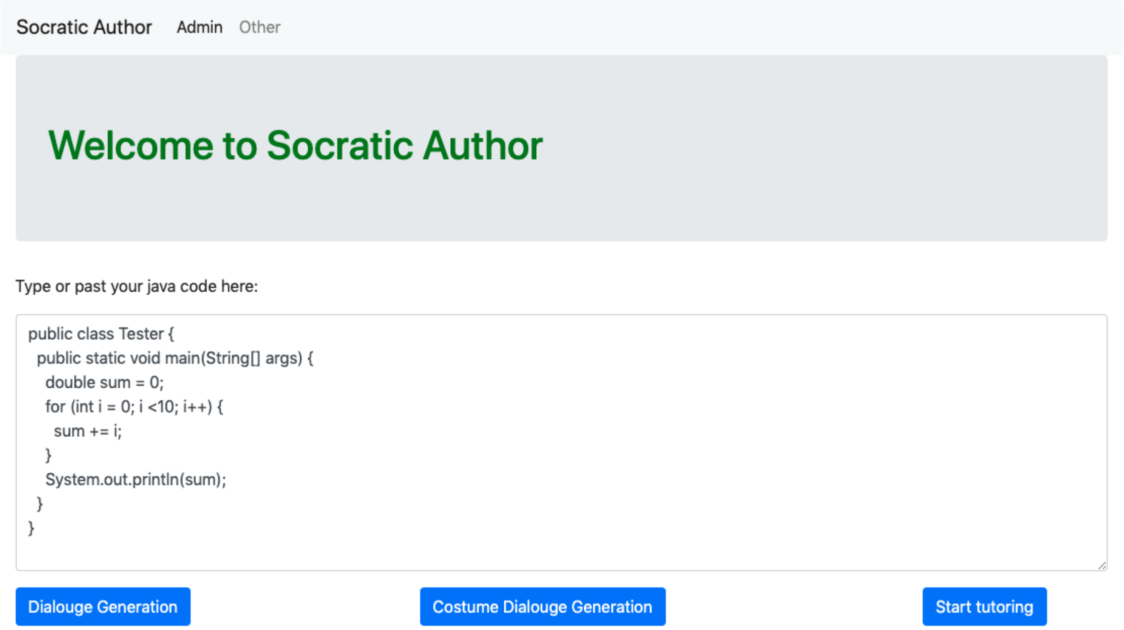
Figure 14. Snapshot of auto-generated dialogue in JSON format

Enacting the Dialogue Model

The dialogue generated for a given code example needs to be played by a dialogue manager as part of a dialogue-based ITS. The role of the dialogue manager is to present the learner with questions in a particular order. Typically, the dialogue manager presents the questions in a sequence corresponding to the lines of code the questions are about. For blocks, the Socratic dialogue starts with questions about each statement within the code block. When all the statements in a block are covered, the ITS asks the students to summarize the block. Other orders of presenting the questions can be envisioned, such as the execution order of the lines of code for a given input.

Besides the questions generated from the abstract syntax trees and the dynamic execution of the code, there are two general types of questions or statements: a question or statement to elicit self-explanation (*Can you read the code and explain in your own words what the code does and what it means to you while you read the code?*) and a prediction question or statement (*Please read the code and predict its output*).

Each generated question is labeled internally according to its type and a list of concepts in order to track students' mastery of key programming concepts. Therefore, this labeling mechanism gives the dialogue manager freedom to chose what questions to present. Various ITS developers may choose different question sequencing strategies. For instance, the following is a potential sequencing. The dialogue manager starts by asking a student to explain the code in detail and predict its output. The Socratic line of questioning is only triggered if the student's explanation and prediction are incorrect or incomplete. For instance, for incomplete explanations, the sequencing strategy may be implemented to ask questions only about the parts of the code that were not explained in sufficient detail.



Welcome to Socratic Author

Type or past your java code here:

```
public class Tester {
 public static void main(String[] args) {
  double sum = 0;
  for (int i = 0; i <10; i++) {
   sum += i;
  }
  System.out.println(sum);
 }
}
```

Figure 15. Socratic authoring tool interface

User Interface

The user interface is simple, easy to use, and consists of a text area to write or paste Java code and three buttons (see Figure 15). The interface offers authors two options: (1) generate a Socratic dialogue and save it as a JSON script or (2) start a tutoring session for testing purposes. Furthermore, the interface provides the ability to choose what concepts the author prefers to generate a dialogue for. For example, if an author wants to encourage students to practice *for* loops, they can select the *for* loop concept only.

The authoring tool also offers an interface for ITSs by using REST API technology. This allows ITSs to easily integrate the tool by requesting a dialogue script for a given Java code and getting as a response the corresponding dialogue as a JSON object.

Conclusion

ITSs can generate impressive learning outcomes in many domains but have seen relatively little use in training and school classrooms due to the time and cost needed to design and author content. This study developed an authoring tool for programming dialogue called Socratic Author to reduce the time and cost associated with manually creating such dialogue. The tool allows teachers to create fully functional Socratic tutoring dialogue for teaching programming from Java code examples.

Socratic Author relies on static analysis and dynamic simulations of code examples, question generation, and automated assessment of student responses. The architecture of the authoring tool consists of five major components: question generation, benchmark answer generation, scaffolding question generation, run time information, and dialogue generation (see Figure 9). The dialogue generation module takes as input the output of the other four modules (question generation, answer generation, feedback generation, run time information) to generate a complete segment of dialogue. The generated dialogue can be adaptively played by conversational ITSs. Therefore, all learners do not need to be asked all questions; questions can be chosen adaptively depending on learners' knowledge and other characteristics, such as their emotional state.

Chapter 6

Socratic Author Evaluations

Two evaluations were conducted To assess the authoring tool. The first had a group of 13 programming instructors evaluate the auto-generated dialogue to explore if this approach would be educationally useful. The second was conducted as a control experiment by having 45 students in an introductory programming course use auto-authored dialogue, expert-authored dialogue, or output only. The second evaluation's goal was to analyze the learning outcomes to see how easy, efficient, and friendly the system was.

Participants

In the first evaluation, the 13 participants were instructors teaching Java programming courses. The second study's participants were 45 undergraduate students enrolled in an introductory programming course at a major 4-year Asian university. These participants were divided into three groups of 15 each. Group 1 was assigned to a tutoring session where experts generated the tutoring dialogue. Group 2 was assigned to a condition in which participants used auto-generated tutoring dialogue. Group 3 was assigned to a scaled-down version of the system, which presented Java code examples and asked about the output without providing any scaffolding questions or feedback.

