Say Hello to '*Coding Tutor*'! Design and Evaluation of a Chatbot-based Learning System Supporting Students to Learn to Program

Completed Research Paper

Sebastian Hobert
University of Goettingen
Platz der Göttinger Sieben 5, D-37073 Göttingen
shobert@uni-goettingen.de

Abstract

The overall goal of this design science research project is to design and evaluate a chatbot-based learning system that is able to support novice programmers to learn to write software code in formal learning settings. Our results indicate that such a conversational intelligent programming tutor is suited to take over tasks of teaching assistants in times when no human teaching assistant or lecturer is available for help, e.g., due to resource constraints. For instance, our so-called 'Coding Tutor' is able to respond to open-ended knowledge questions, to assess submitted source code automatically or to guide students step-by-step through programming exercises using natural language communication. In addition to providing our situated software artifact, we document our results based on the core components of a design theory as proposed by Gregor and Jones (2007). Thus, we provide the first step toward a nascent design theory for chatbot-based learning systems in IS education.

Keywords: Chatbot-based learning system, intelligent programming tutor, pedagogical conversational agent, automatic assessment system, design theory

Introduction

The importance of technology in our daily life has been growing rapidly for many years and it will be essential for shaping our future world. Understanding technology and being able to apply it in the increasingly digital business world is an essential skill for students who want to get adequate preparation for future workplaces. This is and will not only be true for computer scientists and information systems employees, but also for most future workplaces (Popat and Starkey 2019). One important skill that is essential for getting an in-depth understanding of technology, is learning to understand and write software programs – known as *programming* or *coding*. However, currently, only less than 10% of all adults are capable of coding and do it on a regular basis (OECD 2018). Due to the growing importance, teaching to code software is getting part of today's curriculums of primary, secondary and higher education. However, students are often not adequately skilled in programming and consider it as a difficult and complex task (Daradoumis et al. 2019). One main reason why learning to code is challenging (Vial and Negoita 2018) for many students is that it requires multiple knowledge dimensions (like conceptual and procedural knowledge; Passier 2017) and includes several cognitive processes (like understanding, analyzing and evaluating) according to Bloom's revised taxonomy (Anderson and Krathwohl 2001; Bloom et al. 1956). For instance, before being able to create a new piece of software, students need the abilities (1) to understand the programming task, (2) to analyze it in order to identify proper solutions and (3) to evaluate different solution options.

To face this challenge in education, introductory programming courses usually combine multiple teaching approaches like large-scale lectures to teach basic algorithmic skills and classroom tutorials in smaller groups to provide guidance for programming exercises that encourage students to practice coding at home. Despite these – in comparison to other courses – far-reaching support offers that are common in today's programming courses, many students still struggle with it. Even if they have understood the learning contents in the lectures, the homework tasks often pose a great challenge to many of the novice programmers. The creative process of writing software code seems particularly demanding for beginners, especially if no guidance is available (Passier 2017). To address this, in-class tutorial sessions are a starting point as experienced teaching assistants can support the students. However, due to resource constraints, the students-to-lecturer ratio is usually still high in tutorial sessions (often up-to 20-to-1 or even higher) and the timeframe in which teaching assistants are available for help is often limited to only a few hours per week.

To further assist novice programmers, e-learning tools are available as outlined by Sim and Lau (2018) and Crow et al. (2018) in two recent systematic reviews of prior research. Particularly, so-called *intelligent programming tutors*, are state-of-the-art. While there is no distinct feature set of intelligent programming tutors, the most common features include (1) the provision of learning content (e.g., tutorials, explanations of concepts, and formative assessments to query factual knowledge using quiz questions), and (2) automatic assessment tools that evaluate the students' homework assignments automatically. Those existing e-learning systems may improve the level of support of novice programmers and reduce the workload of teaching assistants. However, current systems are not yet able to provide equally good support as teaching assistants would do during in-class tutorial sessions. Particularly, the ability to individually respond to arising questions of novice programmers and to provide adequate explanations are not yet possible in common intelligent tutoring systems. Thus, programmers still face the challenge to solve programming tasks on their own. To address this problem of insufficient communication opportunities and missing individualized support of novice programmers, we leverage the advances in intelligent tutoring systems particularly in chatbot-based learning systems known as *pedagogical conversational agents* (Tamayo-Moreno and Perez-Marin 2016) and adapt them to the field of programming education. Using these technologies, we aim at developing an intelligent programming tutoring system that is able to communicate with novice programmers in natural language using a chatbot interface in order to assist them on an individualized level while solving programming tasks in a programming environment. To achieve our goal of providing students with assistance in a similar manner as teaching assistants would do, we follow a design science research approach (Hevner 2007) to answer the following research questions:

RQ1: What requirements should be considered when designing a chatbot-based learning system that aims at supporting students to solve coding tasks?

RQ2: How do novice programmers assess the changed learning concept based on the developed chatbot-based learning system?

To answer these research questions, the remainder of this paper is structured as follows: In the next section, we describe the theoretical background by focusing on related research and the *ICAP framework* (Chi and Wylie 2014) as our kernel theory. Following this, we describe our research design based on the design science research process and outline in detail how we design our chatbot-based learning system based on both, scientific literature as well as empirical results. After describing our design and evaluation process in eight consecutive steps, we discuss the results and document the generated design knowledge as proposed by Gregor and Jones (2007). Finally, we summarize our results, discuss limitations and point out future research directions in the conclusion.

Theoretical Background

Chatbot-based Learning Systems

Chatbots (also known as chatterbots, conversational agents or talkbots) can be defined as interactive information systems that provide natural language user interfaces and attempt to conduct conversations similar to human beings (Winkler and Söllner 2018). By applying methods known from artificial intelligence, machine learning, and natural language processing, chatbots usually act autonomously either in a reactive or proactive way. To handle natural language input and output, typical technical architectures of chatbots consist of a natural language understanding component that preprocesses the user's input and redirects it to a dialog manager. After computing an appropriate answer based on inputs from a knowledge base, the natural language generation component generates an output message that is sent to the user.

Chatbots have been researched since the 1960ies when Weizenbaum (1966) invented ELIZA – the first chatbot that focused on natural language communication. Other prior chatbots, like the widely known chatbot A.L.I.C.E (Wallace 2009), mainly applied rule-based natural language processing tools. Nowadays more sophisticated natural language processing tools are available. Due to these technology improvements in recent years, chatbots started to spread across many domains. As indicated by recent literature reviews targeting chatbot-based learning systems (e.g., Hobert and Meyer von Wolff 2019; Winkler and Söllner 2018), this also resulted in an increased availability and research interest of so-called pedagogical conversational agents. Pedagogical conversational agents have been developed and analyzed in prior research: For instance, Mikic et al. (2009) developed *Charlie*, a chatbot based on the AIML rule-based language, which provides an alternative user-interface to an e-learning platform. The *MentorChat* software by Tegos et al. (2011) is a conversational agent that implements functionalities to support collaborative learning tasks in chat discussions. Further, Graesser et al. (2017) implemented the *AutoTutor* prototype that focuses on presenting complex problems that require some form of reasoning. Besides these mentioned chatbot-based learning systems further software prototypes have been researched (see e.g. Hobert and Meyer von Wolff (2019) for further examples). However, formal learning settings have only been researched in few cases. Additionally, most available chatbots focus on very specific use cases. Thus, students usually use those chatbots only for a very short time frame. In contrast to that, the learning settings in programming courses as described in the introduction setting would require the regular use of a chatbot-based learning system during the whole semester. For instance, if students need to solve programming tasks every single week, the chatbot needs to be capable of supporting them each time. Thus, the scope of the chatbot needs to be much broader.

E-Learning Approaches in Introductory Programming Courses

E-learning or technology-enhanced learning systems have been used by lecturers of introductory programming courses for a long time. They are known in the literature under the term intelligent programming tutors (Crow et al. 2018; Sim and Lau 2018). Often the aim of such systems is to provide learning contents about basic programming skills via learning management systems or to give access to formative assessments (e.g., single- or multiple-choice quizzes). By providing the learning contents online instead of teaching it in a lecture, blended learning scenarios (e.g., Albrecht et al. 2018) have already been tested. In one case, a conversational agent was used to support students by answering questions about Java terms (Müller et al. 2018). In addition to providing the learning contents via e-learning systems, automatic assessment systems like *JACK* (Goedicke et al. 2018), *Praktomat* (Breitner et al. 2017; Zeller 2000) or many others (e.g., Daradoumis et al. 2019) are widespread among programming courses and are even used in MOOCs (Bey et al. 2018). Often the main reason for using such automatic assessment systems is to reduce the workload of lecturers or teaching assistants to evaluate the submitted homework assignments. This enables lectures to be conducted even if the number of students is increasing rapidly in a programming course as evaluating the exercises is not dependent on the students-to-lecturer ratio anymore (Bey et al. 2018). Students also can benefit from automatic assessment systems as they get immediate feedback about their submitted tasks, i.e. after uploading their source code, the system automatically evaluates it and is able to provide instant feedback (Ihantola et al. 2010). From a technical perspective, automatic assessment systems are capable of evaluating the students' source code with static or dynamic code analysis. Common test scenarios are for instance to compile the submitted source code and to apply dynamic software tests, e.g., JUnit for the Java programming language.