Materials

Materials for the first evaluation study consisted of 15 auto-generated dialogues from Java code examples that featured variables, if and if-else conditionals, for loop, while loop, and array. The dialogues were formatted in a human readable form and then shown one by one to each rater. At the end of each dialogue, raters filled out a survey of 12 questions on a 5-point Likert scale from 1 (strongly disagree) to 5 (strongly agree) (see Table 20).

Materials for the controlled experiment included a pre-test and post-test measuring participants' knowledge of key programming concepts and a seven-question survey to evaluate the authored dialogue. The pre- and post-tests were similar in terms of difficulty and contained six Java programs in which each question assessed students' understanding of a particular concept. For each question in the pre- and post-tests, participants were asked to predict the code's output.

Procedure

For the first evaluation, a survey link was emailed to 15 individuals teaching introductory programming courses. The instructors were given five days to complete the survey.

On the other hand, the controlled experiment was conducted in a computer lab under supervision. First, participants were debriefed about the purpose of the experiment and were given a consent form. Those who consented started by taking a pre-test. Immediately after the pre-test, they participated in an approximately 60-minute tutoring session. Finally, they took the post-test and an evaluation survey. Group 3 took no survey at the end of the experiment since they did not interact with the ITS dialogue.

Assessment

The pre- and post-test questions were scored 1 when the student answer was correct and 0 when incorrect. The learning gain score (LG) was calculated for each participant as follows (Marx & Cummings, 2007).

$$LG = \begin{cases} \frac{\textit{post-test} - \textit{pre-test}}{100 - \textit{pre-test}} & \textit{post-test} > \textit{pre-test} \\ \frac{\textit{post-test} - \textit{pre-test}}{\textit{pre-test}} & \textit{post-test} < \textit{pre-test} \\ \textit{drop} & \textit{pre-test} = \textit{post-test} = 100 \textit{ or } 0 \\ 0 & \textit{post-test} = pre - test \end{cases} \tag{6.1}$$

Table 16: Pre- and post-test means, SD, potential, improvement, and learning gains

Section	N	Pre-test		Post-test		Potential	Improvement	Learning gain
		Mean	SD	Mean	SD			
Group 1	14	58	21	79.3	13	42	21.3	51%
Group 2	15	56	16	74	11	44	18	43%
Group 3	15	57	35	62	33	43	5	12%

*Final scores out of 100%

Table 17: Mean and SD of turns, words, sentences, and content words

Section	Turns		Words		Sentences		Content words	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
Group 1	123	13	402	217	45	8	191	164
Group 2	115	28	334	183	35	18	154	106

Results

To understand each tutoring method's overall effect on students' knowledge, knowledge change is reported by group. Table 16 shows the average scores of the pre-test for Groups 1 (58), 2 (56), and 3 (57), with a range of 2%. Thus, the pre-test indicated the groups had similar knowledge at the outset. However, the post-test scores improved by 21% in Group 1, 18% in Group 2, and 3% in Group 3. These differences between pre- and post-test scores showed vastly different learning gains of 51% (Group 1), 43% (Group 2), and 12% (Group 3). This suggested the auto-authored dialogue improved students' knowledge by 43% and outperformed the output-only group.

Despite the difference in learning gains between Groups 1 and 2, a two-tailed t-test showed no significant difference between them $(t=0.83,\ df=19,\ p>0.05)$. However, the t-test did show a significant difference in learning gains between Group 3 and the other two groups $(t=3.1,\ df=52,\ p<0.05)$.

To evaluate dialogue efficiency, student responses were analyzed from dialogue logs regarding the number of turns, words, sentences, and content words produced by each group (see Table 17). On average, each tutoring session consisted

Table 18: Mean and SD of Socratic scaffolding questions and their success rate

Section	Scaffolding questions		Level 1		Level 2		Level 3	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
Group 1	28	4	55%	16	70%	23	61%	37
Group 2	31	7.5	41%	21	47%	13	73%	24

of 119 turns. Group 1 produced 402 words, 45 sentences, and 191 content words within 123 turns, while Group 2 produced 334 words, 35 sentences, and 154 content words within 115 turns. Therefore, students who worked with expert-written dialogue interacted more with the tutor and produced more words, sentences, and content words. However, a two-tailed t-test showed a significant difference $(t=6.13, df=19, p<0.05)$ only in terms of sentences.

The dialogue logs were further analyzed to understand the difference between expert- and auto-generated Socratic scaffolding questions. Table 18 shows the average amount of questions received by students and the success rate. A question was considered successful if the student provided the correct answer. Table 18 shows that Group 1 received more scaffolding questions and had a higher success rate in the first and second level questions, while the third level of auto-generated feedback had a 12% higher success rate. However, a two-tailed t-test showed a significant difference only in the first $(t=3.6, df=25, p>0.05)$ and second $(t=-4.1, df=25, p<0.05)$ instances of feedback.

To understand how efficient and user-friendly the system was, the results from after the session survey were analyzed, as shown in Table 19. The survey contained seven items on a 5-point Likert scale from 5 (strongly agree) to 1 (strongly disagree). Group 1 students agreed or strongly agreed with all questions except Questions 5 and 6. On the other hand, Group 2 students agreed or strongly agreed with all questions except Questions 5, 6, and 7. Fleiss' kappa scores for inter-rater reliability for Group 1 (0.52) and Group 2 (0.43) suggested the agreement between participants was higher in Group 1 than in Group 2.

Table 19: Mean and SD of student survey questions about tutoring dialogue

#	Question	Group 1		Group 2	
		Mean	SD	Mean	SD
1	The questions were clear and easy to understand	4.3	0.9	4	0.8
2	The system was able to understand my answers and respond accordingly	4.6	1.2	4.3	0.9
3	The scaffolding questions from the system helped me come up with the correct answer	4.4	0.5	4.2	0.4
4	The system was effective at helping me understand the code examples	4.6	0.5	4.4	0.7
5	The system was effective at helping me understand core programming concepts	3.6	1.1	3.7	1.2
6	The system helped me understand Java programming	3.6	0.8	3.4	1.2
7	The system provided a useful learning experience	4.5	0.5	3.4	1.3

A survey of 13 introductory programming course instructors was conducted to evaluate the quality of auto-generated dialogue from an educational standpoint. Table 20 shows the results in terms of average and standard deviation for eight questions on a 5-point Likert scale from 5 (strongly agree) to 1 (strongly disagree). Questions 1 and 2 targeted the syntactic and semantic quality of the dialogue, Question 3 asked about coherence and consistency, and the rest focused on educational goals and whether raters thought they might use the system in their teaching.

The instructors agreed or strongly agreed with all questions except Question 5. Thus, they thought the auto-generated dialogue would help students understand Java programs better and learn programming concepts *(Fleiss' Kappa = 0.51)*. However, they neither agreed nor disagreed that the dialogue would help students develop a robust understanding of programming concepts. At the end of the survey, raters were allowed to provide voluntary feedback, which was positive. For instance, one of the raters stated that "the system looks promising. The dialogue looks coherent to the point you feel it is not auto-generated."

Table 20: Mean and SD of instructor survey questions on auto-generated tutoring dialogue

#	Question	Mean	SD
1	I think the generated questions, scaffolding questions, and answers are syntactically correct.	4.2	0.55
2	I think the generated questions, scaffolding questions, and answers are semantically correct.	4.5	0.84
3	I think the generated dialogue is coherent and consistent.	4.4	0.45
4	I think the generated dialogue would help students understand Java code.	4.2	0.84
5	I think the generated dialogue would help students develop a robust understanding of programming concepts.	3.8	0.55
6	I think the generated Socratic scaffolding questions would help students learn and understand the corresponding Java code.	4.6	0.55
7	I think the generated dialogue covers all important programming concepts presented in the code.	4.8	0.45
8	I think I may use this system in the classroom.	4.2	0.84

Conclusion

To assess the authoring tool, two evaluations were carried out. The first had a group of 13 programming instructors evaluate the auto-generated dialogue and explore whether this approach was beneficial. The second was conducted as a controlled experiment in which 45 introductory programming students used expert-authored dialogue (Group 1), auto-authored dialogue (Group 2), or output only (Group 3). The second evaluation's goal was to analyze the learning outcomes to see how easy, efficient, and friendly the system was.

The controlled experiment results suggested that auto-generated Socratic dialogue for programming comprehension could improve students' knowledge. For example, the average learning gain of Group 2 students was 43%, compared to 12% among Group 3 students. Group 1 students outperformed both Group 2 and Group 3, as anticipated, although this difference was not statistically significant.

Analyzing tutoring logs showed that Group 1 students produced more words, sentences, and content words and showed greater interaction with the system by taking more turns. Furthermore, this group received more scaffolding questions and had higher rates of success providing the correct answer to the first and second level questions.

According to the post-tutoring survey, students in Group 1 agreed more strongly with all questions except Questions 5 and 6, although Group 2 also generally agreed or strongly agreed with the questions. However, the rating dropped to neither agree nor disagree for Questions 5, 6, and 7. The overall results suggested students preferred interacting with expert-written dialogue.

Programming instructors agreed or strongly agreed with all items in a survey about the strengths of the tool. The only exception was Question 5, as instructors neither agreed nor disagreed that the auto-generated dialogue would help students develop a robust understanding of programming concepts. Based on these surveys, expert-written tutoring dialogue outperformed auto-generated dialogue in many areas. However, auto-generated dialogue can be created from Java code examples in less than a minute and requires no technical or educational knowledge. Therefore, considering the cost, skills, and time needed to generate expertly written dialogue, the tool offers a promising opportunity for students and teachers alike.

Chapter 7

Conclusion and Future Work

This study explored an application of natural language technology in an effort to help novice programmers develop a deeper understanding of programming concepts. This work demonstrates how natural language technology and guided self-explanation through the Socratic teaching method can improve understanding of programming concepts. The study has produced a Socratic dialogue-based intelligent tutoring system (ITS) called “Socratic Tutor” and an automated dialogue authoring tool called “Socratic Author” that generates Socratic dialogue from Java source code.

The study aimed to determine how effective the Socratic method would be at eliciting learners to engage in self-explanation when using Socratic Tutor and the quality of Socratic Author’s output. The research questions asked (1) can a Socratic ITS lead to improved code comprehension? and (2) to what extent can Socratic dialogue be developed automatically?

This study used a novel approach by leveraging a Socratic ITS for programming students. Therefore, it helps establish a relationship between code comprehension and the use of the Socratic method in computer science as well as between learning gains and student responses in open-ended dialogue. Furthermore, the work introduces a novel approach for generating Socratic dialogue from Java source code. This approach could help teachers and ITS developers automatically create dialogue by entering Java code without requiring non–domain knowledge.

Socratic Tutor is a dialogue-based ITS that uses NLP techniques to promote a deeper understanding of programming concepts. The system simulates a more realistic student-tutor setting by using natural language, allowing the system to ask any type of question a human tutor might ask and evaluate any natural language answer. Moreover, the system is language-independent and can be used to help with any programming language. Many dialogue-based ITSs have been developed for computer programming; unlike past efforts, however, Socratic Tutor is a novel dialogue-based ITS that uses the Socratic teaching method to tutor students on computer programming.

Socratic Tutor was evaluated through an experiment involving 34 urban university students taking an introduction to computer programming course. Learning gains increased by 52% or 12.58 points from the pre-test to the post-test, and students with lower knowledge at the beginning benefited more from the system, with a 4.57% increase in learning gains on the post-test.

The findings indicated that the type of support given had a definite effect on novice programmers. Socratic Tutor provides three levels of scaffolding, as shown in Table 8. On average, students were able to provide the correct answer after receiving the Level-1 question 62% of the time. This rate increased to 76% at Level 2; however, it dropped by 7% at Level 3. Therefore, the most effective scaffolding questions were fill-in-the-blank, with a success rate of 76%, while multiple-choice was the second most helpful, with a success rate of 69%. The least helpful was concept explanation, with a success rate of 62%.

Another experiment was conducted with 70 undergraduate students enrolled in an introductory computer science course to further explore Socratic Tutor's effectiveness. The participants were divided into two groups: a control group that read a code and predicted the output without any Socratic scaffolding questions or feedback and a treatment group that interacted with Socratic Tutor. The learning gains for the treatment group were 45% higher than the control group. Furthermore, students with lower prior programming knowledge in the treatment group benefited significantly more in terms of pre-/post-test scores. The treatment group's improvement was higher in each programming concept, with a minimum of 10% and a maximum of 33%. However, only *if-else* and *for* loop showed a statistically significant difference in this regard. Furthermore, Socratic Tutor's effect on students' self-confidence was calculated. The treatment group's confidence level rose by 13%, while it went down by -1.6% in the control group. Finally, the relation between scaffolding questions and learning gains showed a positive correlation *(r=0.68)*.