The currently used e-learning systems are capable of supporting novice programmers and can reduce the workload of teaching assistants as homework assignments can be evaluated automatically. Thus, both common usage approaches of intelligent programming tutors – the provision of contents via a learning management system and the automatic evaluation of programming exercises – seems suited to deal with a growing number of students and limited teaching assistant resources. However, the research problem we outlined in the introduction section has not been adequately targeted in prior research. To solve this problem of missing individualized support (e.g., answering of arising questions while trying to solve homework assignments) and missing guidance for novice programmers in times when no lecturers or teaching assistants are available, more sophisticated solutions are required that focus specifically on the needs of students. Chatbot-based learning systems are particularly suited to provide such an interactive and individualized interaction as outlined in the previous subsection. Hence, combining a natural language-based personal assistant interface of chatbot-based learning systems with an automatic assessment system of intelligent programming tutors, pledges to be beneficial for providing a suited learning support. It is to be expected that such a chatbot-enabled intelligent tutoring system is capable of answering arising questions of novice programmers and guiding them individually through solving programming tasks. Thus, such a system to be designed in this research project combines tasks of common intelligent programming tutoring systems with tasks that are currently carried out by teaching assistants.

ICAP Framework as a Kernel Theory

The added value of using the technology of chatbots in formal learning settings in comparison to other common technology-enhanced learning systems is given by an increased engagement of the students due to the conversational interaction of the learners with the chatbot. According to the ICAP Framework (i.e., *Interactive, Constructive, Active and Passive Framework*) by Chi and Wylie (2014), the learners' engagement with learning materials can range "from passive to active to constructive to interactive" (Chi and Wylie 2014) and will result in an improved learning outcome. Whereas in passive engagement activities students only consume or receive learning materials, in active engagement activities students are able to manipulate the content presentation, e.g., by highlighting important text passages. In the two most engaging forms of interaction according to Chi and Wylie (2014), students deepen their interaction, e.g., by comparing the learning materials with prior knowledge (constructive engagement), by debating with others or asking and answering questions (interactive engagement).

Following the hypothesis of the ICAP Framework, chatbot-based learning systems are capable of fostering the students' engagement as they add the new component of dialoguing to e-learning systems in introductory programming courses. In comparison to common e-learning approaches like e-learning modules or automatic assessment systems as described above, chatbots can also start discourses and discussions with students about the learning contents – just like human teaching assistants would do. Thus, an interactive engagement according to the ICAP Framework can be achieved even in times when no human teaching assistants are available.

Research Design

To address our research goal by developing an innovative learning system for supporting students of introductory programming courses individually when no teaching assistants are available for help, we apply a design science research (DSR) method. Using this approach, we aim at (1) providing a relevant solution for the predominant problem setting as described in the introductory section by applying a scientific approach and (2) deriving generalized design implications for the Information Systems (IS) research discipline according to Gregor and Jones (2007).

We use the three cycle information systems research framework by Hevner (2007) and Hevner et al. (2004) as a basis and apply eight research steps accordingly in the *relevance*, *rigor* and *design cycles* as displayed in Figure 1. In the first step, we formulated the research problem by deriving it from common settings in introductory programming courses as outlined in the introduction section. In addition to this practice-oriented motivation, we specified the problem from an educational perspective by deriving the learning objectives of our problem setting using the revised Bloom's Taxonomy (Anderson and Krathwohl 2001; Bloom et al. 1956) as outlined in the following step 1 of our research processes. Following the problem specification, we identified user stories and related requirements from both, scientific literature and educational practice. To get insights into the educational practice, we conducted an expert workshop with eight participants (lecturers, e-learning experts and instructional designers) in which we discussed how pedagogical conversational agents could be used for supporting students in introductory university courses. Based on this, we derived design principles and a conceptual artifact in a first design iteration. We evaluated the conceptual artifact afterward with both, 28 students and 16 teaching assistants, to get feedback about this first iteration of the artifact early in the design process. Using the feedback of this first evaluation, we revised our design features and the conceptual design. As the outcome of our second design iteration, we got a fully functional software artifact called *Coding Tutor* for the specified problem. In a second evaluation setting, 40 other novice programming students tested our software artifact in a field test in the following semester term. During this second evaluation, the students used our Coding Tutor artifact to solve a predefined programming task. By first evaluating the design features in the conceptual design (step 5) and then the software artifact (step 7), we conducted a systematic evaluation before and after implementing the artifact. During both evaluation studies, we followed the evaluation strategy for design science research as proposed by Venable et al. (2016). Finally, after finishing the evaluation, we documented the design knowledge using the core components of a design theory as proposed by Gregor and Jones (2007).

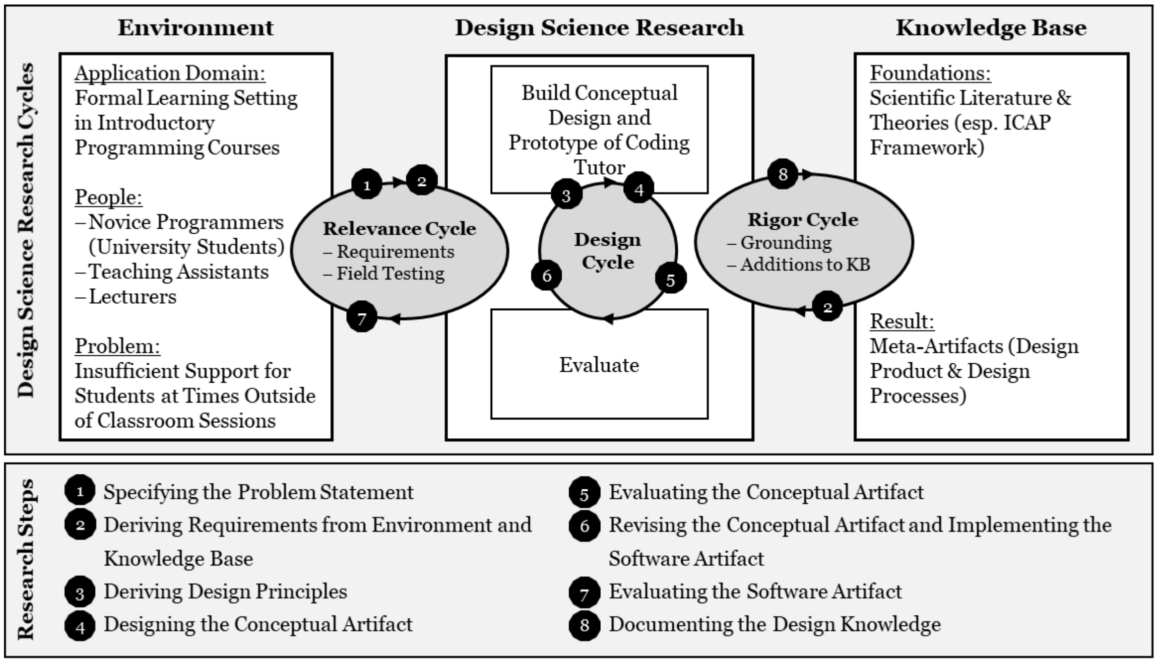

Design Science Research Cycles
Environment
Design Science Research
Knowledge Base
Application Domain:
Formal Learning Setting in Introductory Programming Courses
People:
– Novice Programmers (University Students)
– Teaching Assistants
– Lecturers
Problem:
Insufficient Support for Students at Times Outside of Classroom Sessions
Build Conceptual Design and Prototype of Coding Tutor
Relevance Cycle
– Requirements
– Field Testing
Design Cycle
Evaluate
Rigor Cycle
– Grounding
– Additions to KB
Foundations:
Scientific Literature & Theories (esp. ICAP Framework)
Result:
Meta-Artifacts (Design Product & Design Processes)
Research Steps
1 Specifying the Problem Statement
2 Deriving Requirements from Environment and Knowledge Base
3 Deriving Design Principles
4 Designing the Conceptual Artifact
5 Evaluating the Conceptual Artifact
6 Revising the Conceptual Artifact and Implementing the Software Artifact
7 Evaluating the Software Artifact
8 Documenting the Design Knowledge

Figure 1. Adapted Research Approach (based on Hevner et al. 2004; Hevner 2007; Rietsche et al. 2018)

Designing and Evaluating the Coding Tutor

In the following, we outline the results of all eight steps of our design science research approach with the aim of developing our software artifact called *Coding Tutor*.

#1 Specifying the Problem Statement

As motivated in the introduction section of this paper, we address the challenge of supporting students in large-scale introductory programming courses when they solve homework exercises. Our main goal from an educational perspective is to provide novice programmers with an e-learning system that guides them individually through solving programming tasks when there is no human teaching assistant available (e.g., at the weekend). Main reasons for the challenge to provide individual support in large scale lectures are resource constraints (Hien et al. 2018). To overcome this challenge, the learning system to be created needs to be able to support students in a similar way a teaching assistant would do in in-class trainings.

To specify the research problem, we define the learning objectives based on the revised Bloom's Taxonomy (Anderson and Krathwohl 2001; Bloom et al. 1956). The taxonomy differentiates four different dimensions of knowledge: *factual*, *conceptual*, *procedural* and *metacognitive*. Additionally, six categories exist in the cognitive process dimension: *remember*, *understand*, *apply*, *analyze*, *evaluate* and *create*. By combining both dimensions, learning objectives can be categorized as intersections between knowledge type and cognitive process.

In programming exercises of introductory programming courses particularly three main learning objectives are targeted: First, novice programmers need to understand the syntax of the programming language (objective 1). Especially important is to get familiar with the syntax and semantic of commands including their functionality. Thus, this first objective serves as a basis for programming and can be assigned as factual knowledge. In order to be able to solve programming exercises, conceptual knowledge is needed as well. In particular, students need to understand and to be able to apply the underlying algorithms of the programming problem to be solved (objective 2a). Otherwise, it is usually not possible to solve a programming task, if the underlying theoretical knowledge is missing. In addition to that, the students should be able to break the algorithmic problem into smaller subtasks (objective 2b). This is especially important as soon as the programming tasks are getting more complex. Finally, the actual coding process of writing source code can be categorized as procedural knowledge. This task encompasses, in particular, to translate the algorithm into source code and to structure the overall code. Due to this, the actual coding during exercises goes beyond *apply* and can be classified as an *analytic* cognitive process. Table 1 summarizes the learning objectives in the revised Bloom's Taxonomy (Anderson and Krathwohl 2001).

		Cognitive Process Dimension					
		Remember	**Understand**	**Apply**	**Analyze**	**Evaluate**	**Create**
Knowledge Dimension	**Factual**		Objective 1				
	Conceptual			Objective 2a	Objective 2b		
	Procedural				Objective 3		
	Metacognitive						

Table 1. Learning Objectives Taxonomy (based on Anderson and Krathwohl 2001)

#2 Deriving Requirements from Environment and Knowledge Base

To deduce requirements for the specified problem, we considered the environmental perspective (*relevance*) as well as the knowledge base (*rigor*). To analyze the environmental perspective, we first conducted an expert workshop with eight participants (lecturers, e-learning experts and instructional designers) in which we derived application scenarios of chatbot-based learning systems. Based on this, we identified three user stories that are relevant for supporting students in introductory programming courses (see Figure 2 below): (U1) Students need to understand the theoretical problem statement of programming exercises, (U2) students need to be able to apply the underlying algorithm and (U3) students need to be able to transfer the theoretical algorithms into source code. Those user stories are directly related to the learning objectives as described in step 1. Second, we use the task characteristics of teaching assistants in programming courses as a basis to derive further requirements. This seems especially important as the learning system to be designed should take over tasks of teaching assistants in times when no one is available for help. In in-class programming tutorial sessions, teaching assistants usually deepen the learning contents by explaining the exercises (T1) and the theoretical background (e.g., underlying algorithms; T2). Even though lectures are usually not suited to answer all questions of all students, the teaching assistants often have enough capacity to answer remaining open questions about the learning contents (T3). To help novice programmers in-class, the teaching assistants usually instruct them based on the students' individual needs and experiences (T4). After the students finished the tasks, the teaching assistants need to correct them (T5) and give the students individual formative feedback (T6).

Additionally, we rely on the ICAP framework as a kernel theory as outlined in the *Theoretical Background* section and supplement it with further educational literature. Based on the framework, we derived the overall solution approach to design a chat-based learning system for targeting the problem of insufficient support in introductory programming courses. Based on the assumption of the ICAP framework that an increased interactivity results in enhanced learning effectiveness, the first requirement (R1) for our artifact solution is to provide a natural language interface, which can be used by the students to interact with a chatbot. In this setting, the chatbot takes over the tasks of the instructors (teaching assistants) who are capable of explaining the tasks and learning content. Due to this form of chat-based interactivity, two types of interaction – the learner-content interaction as well as the learner-instructor – can be enhanced, which is advantageous according to Moore (1993). Using the natural language user interface of chatbots, particularly the answering (T3), instructing (T4) and feedback (T6) tasks can be realized.

Additionally, a source code editor that is integrated into the learning system is required (R2) in order to support the actual coding process (U3). It is thereby important that the learning system including the source code editor is accessible independently of a specific place or time as the main objective of the system is to support students outside of traditional learning settings (e.g., students should be able to use it at home when practicing how to code).

Based on the first two requirements targeting the actual user interface design, six additional function-oriented requirements are needed, which are directly related to the teaching assistants' task characteristics. Thus, the chatbot-based learning system needs to be able to provide explanations of the programming exercises (R3 based on T1 and U1) and of the theoretical background of the exercises (R4 based on T2, U1, and U2), when students need additional advice. Besides this mostly static information provision, teaching assistants have the task to answer on-demand questions (R5 based on T3, U1) and to provide step-by-step guidance to students that require additional input (R6 based on T4 and U3). With these requirements, the essential tasks that are usually carried out by teaching assistants can be addressed. In addition to these in-class tasks, the teaching assistants usually need to correct submitted assignments (R7 based on T5 and T6) after the in-class sessions. Based on this, the chatbot-based system should give the students feedback and additional advice (R8 based on T6 and U3).

A detailed overview of the relationships between the students' user stories (U1 - U3), the teaching assistants' task characteristics (T1 - T6) and the requirements (R1 – R8) is displayed in Figure 2. It also visualizes the connection between the requirements and the design principles that are derived in the next section.

#3 Deriving Design Principles

Based on the eight requirements, we derived five design principles and a conceptual artifact that is the first iteration of our design process (see next subsection). Whereas two design principles are targeted at the user interface, three design principles encompass the functional aspects of the artifact.

The first design principle defines that the main interactivity as derived from the kernel theory should take place in a natural language-based user interface (DP1). To meet this requirement in the learning system, a chatbot is required that is able to automatically communicate with the students. Thus, a chat-based user interface needs to be integrated into a web-based programming environment (DP2). Due to this, the students will be able to interact with the system independently of time and place as required by R2. Close integration of DP1 and DP2 is needed as the requirement elicitation in the previous step indicated that the chat-based communication needs to be adapted based on the written source code of students. Otherwise, it would, for instance, not be possible to provide individualized feedback to students as for this, the system needs to analyze the source code in order to formulate adequate natural language-based communication. As these two design principles target the overall user interface design, they need to be considered when implementing the following function-oriented design principles.

As the main function-oriented design principle, a learning path is required for each programming exercise that should be provided (DP3). This enables that students can be guided step-by-step through solving the task. As a major aspect for enabling interactivity based on the individual needs of the students, the learning paths need to be adaptive to the students' state of knowledge, i.e. their programming skills. From a didactical point of view, adaptive learning paths seem particularly desirable as they pledge an enhanced teaching efficiency for the students. Due to adaptive learning paths, more experienced students do not have to deal with learning contents targeted at beginners, but they can interact with more advanced learning content. Thus, the individual skills of the students can be promoted, which could not be easily achieved by human tutors.

As it is usually not feasible that learning paths cover all theoretically possible topics of a whole lecture, an additional question and answering component is required that can answer open questions concerning the learning contents (DP4). By implementing such a component, the requirements R4 and R5 can be met. As a basis for the generation of answers, a learning object database is required that acts as the underlying knowledge base. To be effective, the learning system should be able to answer all exercise related questions, which is only possible if the knowledge base is large enough to cover all related topics of the programming exercise and even contains learning objects that go beyond this.

Finally, automatic feedback should be enabled (DP5), which is the basis for correcting the students' assignments automatically and for providing formative feedback which helps to improve their performance

(Rietsche et al. 2018). The concepts of static and dynamic testing known from common software testing approaches seem particularly important in order to provide as individualized feedback as possible.