Socratic Author relies on static analysis and dynamic simulations of code examples, question generation, and automated assessment of student responses. The architecture of the authoring tool consists of five major components: question generation, benchmark answer generation, scaffolding question generation, run time information, and dialogue generation (see Figure 9). The dialogue generation module uses the output of the other four modules to generate a complete segment of dialogue. The generated dialogue can be adaptively played by conversational ITSs. Therefore, all learners do not need to be asked all questions; questions can be chosen adaptively depending on learners' knowledge and other characteristics, such as their emotional state.

To assess the authoring tool, two evaluations were carried out. The first had a group of 13 programming instructors evaluate the auto-generated dialogue and explore whether this approach was beneficial. The second was conducted as a controlled experiment in which 45 introductory programming students used expert-authored dialogue (Group 1), auto-authored dialogue (Group 2), or output only (Group 3). The second evaluation's goal was to analyze the learning outcomes to see how easy, efficient, and friendly the system was.

The controlled experiment results suggested that auto-generated Socratic dialogue for programming comprehension could improve students' knowledge. For example, the average learning gain of Group 2 students was 43%, compared to 12% among Group 3 students. Group 1 students outperformed both Group 2 and Group 3, as anticipated, although this difference was not statistically significant.

Analyzing tutoring logs showed that Group 1 students produced more words, sentences, and content words and showed greater interaction with the system by taking more turns. Furthermore, this group received more scaffolding questions and had higher rates of success providing the correct answer to the first and second level questions.

According to the post-tutoring survey, students in Group 1 agreed more strongly with all questions except Questions 5 and 6, although Group 2 also generally agreed or strongly agreed with the questions. However, the rating dropped to neither agree nor disagree for Questions 5, 6, and 7. The overall results suggested students preferred interacting with expert-written dialogue.

Programming instructors agreed or strongly agreed with all items in a survey about the strengths of the tool. The only exception was Question 5, as instructors neither agreed nor disagreed that the auto-generated dialogue would help students develop a robust understanding of programming concepts. Based on these surveys, expert-written tutoring dialogue outperformed auto-generated dialogue in many areas. However, auto-generated dialogue can be created from Java code examples in less than a minute and requires no technical or educational knowledge. Therefore, considering the cost, skills, and time needed to generate expertly written dialogue, the tool offers a promising opportunity for students and teachers alike.

Future Work

Future work could involve improving Socratic Author to generate Socratic dialogue for other programming languages, such as Python or C++. Furthermore, Socratic Author has many potential applications to help novice programmers, such as code explanation and question answering. For example, students could paste a code and ask the tool to explain it and answer questions about the code. Finally, the researcher plan to improve the answer generation module to produce the function or goals of code examples.

References

Aleven, V., McLaren, B. M., Sewall, J., & Koedinger, K. R. (2009). A new paradigm for intelligent tutoring systems: Example-tracing tutors. *International Journal of Artificial Intelligence in Education*, *19*(2), 105–154.

Aleven, V. A., & Koedinger, K. R. (2002). An effective metacognitive strategy: Learning by doing and explaining with a computer-based cognitive tutor. *Cognitive science*, *26*(2), 147–179.

Alhassan, R. (2017). The Effect of Employing Self-Explanation Strategy with Worked Examples on Acquiring Computer Programing Skills. *Journal of Education and Practice*, *8*(6), 186–196.

Alqadi, B. S., & Maletic, J. I. (2017). An empirical study of debugging patterns among novices programmers. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education* (pp. 15–20).

Altamura, O., & Roselli, T. (1995). HYPEREX: an intelligent tutoring hypertext system for learning programming. In *Proceedings of the International Conference on Multimedia Computing and Systems* (pp. 341–344).

Anderson, J. R., & Skwarecki, E. (1986). The automated tutoring of introductory computer programming. *Communications of the ACM*, *29*(9), 842–849.

Bain, G., & Barnes, I. (2014). Why is programming so hard to learn? In *Proceedings of the 2014 conference on Innovation & technology in computer science education* (pp. 356–356).

Banjade, R., Niraula, N. B., Maharjan, N., Rus, V., Stefanescu, D., Lintean, M., & Gautam, D. (2015). Nerosim: A system for measuring and interpreting semantic textual similarity. In *Proceedings of the 9th international workshop on semantic evaluation (SemEval 2015)* (pp. 164–171).

Barr, A., Beard, M., Number, C. A., Farr, M., & Young, J. (1976). The computer as a tutorial laboratory: The stanford bip project.

Basili, V., & Boehm, B. (2001). COTS-Based Systems Top 10 List. *IEEE Annals of the History of Computing*, *34*(05), 91–93.

Beaubouef, T., & Mason, J. (2005). Why the high attrition rate for computer science students: some thoughts and observations. *ACM SIGCSE Bulletin*, *37*(2), 103–106.

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *AcM SIGcSE Bulletin*, *39*(2), 32–36.

Bennedsen, J., & Caspersen, M. E. (2019). Failure rates in introductory programming: 12 years later. *ACM inroads*, *10*(2), 30–36.

Bielaczyc, K., Pirolli, P. L., & Brown, A. L. (1995). Training in self-explanation and self-regulation strategies: Investigating the effects of knowledge acquisition activities on problem solving. *Cognition and instruction*, *13*(2), 221–252.

Blank, G., Parvez, S., Wei, F., & Moritz, S. (2005). A web-based its for oo design. In *Proceedings of Workshop on Adaptive Systems for Web-based Education at 12th International Conference on Artificial Intelligence in Education (AIED'2005). Amsterdam, the Netherland* (pp. 59–64).

Blessing, S. B. (1997). A programming by demonstration authoring tool for model-tracing tutors. *International Journal of Artificial Intelligence in Education (IJAIED)*, *8*, 233–261.

Bloom, B. S. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational researcher*, *13*(6), 4–16.

Boghossian, P. (2012). Socratic pedagogy: perplexity, humiliation, shame and a broken egg. *Educational Philosophy and Theory*, *44*(7), 710–720. (disadvantages of Socratic Method)

Bonar, J. G., & Cunningham, R. (1988). *Bridge: Intelligent tutoring with intermediate representations* (Tech. Rep.). Carnegie Mellon University, University of Pittsburgh.