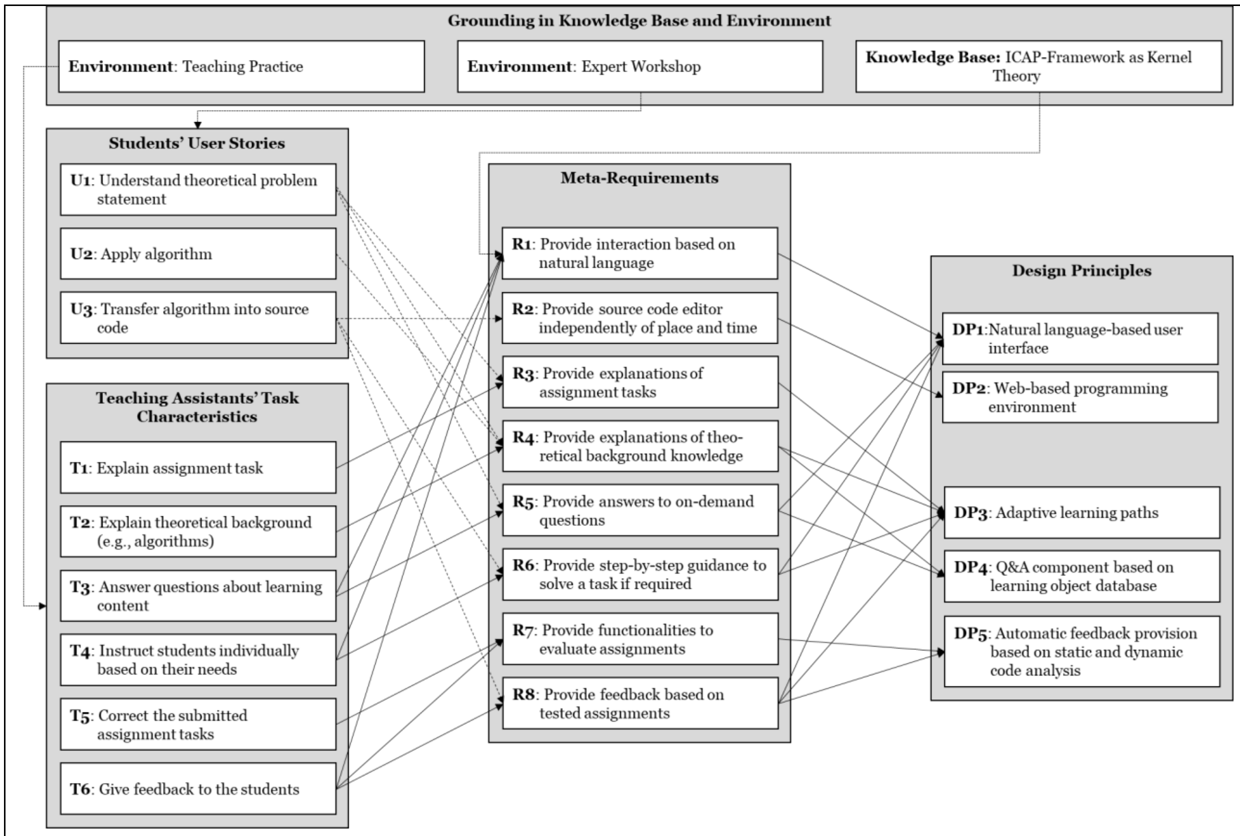

Grounding in Knowledge Base and Environment
Environment: Teaching Practice
Environment: Expert Workshop
Knowledge Base: ICAP-Framework as Kernel Theory
Students' User Stories
U1: Understand theoretical problem statement
U2: Apply algorithm
U3: Transfer algorithm into source code
Teaching Assistants' Task Characteristics
T1: Explain assignment task
T2: Explain theoretical background (e.g., algorithms)
T3: Answer questions about learning content
T4: Instruct students individually based on their needs
T5: Correct the submitted assignment tasks
T6: Give feedback to the students
Meta-Requirements
R1: Provide interaction based on natural language
R2: Provide source code editor independently of place and time
R3: Provide explanations of assignment tasks
R4: Provide explanations of theoretical background knowledge
R5: Provide answers to on-demand questions
R6: Provide step-by-step guidance to solve a task if required
R7: Provide functionalities to evaluate assignments
R8: Provide feedback based on tested assignments
Design Principles
DP1: Natural language-based user interface
DP2: Web-based programming environment
DP3: Adaptive learning paths
DP4: Q&A component based on learning object database
DP5: Automatic feedback provision based on static and dynamic code analysis

Figure 2. Overview of Elicitation of Requirements (step 3) and Design Principles (step 4)

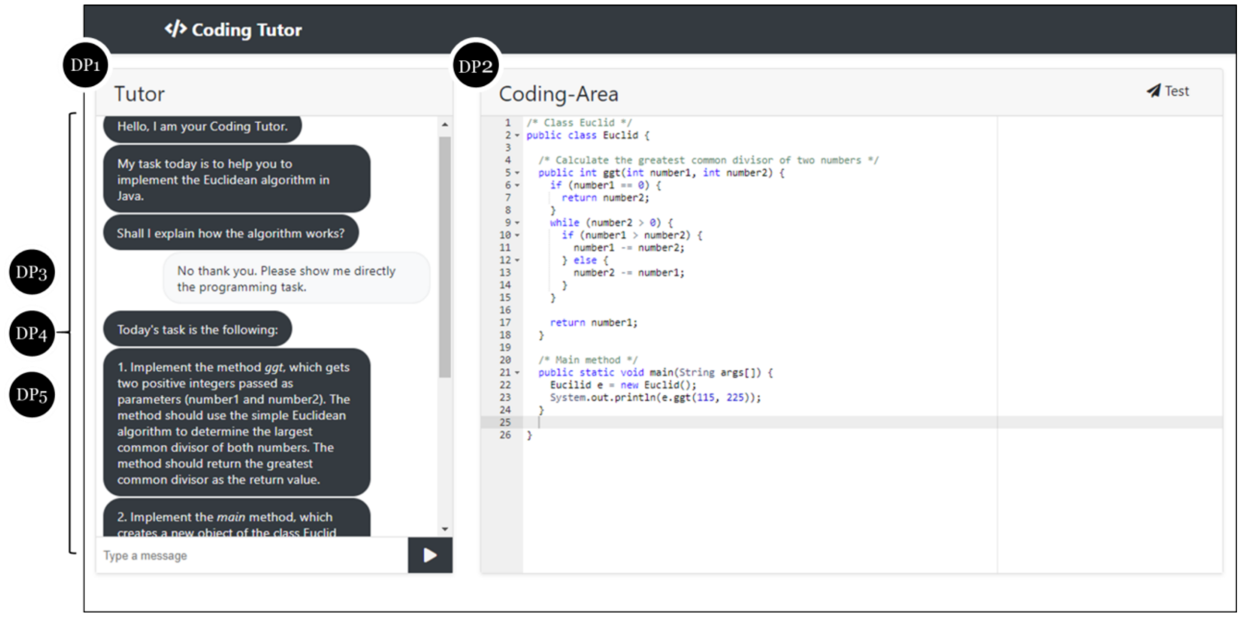
#4 First Iteration: Designing the Conceptual Artifact

Based on the derived design principles, we implemented a conceptual artifact called *Coding Tutor* in the first design iteration. To this aim, we implemented first a clickable mockup as a web-based front-end that visualizes how the user interface (DP1 and DP2) can be designed. Second, we designed an exemplary learning path that visualizes possible usage flows of students in the learning system to address the remaining function-oriented design principles.

We developed the clickable mockup front-end using state-of-the-art web technology (HTML5, CSS3, and JavaScript) and implemented a simple – but predefined – chat dialog that enables us to visualize the interactivity. To provide a ubiquitously accessible front-end, we used the Bootstrap 4 framework that enables a responsive layout. Due to this, the front-end can be accessed on any computer with varying screen sizes and equipped with state-of-the-art browsers. Even though the user interface is designed to be responsive to various screen sizes, using it on smartphones seems not suited to us due to the usual complexity of programming exercises. As displayed in Figure 3, the front-end is structured in two main parts. On the left side, the natural language interface (DP1) is shown as a chat as known from common instant messengers. The programming environment (DP2) is displayed on the right side and occupies the largest space of the front-end as this is the most important user interface component where the actual programming task takes place. The programming environment is implemented using the well-known *Ace editor* that enables common features like syntax highlighting, automatic intents and line wrapping (Ace 2019).

The interaction of the functional design principles (DP3, DP4, and DP5) takes place within the chat-based component. Both, predefined learning paths and on-demand question and answer dialogs, can be realized.

When a predefined learning path is taught, the interaction is pushed forward by the Coding Tutor as the main intent is to provide explanations or additional hints. For instance, the explanation of the programming exercise is presented, and the students are asked whether they need more learning contents related to the theoretical background (e.g., a suited algorithm that can be used to solve the task). If the students ask the Coding Tutor for such theoretical learning contents, a predefined learning path is visualized (DP3). Besides this, students can always interrupt the current conversation to ask further open questions to request specific explanations and learning contents on-demand (DP4). Additionally, the Coding Tutor is always capable of analyzing the students' source code to provide textual feedback in the chat component (DP5).



Hello, I am your Coding Tutor.

My task today is to help you to implement the Euclidean algorithm in Java.

Shall I explain how the algorithm works?

No thank you. Please show me directly the programming task.

Today's task is the following:

1. Implement the method *ggt*, which gets two positive integers passed as parameters (number1 and number2). The method should use the simple Euclidean algorithm to determine the largest common divisor of both numbers. The method should return the greatest common divisor as the return value.

2. Implement the *main* method, which creates a new object of the class Euclid

```
/* Class Euclid */
public class Euclid {

  /* Calculate the greatest common divisor of two numbers */
  public int ggt(int number1, int number2) {
    if (number1 == 0) {
      return number2;
    }
    while (number2 > 0) {
      if (number1 > number2) {
        number1 -= number2;
      } else {
        number2 -= number1;
      }
    }

    return number1;
  }

  /* Main method */
  public static void main(String args[]) {
    Eucilid e = new Euclid();
    System.out.println(e.ggt(115, 225));
  }

}
```

Figure 3. Screenshot of the Clickable Front-end Mockup of Coding Tutor

To visualize the learning process supported by Coding Tutor, we designed an exemplary learning path that shows the most important learning steps and exemplary chat dialog excerpts. The learning path of the first iteration is displayed in Figure 4 on the left side. The displayed transitions indicate that the students can choose different ways through the learning process. In some cases, the transition can be triggered by Coding Tutor automatically, e.g., if errors are identified in a student's source code, Coding Tutor should proactively inform the student and offers assistance to solve them. Based on both, the students' choice and the automatic evaluation of the source code, adaptive learning can be achieved.