Bosse, Y., & Gerosa, M. A. (2017). Why is programming so difficult to learn? Patterns of Difficulties Related to Programming Learning Mid-Stage. *ACM SIGSOFT Software Engineering Notes*, *41*(6), 1–6.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, *18*(6), 543–554.

Brown, A. L., & Kane, M. J. (1988). Preschool children can learn to transfer: Learning to learn and learning from example. *Cognitive Psychology*, *20*(4), 493–523.

Brusilovsky, P. (1992). Intelligent tutor, environment and manual for introductory programming. *Educational and Training Technology International*, *29*(1), 26–34.

Bryce, R. C., Cooley, A., Hansen, A., & Hayrapetyan, N. (2010). A one year empirical study of student programming bugs. In *2010 IEEE Frontiers in Education Conference (FIE)* (pp. F1G–1).

Burkhardt, J.-M., Détienne, F., & Wiedenbeck, S. (2002). Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, *7*(2), 115–156.

Buse, R. P., & Weimer, W. R. (2008). A metric for software readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (p. 121–130). New York, NY, USA: Association for Computing Machinery.

Butz, C. J., Hua, S., & Maguire, R. B. (2006). A web-based bayesian intelligent tutoring system for computer programming. *Web Intelligence and Agent Systems: An International Journal*, *4*(1), 77–97.

Cabada, R. Z., Estrada, M. L. B., Hernández, F. G., & Bustillos, R. O. (2015). An affective learning environment for Java. In *Advanced learning technologies (icalt), 2015 ieee 15th international conference on* (pp. 350–354).

Cazzola, W., & Olivares, D. M. (2016). Gradually learning programming supported by a growable programming language. *IEEE Transactions on Emerging Topics in Computing*, *4*(3), 404–415.

Chang, C.-K. (2014). Effects of using Alice and Scratch in an introductory programming course for corrective instruction. *Journal of Educational Computing Research*, *51*(2), 185–204.

Chang, K.-E., Lin, M.-L., & Chen, S.-W. (1998). Application of the Socratic dialogue on corrective learning of subtraction. *Computers & Education*, *31*(1), 55–68. (shows positive effect of Socratic method on students knowledge (Math))

Chang, K.-E., Sung, Y.-T., Wang, K.-Y., & Dai, C.-Y. (2003). Web/spl I. bar/Soc: a Socratic-dialectic-based collaborative tutoring system on the World Wide Web. *IEEE Transactions on Education*, *46*(1), 69–78.

Chassignol, M., Khoroshavin, A., Klimova, A., & Bilyatdinova, A. (2018). Artificial Intelligence trends in education: a narrative overview. *Procedia Computer Science*, *136*, 16–24.

Chi, M. T., De Leeuw, N., Chiu, M.-H., & LaVancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive science*, *18*(3), 439–477.

Chi, M. T., Siler, S. A., & Jeong, H. (2004). Can tutors monitor students' understanding accurately? *Cognition and instruction*, *22*(3), 363–387.

Chrysafiadi, K., & Virvou, M. (2013). Student modeling approaches: A literature review for the last decade. *Expert Systems with Applications*, *40*(11), 4715–4729.

Cohen, P. A., Kulik, J. A., & Kulik, C.-L. C. (1982). Educational outcomes of tutoring: A meta-analysis of findings. *American educational research journal*, *19*(2), 237–248.

Conati, C., Gertner, A., & Vanlehn, K. (2002). Using bayesian networks to manage uncertainty in student modeling. *User modeling and user-adapted interaction*, *12*(4), 371–417.

Conati, C., & Vanlehn, K. (2000). Toward computer-based support of meta-cognitive skills: A computational framework to coach self-explanation. *International Journal of Artificial Intelligence in Education (IJAIED)*, *11*, 389–415.

Corbett, A., Anderson, J., Graesser, A., Koedinger, K., & VanLehn, K. (1999). Third generation computer tutors: learn from or ignore human tutors? In *CHI'99 Extended Abstracts on Human Factors in Computing Systems* (pp. 85–86).

Corbett, A. T., Koedinger, K. R., & Hadley, W. (2001). Cognitive Tutors: From the research classroom to all classrooms. *Technology enhanced learning: Opportunities for change*, 235–263.

Dadic, T., Stankov, S., & Rosic, M. (2006). Prototype model of tutoring system for programming. In *Information technology interfaces, 2006. 28th international conference on* (pp. 41–46).

Dadic, T., Stankov, S., & Rosic, M. (2008). Meaningful learning in the tutoring system for programming. In *Information technology interfaces, 2008. iti 2008. 30th international conference on* (pp. 483–488).

Danielson, R. L., & Nievergelt, J. (1975). An automatic tutor for introductory programming students. In *Proceedings of the fifth SIGCSE technical symposium on Computer science education* (pp. 47–50).

Delić, H., & Bećirović, S. (2016). Socratic method as an approach to teaching. *European researcher. Series A*(10), 511–517.

DeLooze, L. L. (1991). *Its ada: An intelligent tutoring system for the ada programming language* (Tech. Rep.). Naval Postgraduate School Monterey CA.

Desmarais, M. C., & Baker, R. S. (2012). A review of recent advances in learner and skill modeling in intelligent learning environments. *User Modeling and User-Adapted Interaction*, *22*(1-2), 9–38.

Dobre, I. (2014). An Intelligent Tutoring System for Tutoring the Computers Programming and C Language Discipline. In *The International Scientific Conference E-Learning and Software for Education* (Vol. 2, p. 142).

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73.

Freedman, R., Rosé, C. P., Ringenberg, M. A., & VanLehn, K. (2000). ITS tools for natural language dialogue: A domain-independent parser and planner. In *International Conference on Intelligent Tutoring Systems* (pp. 433–442).

Gálvez, J., Guzmán, E., & Conejo, R. (2009). A blended E-learning experience in a course of object oriented programming fundamentals. *Knowledge-Based Systems*, *22*(4), 279–286.

Goldin, A. P., Pedroncini, O., & Sigman, M. (2017). Producing or reproducing reasoning? Socratic dialog is very effective, but only for a few. *PloS one*, *12*(3), e0173584.

Good, J. (1999). *Programming paradigms, information types and graphical representations: Empirical investigations of novice comprehension* (Unpublished doctoral dissertation). Doctoral dissertation, University of Edinburgh.

Goyal, M. (2011). Development of agent-based intelligent tutoring system for teaching object-oriented programming concepts. In *Proceedings of the 9th International Conference on Education and Information Systems, Technologies and Applications (EISTA 2011)* (pp. 17–22).