#5 Evaluating the Conceptual Artifact

After finishing the first design iteration by designing the user interface of Coding Tutor and outlining an exemplary learning path, we collected feedback about the resulting conceptual artifact. To this aim, we presented the overall concept of providing an interactive chatbot-based learning system as well as a clickable mockup of the system's user interface to the students of an introductory Java programming course in an information systems study program and requested their feedback. In addition to that, we asked teaching assistants who are assisting novice programmers in multiple introductory programming classes as well to get their opinion about the level of support a solution like Coding Tutor can add. In total, we asked 28 students and 16 teaching assistants in this first evaluation to participate and received 22 and 16 valid responses respectively. In this first evaluation, we were interested in the students' and teaching assistants' overall opinion concerning the concept using open-ended free-text questions and – more importantly – we wanted to get suggestions for improvement before we started the actual software implementation.

Overall, the students rated the concept as a useful addition to the teaching concept of introductory programming courses. It would make it easier for them to complete the exercises. For instance, one student evaluated the overall concept as follows: "I think the concept makes a lot of sense, mainly because you don't always find the right answers to your questions on the Internet when problems during solving the exercises arise.” Particularly, the learning path was highlighted by students as “very helpful for explanations and understanding”. Overall, approx. 72% of the students supported the idea of the Coding Tutor concept and only 9% would rather not use it. Especially students that rated their programming skills as high, would rather not use the Coding Tutor: “It wouldn't help me very much, because I can solve the tasks on my own without any problems.”

The students’ comments and suggestions for improvement focused on three aspects: (1) The step-by-step guidance was on the one hand rated as very helpful, but on the other hand some students had concerns whether the guidance would reduce the learning effectiveness when too much guidance is provided: “I am unsure if the step-by-step guidance is useful. After the course, there are no step-by-step instructions available in the reality of a programmer”. Another student added that “with too many explanations, students no longer have to think for themselves”. (2) In addition to that, students had concerns whether the Coding Tutor will be able to understand questions properly and whether it will have answers to all questions. For instance, one student acknowledged: “Brilliant idea, with the problem of a sufficient knowledge base”. (3) Finally, the students requested that the Coding Tutor should not only indicate when the source code was faulty, but it should also explain how to solve it and provide the original error message from the compiler. This would help in the long-term, to solve programming errors on their own: “In addition to the explanations of the artificial tutor, the errors of the compiler should also be displayed.”

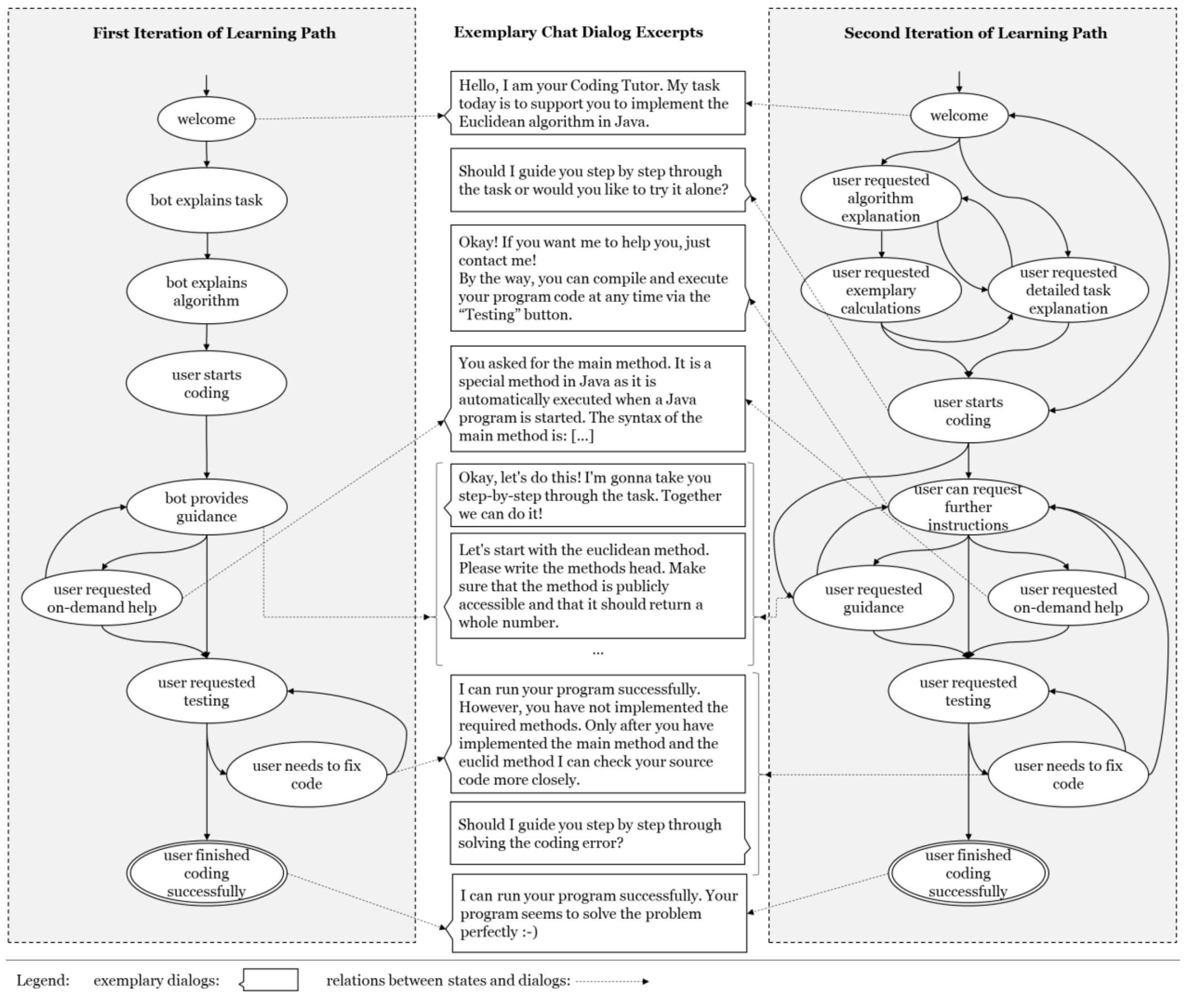
The teaching assistants rated the overall concept of Coding Tutor similarly. Remarkable is that no teaching assistant evaluated the concept negatively. The teaching assistants also shared concerns regarding too much guidance as it “could inhibit a student's ability to seek individual solutions”. One assistant added that “it's important that beginners don't just only learn a programming language but learn that programming sometimes means long troubleshooting”. As a solution, one assistant suggested that instead of detailed step-by-step instructions that are strictly related to the solution, “guided instructions should focus on solution strategies”. Another one suggested to “not use the Coding Tutor during the whole semester for all tasks”, but only in the beginning. Also, the answering of open questions was mentioned as “it only makes sense if it works well and is strongly related to the topic.” Thus, the teaching assistants requested that the Coding Tutor needs to cover all contents of the lecture.

Based on this valuable feedback of the students and teaching assistants, we revised our conceptual artifact and implemented a fully working software artifact as described in the next step.

#6 Second Iteration: Revising the Concept and Implementing the Software Artifact

As outlined in step 5, the participants of our evaluation of the conceptual artifact confirmed that the overall idea as well as the overall concept are suited for the given problem. However, guidance based on learning paths (DP3) was particularly discussed critically. Both, students and teaching assistants, had concerns that too many hints based on guided instructions could possibly negatively affect the students’ ability to code on their own. However, despite this concern of providing too much help, the provision of guidance was not questioned in general. Only the level of support should be chosen carefully.

We agreed with the opinion of the students and teaching assistants and revised the design principle DP3 accordingly. Using adaptive learning paths based on the concept of scaffolding (Kim and Hannafin 2011; van de Pol et al. 2010) could solve the problem of providing too much guidance to students. As proposed in the scaffolding concept, the provision of learning support should be adapted to meet the learners’ level of knowledge. Hence, the learners should be encouraged and supported to solve tasks or problems they wouldn’t be able to solve on their own without proper support. Scaffolding is thus transferable to the problem of solving programming exercises at hand: If a student is a novice programmer and cannot solve an exercises on his/her own, the Coding Tutor should provide detailed step-by-step instructions to guide the student. When the student’s ability to code evolves during the semester term, Coding Tutor should reduce the guidance to a minimum to encourage the students to solve the exercises on their own. In contrast to that, for a student who already has some prior knowledge about programming, the guidance should be reduced to a minimum from the beginning. Instead, Coding Tutor should only provide assistance if a student is not able to solve the task on his/her own. By implementing this guidance behavior, the students’ and teaching assistants’ concerns can be addressed properly by adaptive learning paths based on scaffolding. To take this into account, we have revised our learning concept by adding further transitions, branches, and states, as shown in Figure 4. Thus, the guidance can be adapted based on the students’ needs.


First Iteration of Learning Path
welcome
bot explains task
bot explains algorithm
user starts coding
bot provides guidance
user requested on-demand help
user requested testing
user needs to fix code
user finished coding successfully
Exemplary Chat Dialog Excerpts
Second Iteration of Learning Path
welcome
user requested algorithm explanation
user requested exemplary calculations
user requested detailed task explanation
user starts coding
user can request further instructions
user requested guidance
user requested on-demand help
user requested testing
user needs to fix code
user finished coding successfully
Legend: exemplary dialogs:
relations between states and dialogs:

Hello, I am your Coding Tutor. My task today is to support you to implement the Euclidean algorithm in Java.

Should I guide you step by step through the task or would you like to try it alone?

Okay! If you want me to help you, just contact me!
By the way, you can compile and execute your program code at any time via the "Testing" button.

You asked for the main method. It is a special method in Java as it is automatically executed when a Java program is started. The syntax of the main method is: [...]

Okay, let's do this! I'm gonna take you step-by-step through the task. Together we can do it!

Let's start with the euclidean method. Please write the methods head. Make sure that the method is publicly accessible and that it should return a whole number.

...

I can run your program successfully. However, you have not implemented the required methods. Only after you have implemented the main method and the euclid method I can check your source code more closely.

Should I guide you step by step through solving the coding error?

I can run your program successfully. Your program seems to solve the problem perfectly :-)

Figure 4. First and Second Iteration of the Learning Path including Exemplary Dialog Excerpts

The remaining two remarks of the evaluation participants, the Coding Tutors' capabilities to answer open, topic-related questions and the provision of detailed error message from the compiler, are valuable for our implementation as well. However, a revision of the design principles proposed in step 3 is not necessary, as both aspects focus on very specific implementation details and are covered by the existing design principles. To address the first aspect in the Coding Tutor prototype, the underlying knowledge base that is used to answer the students' open-ended questions needs to cover all important terms and concepts that are relevant for the programming exercises. Additionally, the natural language understanding component needs to be trained properly that it will recognize the students' questions properly. Thus, this aspect is an additional specification of DP4 (Q&A component based on learning object database). The second aspect – the provision of detailed error messages – is also already covered. DP5 specifies that automatic feedback should be provided to the students. To address the students' concerns, we specify DP5 further such that the compiler's error message should be provided to the students in addition to a natural language explanation.

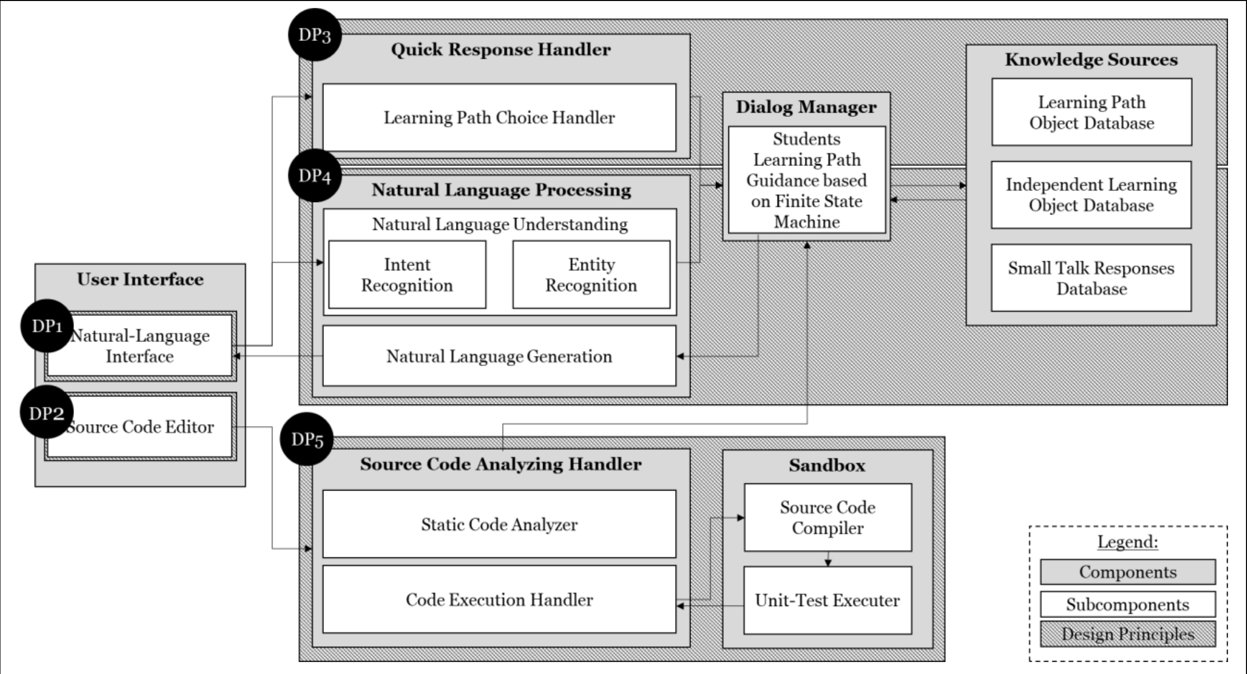

DP3
Quick Response Handler
Learning Path Choice Handler
DP4
Natural Language Processing
Natural Language Understanding
Intent Recognition
Entity Recognition
Natural Language Generation
Dialog Manager
Students Learning Path Guidance based on Finite State Machine
Knowledge Sources
Learning Path Object Database
Independent Learning Object Database
Small Talk Responses Database
User Interface
DP1
Natural-Language Interface
DP2
Source Code Editor
DP5
Source Code Analyzing Handler
Static Code Analyzer
Code Execution Handler
Sandbox
Source Code Compiler
Unit-Test Executer
Legend:
Components
Subcomponents
Design Principles

Figure 5. Software Architecture of the Coding Tutor Artifact

As the clickable front-end mockup as shown in Figure 3 (step 4) was positively evaluated by the students and teaching assistants, we used it as the basis for our subsequent software implementation of Coding Tutor. To integrate the process logic into the front-end, we rely on RESTful web services, i.e. we extended the existing front-end with a JavaScript-based logic to call the needed functionalities via AJAX from our backend infrastructure. As visualized in our software architecture (see Figure 5), in addition to the user interface that implements DP1 and DP2, we implemented three backend components: (1) To implement the adaptive learning paths based on scaffolding in the backend (DP3), we implemented a component that can guide students based on the redefined learning paths through solving the programming task (see Figure 4). The learning paths are implemented in Coding Tutor as finite state machines. Based on the students' abilities to code as well as on their own decisions, our so-called quick response handler is responsible for selecting the appropriate level of support. By relying on the dialog manager and the knowledge sources (particularly the learning path object database), appropriate instructions are provided to the students. (2) To implement DP4 and to enable answering based on open-ended questions, we added a natural language processing component to Coding Tutor based on the open source library NLP.js by AXA Shared Services Spain S.A. (2019). After detecting the student's intent behind a question, an appropriate answer is generated by the dialog manager using the independent learning object database that stores definitions of all relevant terms and concepts that are related to the homework exercises or using the small talk response database if the student's intent was not topically related. We chose to integrate a small talk component as well, as prior research indicates that it could improve the users' acceptance even though it is not complimentary for solving the homework tasks. (3) We implemented the source code analyzing component to meet DP5. To this aim, we integrated both, a static and a dynamic code analyzer, as discussed in step 3. Particularly important for dynamic code testing in *our code execution handler* is to ensure a secure environment that cannot be exploited by the students. We chose to compile and execute the students' source code only after starting a sandbox environment on the server. In this secure environment that we reset after each execution, we execute unit tests based on JUnit, which enables us to properly examine and evaluate the students' assignments. The results generated by the unit tests are particularly important to give the students in-depth feedback automatically and are thus sent via the messenger interface to the students. When testing the programming tasks, we proceed according to the black-box testing method. Therefore, we do not force the implementation of a specific algorithm. Instead, the students can implement their problem solution and our test procedure uses multiple exemplary input values to check whether expected results are calculated by the program. As a result of this second design interaction, we resulted in a fully-functional software artifact that implements all (revised) design principles as specified in step 3 and step 5.

#7 Evaluating the Software Artifact

To evaluate the result of the second design cycle, we demonstrated the Coding Tutor artifact to 40 students in an introductory programming course. According to a self-assessment before the evaluation, students participating in the course evaluated their programming experience as low (approx. 50 %). Only 10 % rated their skills as high. This self-evaluation is in line with our expectation as the course is targeted at information systems students from the 3rd semester onwards. During the evaluation study, the participants had the possibility to use the Coding Tutor software to solve a predefined programming exercise. In this task, the students should code a program that can calculate the greatest common divisor of two specified numbers using the Euclidean algorithm. To give the participants enough time and space to solve the task and to test our software artifact carefully, we didn't restrict the time duration for the test scenario. On average, the students interacted with Coding Tutor for approx. 20 minutes. Whereas the more experienced students only needed approx. 14 minutes to solve the task completely, the last student successfully finished it after 32 minutes. Although participation in the evaluation was voluntary and successful completion of the programming task was not enforced, a majority successfully solved the task completely. After completing the test, we asked all participants to fill out a questionnaire in which we aimed at evaluating our design principles. To measure the participants' perception of our developed software artifact, we provided a quantitative questionnaire based on a 5-point Likert scale (-2: strongly disagree to +2 strongly agree) in which we focused on functional (i.e., usefulness of the implemented design principles) and form aspects (i.e., ease of use of the implemented design principles) of each design principle by adapting the well-known constructs perceived usefulness and perceived ease of use (Davis et al. 1989). Furthermore, we asked the students if they consider Coding Tutor as suitable for practical use in introductory programming courses and if they intent to use it in the future (Davis et al. 1989). In addition to these quantitative items, the students had the possibility to provide written feedback, e.g., about required improvements and further remarks. At the end of our evaluation, we received 35 valid questionnaires, which are the basis for our analysis. We analyzed each design principle individually as shown in Figure 6 and tested whether the data significantly differs from the mean using one-sided t-tests.