Graesser, A. C., D'Mello, S., & Person, N. (2009). 19 meta-knowledge in tutoring. *Handbook of metacognition in education*, 361.

Graesser, A. C., Lu, S., Jackson, G. T., Mitchell, H. H., Ventura, M., Olney, A., & Louwerse, M. M. (2004). AutoTutor: A tutor with dialogue in natural language. *Behavior Research Methods, Instruments, & Computers*, *36*(2), 180–192.

Graesser, A. C., McNamara, D. S., & Kulikowich, J. M. (2011). Coh-Metrix: Providing multilevel analyses of text characteristics. *Educational researcher*, *40*(5), 223–234.

Graesser, A. C., Person, N. K., & Magliano, J. P. (1995). Collaborative dialogue patterns in naturalistic one-to-one tutoring. *Applied cognitive psychology*, *9*(6), 495–522.

Gross, S., & Pinkwart, N. (2015). Towards an integrative learning environment for Java programming. In *Advanced learning technologies (icalt), 2015 ieee 15th international conference on* (pp. 24–28).

Haddad, I. A. E., & Naser, S. S. A. (2017). ADO-Tutor: Intelligent Tutoring System for Leaning ADO.NET. *European Academic Research*, *4*(10).

Hahn, S.-H., & Kim, J. (1997). Automatic problem description from model program for knowledge-based programming tutor. *Automated Software Engineering*, *4*(4), 439–461.

Hastie, T., Friedman, J., & Tibshirani, R. (2001). Model assessment and selection. In *The elements of statistical learning* (pp. 193–224). Springer.

Hattori, N., & Ishii, N. (1999). An extended educational system for programming and its evaluation. *Information and Software Technology*, *41*(8), 525–532.

Heffernan, N. T., Turner, T. E., Lourenco, A. L., Macasek, M. A., Nuzzo-Jones, G., & Koedinger, K. R. (2006). The ASSISTment Builder: Towards an Analysis of Cost Effectiveness of ITS Creation. In *FLAIRS Conference* (pp. 515–520).

Holland, J., Mitrovic, A., & Martin, B. (2009). J-LATTE: a Constraint-based Tutor for Java. *Hong Kong: 17th International on Conference Computers in Education (ICCE 2009)*, 142-146.

Holmes, M., Latham, A., Crockett, K., Lewin, C., & O'Shea, J. (2015). Hendrix: A conversational intelligent tutoring system for Java programming. In *Proceedings of UKCI 2015 15th UK Workshop on Computational Intelligence.*

Hung, Y.-C. (2012). Combining self-explaining with computer architecture diagrams to enhance the learning of assembly language programming. *IEEE Transactions on Education*, *55*(4), 546–551.

Jackson, J. D. (2006). Socrates and Langdell in legal writing: Is the Socratic method a proper tool for legal writing courses. *Cal. WL Rev.*, *43*, 267.

Johnson, W. L. (1990). Understanding and debugging novice programs. *Artificial intelligence*, *42*(1), 51–97.

Johnson, W. L., & Soloway, E. (1985). PROUST: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*(3), 267–275.

Jordan, P., Rosé, C. P., & VanLehn, K. (2001). Tools for authoring tutorial dialogue knowledge. In *Proceedings of AI in Education 2001 Conference.*

Jurafsky, D., & Martin, J. H. (2008). Speech and Language Processing: An introduction to speech recognition, computational linguistics and natural language processing. *Upper Saddle River, NJ: Prentice Hall.*

Karkalas, S., & Gutierrez-Santos, S. (2014). Enhanced Javascript learning using code quality tools and a rule-based system in the flip exploratory learning environment. In *2014 IEEE 14th International Conference on Advanced Learning Technologies* (pp. 84–88).

Kass, R. (1989). Student modeling in intelligent tutoring systems—implications for user modeling. In *User models in dialog systems* (pp. 386–410). Springer.

Kay, J., & Kummerfeld, R. (1994). An individualised course for the C programming language. In *Proceedings of Second International WWW Conference* (pp. 17–20).

Keim, G., Fulkerson, M., & Biermann, A. (1997). Initiative in tutorial dialogue systems. In *Proceedings of the American Association for Artificial Intelligence (AAAI) Spring Symposium on Computational Models for Mixed-Initiative Interaction.*

Khayi, N. A., & Rus, V. (2019). BI-GRU Capsule Networks for Student Answers Assessment. In *2019 KDD Workshop on Deep Learning for Education (DL4Ed).*

Kintsch, W., & Walter Kintsch, C. (1998). *Comprehension: A paradigm for cognition.* Cambridge university press.

Kularbphettong, K., Kedsiribut‌, P., & Roonrakwit, P. (2015). Developing an adaptive web-based intelligent tutoring system using mastery learning technique. *Procedia-Social and Behavioral Sciences*, *191*, 686–691.

Kwon, K., & Jonassen, D. H. (2011). The influence of reflective self-explanations on problem-solving performance. *Journal of Educational Computing Research*, *44*(3), 247–263.

Kwon, K., Kumalasari, C. D., & Howland, J. L. (2011). Self-Explanation Prompts on Problem-Solving Performance in an Interactive Learning Environment. *Journal of Interactive Online Learning*, *10*(2).

Lane, H. C., & VanLehn, K. (2003). Coached program planning: dialogue-based support for novice program design. In *ACM SIGCSE Bulletin* (Vol. 35(1), pp. 148–152).

Lane, H. C., & VanLehn, K. (2004). A Dialogue-Based Tutoring System for Beginning Programming. In *FLAIRS Conference* (pp. 449–454).

Lee, C., & Baba, M. S. (2005). The intelligent web-based tutoring system using the C++ standard template library. *Malaysian Online Journal of Instructional Technology (MOJIT)*, *2*(3), 34–42.

Lee, P., & Phillips, C. (1998). Programming versus design (poster): teaching first year students. In *Acm sigcse bulletin* (Vol. 30, p. 289).

Letovsky, S., & Soloway, E. (1986). Delocalized plans and program comprehension. *IEEE Software*, *3*(3), 41.

Liu, L., Wang, H., Li, C., & Zhao, C. (2011). Research and application of plan recognition in intelligent tutoring system. In *2011 International Symposium on it in Medicine and Education (ITME)* (Vol. 2, pp. 53–56).