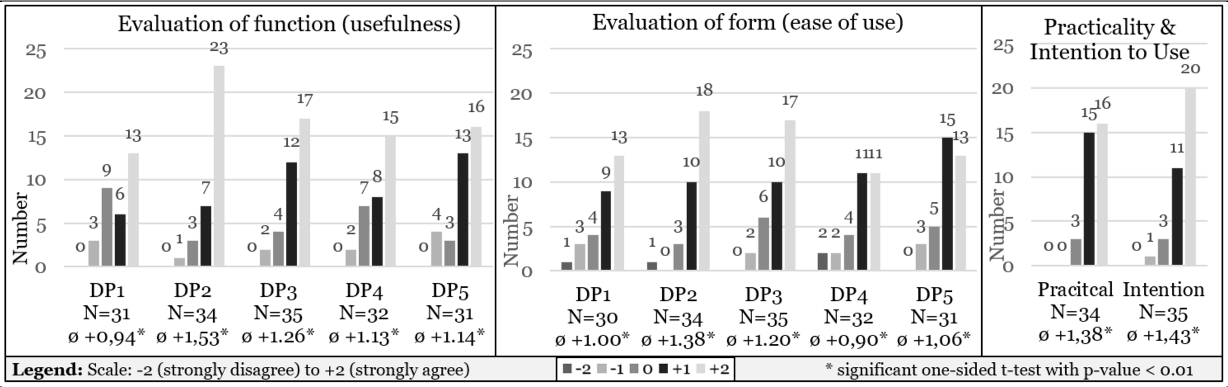


Figure 6. Overview of the evaluation results of function and form

The results show that all design principles were evaluated by the students as useful and easy to use. The averages of all constructs are significantly better than the Likert scale's mean value and no student rated the usefulness of any design principle with -2. Particularly noticeable is the fact that the students rated the usefulness of DP3 (adaptive learning path), DP4 (Q&A component), and DP5 (automatic feedback provision) with average values of +1.26, +1.13 and +1.14 very positively. This indicates that the Coding Tutor artifact is capable of performing tutoring tasks that are usually carried out by teaching assistants. This is also reflected in the fact that the students rated the software as useable in practice (practicality) with an average of +1.38 and intent to use it in the future (mean of +1.43).

In the written feedback, the students particularly emphasized the usefulness of the step-by-step guidance and rated the "proactive offering of step-by-step guidance" at the beginning of the programming task as well as in situations when the static code analysis indicates errors in the source code as "very helpful". Additionally, several students named the "interactivity of the chatbot as a significant improvement" during programming tasks when additional help is required to solve a task. Besides these positive remarks, two aspects for improvements of Coding Tutor were mentioned: (1) Some students suggested improving the responsiveness of Coding Tutor, especially when providing feedback about the source code. In particular, the execution and dynamic code testing using unit tests took some time in some situations. For instance, during the evaluation study, a large number of students tried to execute and test their source code almost at the same time. At this moment, the students needed to wait some seconds before Coding Tutor could provide detailed feedback. As the duration for executing and evaluating the students' source code is mainly dependent on Coding Tutor's server computing power, it can easily be improved by using a more powerful server. During the field test, we only used a virtual server with low computing power and restricted memory. By increasing the number of CPU cores of our virtual server after the test, we were able to significantly increase Coding Tutors responsiveness and to solve the issue. (2) As a second possibility for improvements, expending the knowledge base of the chatbot was named. This became apparent as two students tried to figure out which questions could be answered by Coding Tutor and which could not. In our test setting, the knowledge base was restricted to cover programming aspects related to the provided task. Thus, it was easy for the students to identify the knowledge base's boundaries. For later use in introductory programming courses, it is necessary to provide a sufficient knowledge base. Overall both mentioned possibilities for improvements didn't question our design principles. However, both aspects need to be considered when using chatbot-based learning systems in practice. Thus, a sufficient server capacity needs to be arranged to provide Coding Tutor to larger audiences and the learning content should be extended to cover all relevant aspects of the course's content.

In summary, the evaluation of our fully-functional software artifact indicates that our conceptual design and the design principles are valid for using chatbot-based learning systems in introductory programming courses. The students rated both, the form and the function, of our implemented design principles as suited. As the second evaluation cycle didn't reveal any major design issues, we terminated our design process successfully after the second design cycle.

#8 Documenting the Design Knowledge

To document the results of our design process and to communicate our results to the scientific knowledge base, we use the core components of a design theory as proposed by Gregor and Jones (2007). In this way, we summarize our theoretical contributions in form of a "design and action" theory (Gregor and Jones 2007) based on our rigorous design process. Thus, Table 2 summarizes our systematically derived design knowledge using the components *purpose and scope*, *constructs*, *principles of form and function*, *artifact mutability*, *testable proposition* and *justificatory knowledge* (Gregor and Jones 2007).

Component	Description
Purpose and scope	The purpose of the concept and implementation of Coding Tutor is to support novice programmers in introductory programming courses to learn how to code while solving programming tasks.
Constructs	Chatbot, programming environment, learning paths, knowledge base of learning objects, source code analyzer
Principles of form and function	DP1: Support students using natural language in a chatbot-based user interface; DP2: Provide an easily accessible (web-based) programming environment; DP3: Provide guidance based on adaptive learning paths using scaffolding; DP4: Answer topic-related questions based on a sufficient knowledge base; DP5: Provide automatic feedback based on static and dynamic code analysis
Artifact mutability	The artifact can be applied in every introductory programming course independently of a specific programming language by revising the programming tasks, the knowledge base and particularly the learning paths. By replacing the coding environment with other interactive tools (e.g., modelling tools for UML), it can even be used to impart procedural knowledge in other domains of (IS) education (see following discussion section).
Testable propositions	To test the design principles and implementation, each aspect as mentioned above needs to be surveyed. To evaluate the effects on students' learning performance, the following propositions should be considered: (1) Using Coding Tutor increases the students' ability to solve programming exercises. (2) Using Coding Tutor improves the provision of guidance to students. (3) Using Coding Tutor reduces the required amount of teaching

	assistants' resources when supporting students even in times when no lecture or in-class tutorial is available.
Justificatory knowledge	Scientific literature, particularly the ICAP framework (Chi and Wylie 2014) and the scaffolding concept (Kim and Hannafin 2011), as well as empirical knowledge, particularly results from a workshop and tasks characteristics of teaching assistants.

Table 2. Documentation of the Design Knowledge based on Gregor and Jones (2007)

Discussion and Conclusion

The overall research goal of this DSR study was to design a chatbot-based learning system that aims at supporting students to learn to code in university courses. In particular, the aim was to support novice programmers in introductory programming courses in times when no teaching assistant or no in-class support is available (e.g., due to resource constraints). To this aim, we derived requirements from scientific literature (especially the ICAP framework by Chi and Wylie 2014), a focus group workshop and the task characteristics of teaching assistants. Based on these inputs from the knowledge base (*rigor*) and the current teaching practice (*relevance*), we deduced design principles and implemented our software artifact called Coding Tutor in an iterative process by adapting the three cycle design science research approach (Hevner et al. 2004; Hevner 2007). During this design process, we evaluated the results of each design iteration and successfully terminated our design process after two iterations.

Besides the software artifact as a situated implementation of a chatbot-based learning system, we contribute design knowledge to the scientific knowledge base. We systematically deduced design knowledge as documented in our last step of the design process (see Table 2 in step 8). Due to the systematic procedure, we aimed at generating a satisfying design contribution as indicated by Gregory and Muntermann (2014). The resulting design knowledge is not only valid for our specific case but can also be transferred to further use cases in programming education. For instance, it is easily possible to apply the concept of Coding Tutor in courses that deal with other programming languages than Java that we used in our case. To this aim, only the knowledge base of learning paths, learning objectives as well as the programming tasks need to be adapted. The design principles of form and function and the overall system design don't need to be adapted for those use cases. Furthermore, it is also possible to transfer the design knowledge to other use cases that target the training of procedural knowledge in other areas of (IS) education. For instance, if the application of other IS-related software tools like UML-modelling should be trained, a similar chatbot-based learning system can be used. However, in this case, the system design needs to be revised partially. Particularly the coding environment needs to be replaced, e.g., by an UML-modelling tool. Due to this transferability of our design knowledge, our research does not only provide a Level 1 DSR contribution by showing a situated artifact implementation but also provides a nascent design theory (Level 2 contribution) (Gregor and Hevner 2013).

In addition to these contributions to the scientific knowledge base, our results are also valuable for the practical use of chatbot-based learning system in (IS) education. Our results indicate that the state-of-the-art chatbot-technology is suited to design complex chatbot-based learning systems that are able to support students individually – even in complex teaching tasks when procedural knowledge and advanced cognitive processes (like *analyzing*; Anderson and Krathwohl 2001) should be trained. We have thus shown that more advanced support possibilities in programming education can be implemented using chatbot-based learning systems in contrast to common e-learning approaches that are currently used in university education.