Margulieux, L. E., Guzdial, M., & Catrambone, R. (2012). Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. In *Proceedings of the ninth annual international conference on International computing education research* (pp. 71–78).

Marx, J. D., & Cummings, K. (2007). Normalized change. *American Journal of Physics*, *75*(1), 87–91.

McCalla, G., & Murtagh, K. (1985). *GENIUS: An experiment in ignorance-based automated program advising.* University of Saskatchewan, Department of Computational Science.

Mcgettrick, A., Boyle, R., Ibbett, R., Lloyd, J., Lovegrove, G., & Mander, K. (2005). Grand challenges in computing: Education—a summary. *The Computer Journal*, *48*(1), 42–48.

Medeiros, R. P., Ramalho, G. L., & Falcão, T. P. (2018). A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, *62*(2), 77–90.

Miller, J. R., Kehler, T. P., Michaelis, P. R., & Murray, W. R. (1982). *Intelligent Tutoring for Programming Tasks: Using Plan Analysis to Generate Better Hints* (Tech. Rep.). Texas Instruments inc Dallas central research labs.

Mitrović, A. (1998). Experiences in implementing constraint-based modeling in sql-tutor. In *International conference on intelligent tutoring systems* (pp. 414–423).

Moore, J. D., & Moore, J. D. (1995). *Participating in explanatory dialogues: interpreting and responding to questions in context*. MIT press Cambridge, MA.

Muansuwan, N., Sirinaovakul, B., & Thepruangchai, P. (2004). Intelligent tutoring and knowledge base creation for the subject of computer programming. In *Tencon 2004. 2004 ieee region 10 conference* (pp. 353–356).

Murray, T. (2003). An Overview of Intelligent Tutoring System Authoring Tools: Updated analysis of the state of the art. In *Authoring tools for advanced technology learning environments* (pp. 491–544). Springer.

Naser, S. S. A. (2008a). Developing an intelligent tutoring system for students learning to program in C++. *Information technology journal*, *7*(7), 1055–1060.

Naser, S. S. A. (2008b). JEE-Tutor: An Intelligent Tutoring System For Java Expressions Evaluation. *Information Technology Journal*, *7*(3), 528–532.

Nguyen, A., Piech, C., Huang, J., & Guibas, L. (2014). Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd international conference on World wide web* (pp. 491–502).

Núñez, A., Fernández, J., Garcia, J. D., Prada, L., & Carretero, J. (2008). M-PLAT: Multi-programming language adaptive tutor. In *Eighth IEEE International Conference on Advanced Learning Technologies, 2008. ICALT'08* (pp. 649–651).

Odekirk-Hash, E., & Zachary, J. L. (2001). Automated feedback on programs means students need less help from teachers. In *Acm sigcse bulletin* (Vol. 33, pp. 55–59).

Ohlsson, S. (1994). Constraint-based student modeling. In *Student modelling: the key to individualized knowledge-based instruction* (pp. 167–189). Springer.

O'brien, M. P. (2003). Software comprehension–a review & research direction. *Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report*.

Pennington, N. (1987a). Comprehension strategies in programming. In *Empirical studies of programmers: second workshop* (pp. 100–113).

Pennington, N. (1987b). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, *19*(3), 295–341.

Person, N., Lehman, B., & Ozbun, R. (2007). Pedagogical and motivational dialogue moves used by expert tutors. In *17th Annual Meeting of the Society for Text and Discourse. Glasgow, Scotland.*

Petersen, A., Craig, M., Campbell, J., & Tafliovich, A. (2016). Revisiting why students drop CS1. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (pp. 71–80).

Pillay, N. (2003). Developing intelligent programming tutors for novice programmers. *ACM SIGCSE Bulletin*, *35*(2), 78–82.

Pintrich, P. R., Berger, C. F., & Stemmer, P. M. (1987). Students' programming behavior in a pascal course. *Journal of Research in Science Teaching*, *24*(5), 451–466.

Pirolli, P., & Recker, M. (1994). Learning strategies and transfer in the domain of programming. *Cognition and instruction*, *12*(3), 235–275.

Piteira, M., & Costa, C. (2013). Learning computer programming: study of difficulties in learning programming. In *Proceedings of the 2013 International Conference on Information Systems and Design of Communication* (pp. 75–80).

Pressley, M., Wood, E., Woloshyn, V. E., Martin, V., King, A., & Menke, D. (1992). Encouraging mindful use of prior knowledge: Attempting to construct explanatory answers facilitates learning. *Educational psychologist*, *27*(1), 91–109.

Proulx, V. K. (2000). Programming patterns and design patterns in the introductory computer science course. In *ACM SIGCSE Bulletin* (Vol. 32(1), pp. 80–84).

Ramalingam, V., LaBelle, D., & Wiedenbeck, S. (2004). Self-efficacy and mental models in learning to program. In *Proceedings of the 9th annual SIGCSE* *conference on Innovation and technology in computer science education* (Vol. 36(3), pp. 171–175).

Rashid, P. Q. (2015). *Semantic network and frame knowledge representation formalisms in artificial intelligence* (Unpublished doctoral dissertation). Doctoral dissertation, Eastern Mediterranean University (EMU)-Doğu Akdeniz Üniversitesi (DAÜ).

Recker, M. M., & Pirolli, P. (1990). A Model of Self-Explanation Strategies of Instructional Text and Examples in the Acquisition of Programming Skills.

Renumol, V., Jayaprakash, S., & Janakiram, D. (2009). Classification of cognitive difficulties of students to learn computer programming. *Indian Institute of Technology, India*, *12*.

Rezel, E. S. (2003). The effect of training subjects in self-explanation strategies on problem solving success in computer programming.

Rivers, R. (1989). Embedded user models—where next? *Interacting with Computers*, *1*(1), 13–30.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer science education*, *13*(2), 137–172.

Rosé, C. P., Moore, J. D., VanLehn, K., & Allbritton, D. (2001). A comparative evaluation of Socratic versus didactic tutoring. In *Proceedings of the Annual Meeting of the Cognitive Science Society* (Vol. 23).

Roy, M., & Chi, M. T. (2005). The self-explanation principle in multimedia learning. *The Cambridge handbook of multimedia learning*, 271–286.

Rugaber, S. (2000). The use of domain knowledge in program understanding. *Annals of Software Engineering*, *9*(1-2), 143–192.

Rus, V., D'Mello, S., Hu, X., & Graesser, A. (2013). Recent advances in conversational intelligent tutoring systems. *AI magazine*, *34*(3), 42–54.

Rus, V., Stefanescu, D., Baggett, W., Niraula, N., Franceschetti, D., & Graesser, A. C. (2014). Macro-adaptation in conversational intelligent tutoring matters. In *International conference on intelligent tutoring systems* (pp. 242–247).

Schulte, C. (2008). Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceedings of the Fourth international Workshop on Computing Education Research* (pp. 149–160).

Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., & Paterson, J. H. (2010). An introduction to program comprehension for computer science educators. In *Proceedings of the 2010 ITiCSE working group reports* (pp. 65–86).

Shaffer, S. C. (2005). Ludwig: an online programming tutoring and assessment system. *ACM SIGCSE Bulletin*, *37*(2), 56–60.

Shaft, T. M. (1992). The role of application domain knowledge in computer program comprehension and enhancement.

Sheppard, S. (2007). *The History of Legal Education in the United States: Commentaries and Primary Sources* (Vol. 1). The Lawbook Exchange, Ltd. (shows positive effect of Socratic method on Legal education)

Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, *8*(3), 219–238.

Sierra, E., Hossian, A., Britos, P., Rodriguez, D., & Garcia-Martinez, R. (2007). A multi-agent intelligent tutoring system for learning computer programming. In *Electronics, Robotics and Automotive Mechanics Conference (CERMA 2007)* (pp. 382–385).

Singley, M. K., Carroll, J. M., & Alpert, S. R. (1993). Incidental reification of goals in an intelligent tutor for smalltalk. In *Cognitive Models and Intelligent Environments for Learning Programming* (pp. 145–155). Springer.

Sleeman, D., & Brown, J. S. (1982). *Intelligent tutoring systems.* London: Academic Press.

Snow, C. (2002). *Reading for understanding: Toward an R&D program in reading comprehension.* Rand Corporation.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on software engineering*(5), 595–609.

Soloway, E., & Spohrer, J. C. (2013). *Studying the novice programmer.* Psychology Press.

Soloway, E. M., Woolf, B., Rubin, E., & Barth, P. (1981). Meno-ii: An intelligent tutoring system for novice programmers. In *Proceedings of the 7th international joint conference on Artificial intelligence-Volume 2* (pp. 975–977).

Song, J. S., Hahn, S. H., Tak, K. Y., & Kim, J. H. (1997). An intelligent tutoring system for introductory C language course. *Computers & Education*, *28*(2), 93–102.

Sowa, J. F. (1992). Semantic networks. *Encyclopedia of artificial intelligence*, *2*, 1493–1511.

Spohrer, J. C., & Soloway, E. (1989). Simulating Student Programmers. In *IJCAI* (Vol. 89, pp. 543–549).

Stansfield, J. L., Carr, B. P., & Goldstein, I. P. (1976). Wumpus advisor 1: A first implementation program that tutors logical and probabilistic reasoning skills.

Sykes, E. R., & Franek, F. (2003). A prototype for an intelligent tutoring system for students learning to program in Java (TM). In *Proceedings of the IASTED International Conference on Computers and Advanced Technology in Education* (pp. 78–83).

Tchetagni, J., Nkambou, R., & Bourdeau, J. (2005). Supporting student reflection in an intelligent tutoring system for logic programming. In *Proceedings of the Workshop on LeMore at Artificial Intelligence in Education, 12th International Conference, AIED2005* (pp. 42–51).

Tom, M. (2015). Five C Framework: A Student-Centered Approach for Teaching Programming Courses to Students with Diverse Disciplinary Background. *Journal of Learning Design*, *8*(1), 21–27.

Truong, N., Bancroft, P., & Roe, P. (2005). Learning to program through the web. *ACM SIGCSE Bulletin*, *37*(3), 9–13.

Ueno, H. (1991). An integrated knowledge-based intelligent programming environment for novice programmers. In *1991 The Fifteenth Annual International Computer Software & Applications Conference* (pp. 124–125).

VanLehn, K. (1988). Student modeling. *Foundations of intelligent tutoring systems*, *55*, 78.

Vanlehn, K. (2006). The behavior of tutoring systems. *International journal of artificial intelligence in education*, *16*(3), 227–265.

VanLehn, K., Graesser, A. C., Jackson, G. T., Jordan, P., Olney, A., & Rosé, C. P. (2007). When are tutorial dialogues more effective than reading? *Cognitive science*, *31*(1), 3–62.

Vesin, B., Ivanović, M., & Budimac, Z. (2009). Learning management system for programming in Java. *Annales Universitatis Scientiarum De Rolando Eötvös Nominatae, Sectio-Computatorica*, *31*, 75–92.

Vesin, B., Ivanović, M., Klašnja-Milićević, A., & Budimac, Z. (2015). Personal Assistance Agent in Programming Tutoring System. In *Agent and Multi-Agent Systems: Technologies and Applications* (pp. 441–451). Springer.

Von Mayrhauser, A., & Vans, A. M. (1995). Program understanding: Models and experiments. In *Advances in Computers* (Vol. 40, pp. 1–38). Elsevier.

Wang, W. (2011). Teaching programming online. In *International conference on the future of education .*

Warnakulasooriya, R., Palazzo, D. J., & Pritchard, D. E. (2006). Evidence of Problem-Solving Transfer in Web-Based Socratic Tutor. In *AIP Conference Proceedings* (Vol. 818, pp. 41–44). (shows positive effect of Socratic method on students knowledge (Physics))

Weber, G., & Brusilovsky, P. (2001). ELM-ART: An adaptive versatile system for Web-based instruction. *International Journal of Artificial Intelligence in Education (IJAIED)*, *12*, 351–384.

Wescourt, K. T., Beard, M., & Gould, L. (1977). Knowledge-based adaptive curriculum sequencing for CAI: Application of a network representation. In *Proceedings of the 1977 annual conference* (pp. 234–240).

Woods, P. J., & Warren, J. R. (1995). Rapid prototyping of an intelligent tutorial system. *Proceedings of 12th Australian Society for Computers in Learning in Tertiary Education*, 557–563.

Xu, L., & Sarrafzadeh, A. (2004). Haskell-Tutor: An Intelligent Tutoring System for Haskell Programming language. In *Postgraduate Conference of the Institute of Information and Mathematical Sciences* (p. 108).

Yuasa, M. (1994). The effects of active learning exercises on the acquisition of SQL query writing procedures.

Zwaan, R. A., & Radvansky, G. A. (1998). Situation models in language comprehension and memory. *Psychological bulletin*, *123*(2), 162.