Despite these contributions to the scientific knowledge base and the practical application of chatbot-based learning systems, limitations should be considered. For the aim of this study, we focused our research scope on introductory programming courses. Even though it is reasonable to assume that the transferability to other cases is possible without major changes, we cannot prove it with our research design. Additionally, we focused our research on generating design knowledge by applying a DSR approach. Due to this, we focused on deducing and evaluating design principles and assessing the students' evaluation. For future research, particularly analyzing the long-term effect of using Coding Tutor or a similar chatbot-based learning system in IS education pledges interesting insights into the effectiveness and long-term acceptance of chatbot-based learning systems. In doing so, not only the effects of these technology-enhanced learning systems on the learning behavior of students can be analyzed, but also (necessary) changes in the teaching concept can be surveyed. Additionally, potential downsides of using chatbot-based learning systems should be analyzed. Even though we do not plan to replace human teaching assistants in our learning setting but provide students with our Coding Tutor as an additional learning opportunity, others might think of doing this. Obviously, experienced human teaching assistants might be able to assist students who have difficulties with solving programming tasks better than our Coding Tutor. Nevertheless, introducing a chatbot-based learning system might still be beneficial in those cases as the workload of human teaching assistants might be reduced as the Coding Tutor system might be able to answer most questions automatically. Human teaching assistants are then able to focus on answering the remaining, more difficult questions.

References

Ace 2019. *Ace - The High Performance Code Editor for the Web*. https://ace.c9.io/. Accessed 29 March 2019.

Albrecht, E., Gumz, F., and Grabowski, J. 2018. "Experiences in Introducing Blended Learning in an Introductory Programming Course," in *Proceedings of the 3rd European Conference Software Engineering Education*, J. Mottok (ed.), New York: ACM, pp. 93–101.

Anderson, L. W., and Krathwohl, D. R. (eds.) 2001. *A taxonomy for learning, teaching, and assessing*: *A revision of Bloom's taxonomy of educational objectives*, New York: Longman.

AXA Shared Services Spain S.A. 2019. *axa-group/nlp.js*. https://github.com/axa-group/nlp.js. Accessed 4 April 2019.

Bey, A., Jermann, P., and Dillenbourg, P. 2018. "A Comparison between Two Automatic Assessment Approaches for Programming: An Empirical Study on MOOCs," *Journal of Educational Technology & Society* (21:2), pp. 259–272.

Bloom, B. S., Engelhart, M. B., Furst, E. J., Hill, W. H., and Krathwohl, D. R. 1956. *Taxonomy of educational objectives. The classification of educational goals (Handbook I. Cognitive Domain)*, New York: Longman.

Breitner, J., Hecker, M., and Snelting, G. 2017. "Der Grader Praktomat," in *Automatisierte Bewertung in der Programmierausbildung*, O. J. Bott, P. Fricke, U. Priss and M. Striewe (eds.): Waxmann Verlag GmbH, pp. 159–172.

Chi, M. T. H., and Wylie, R. 2014. "The ICAP Framework: Linking Cognitive Engagement to Active Learning Outcomes," *Educational Psychologist* (49:4), pp. 219–243.

Crow, T., Luxton-Reilly, A., and Wuensche, B. 2018. "Intelligent tutoring systems for programming education: a systematic review," in *Proceedings of the 20th Australasian Computing Education Conference,* Brisbane, pp. 53–62.

Daradoumis, T., Marquès Puig, J. M., Arguedas, M., and Calvet Liñan, L. 2019. "Analyzing students' perceptions to improve the design of an automated assessment tool in online distributed programming," *Computers & Education* (128), pp. 159–170.

Davis, F. D., Bagozzi, R. P., and Warshaw, P. R. 1989. "User Acceptance of Computer Technology: A Comparison of Two Theoretical Models," *Management Science* (35:8), pp. 982–1003.

Goedicke, M., Striewe, M., and Balz, M. 2018. *Computer Aided Assessments and Programming Exercises with JACK:* DuEPublico: Duisburg-Essen Publications Online, University of Duisburg-Essen, Germany.

Graesser, A. C., Cai, Z., Morgan, B., and Wang, L. 2017. "Assessment with computer agents that engage in conversational dialogues and trialogues with learners," *Computers in Human Behavior* (76), pp. 607–616.

Gregor, S., and Hevner, A. R. 2013. "Positioning and presenting design science research for maximum impact," *MIS Quarterly* (37:2), pp. 337–356.

Gregor, S., and Jones, D. 2007. "The Anatomy of a Design Theory," *Journal of the Association for Information Systems* (8:5), pp. 312–335.

Gregory, R. W., and Muntermann, J. 2014. "Research Note —Heuristic Theorizing: Proactively Generating Design Theories," *Information Systems Research* (25:3), pp. 639–653.

Hevner, A. 2007. "A Three Cycle View of Design Science Research," *Scandinavian Journal of Information Systems* (19:2), pp. 87–92.

Hevner, A., March, S., Park, J., and Ram, S. 2004. "Design Science in Information Systems Research," *Management Information Systems Quarterly* (28:1), pp. 75–105.

Hien, H. T., Cuong, P.-N., Le Nam, N. H., Nhung, Ho Le Thi Kim, and Le Thang, D. 2018. "Intelligent Assistants in Higher-Education Environments: The FIT-EBot, a Chatbot for Administrative and Learning Support," in *Proceedings of the Ninth International Symposium on Information and Communication Technology*, New York: ACM, pp. 69–76.

Hobert, S., and Meyer von Wolff, R. 2019. "Say Hello to Your New Automated Tutor – A Structured Literature Review on Pedagogical Conversational Agents," in *14th International Conference on Wirtschaftsinformatik*, pp. 301–314.

Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. 2010. "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10*, C. Schulte and J. Suhonen (eds.), New York: ACM Press, pp. 86–93.

Kim, M. C., and Hannafin, M. J. 2011. "Scaffolding problem solving in technology-enhanced learning environments (TELEs): Bridging research and theory with practice," *Computers & Education* (56:2), pp. 403–417.

Mikic, F. A., Burguillo, J. C., Llamas, M., Rodriguez, D. A., and Rodriguez, E. 2009. "CHARLIE: An AIML-based chatterbot which works as an interface among INES and humans," in *2009 EAEEIE Annual Conference*. 2009, pp. 1–6.

Moore, M. G. 1993. "Three types of interaction," in *Distance Education: New Perspectives*, K. Harry, M. John and D. Keegan (eds.), London: Routledge, pp. 19–24.

Müller, S., Bergande, B., and Brune, P. 2018. "Robot Tutoring," in *Proceedings of the 3rd European Conference Software Engineering Education*, J. Mottok (ed.), New York: ACM, pp. 45–49.

OECD 2018. *Education at a Glance 2018*: *OECD Indicators*, Paris: OECD Publishing.

Passier, H. 2017. "The role of Procedural Guidance in Software Engineering Education," in *Proceedings of the International Conference on the Art, Science, and Engineering of Programming - Programming '17*, J. B. Sartor, T. D'Hondt and W. de Meuter (eds.), New York: ACM Press, pp. 1–2.

Popat, S., and Starkey, L. 2019. "Learning to code or coding to learn? A systematic review," *Computers & Education* (128), pp. 365–376.

Rietsche, R., Duss, K., Persch, J. M., and Söllner, M. 2018. "Design and Evaluation of an IT-based Formative Feedback Tool to Forster Student Performance," in *ICIS 2018 Proceedings*, pp. 1–17.

Sim, T. Y., and Lau, S. L. 2018. "Online Tools to Support Novice Programming: A Systematic Review," in *2018 IEEE Conference on e-Learning, e-Management and e-Services (IC3e)*, IEEE, pp. 91–96.

Tamayo-Moreno, S., and Perez-Marin, D. 2016. "Adapting the design and the use methodology of a pedagogical conversational agent of secondary education to childhood education," in *2016 International Symposium on Computers in Education (SIIE)*, Piscataway, NJ: IEEE, pp. 1–6.

Tegos, S., Demetriadis, S., and Karakostas, A. 2011. "MentorChat: Introducing a Configurable Conversational Agent as a Tool for Adaptive Online Collaboration Support," in *2011 15th Panhellenic Conference on Informatics*, pp. 13–17.

van de Pol, J., Volman, M., and Beishuizen, J. 2010. "Scaffolding in Teacher–Student Interaction: A Decade of Research," *Educational Psychology Review* (22:3), pp. 271–296.

Venable, J., Pries-Heje, J., and Baskerville, R. 2016. "FEDS: a Framework for Evaluation in Design Science Research," *European Journal of Information Systems* (25:1), pp. 77–89.

Vial, G., and Negoita, B. 2018. "Teaching Programming to Non-Programmers: The Case of Python and Jupyter Notebooks," *ICIS 2018 Proceedings*, pp. 1–17.

Wallace, R. S. 2009. "The Anatomy of A.L.I.C.E," in *Parsing the Turing Test*: *Philosophical and Methodological Issues in the Quest for the Thinking Computer*, R. Epstein, G. Beber and G. Roberts (eds.), Dordrecht: Springer Netherlands, pp. 181–210.

Weizenbaum, J. 1966. "ELIZA - a computer program for the study of natural language communication between man and machine," *Communications of the ACM* (9:1), pp. 36–45.

Winkler, R., and Söllner, M. 2018. "Unleashing the Potential of Chatbots in Education: A State-Of-The-Art Analysis," in *Academy of Management Annual Meeting (AOM)*, Chicago, USA.

Zeller, A. 2000. "Making students read and review code," in *Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSEconference on Innovation and technology in computer science education - ITiCSE '00*, J. Tarhio, S. Fincher and D. Joyce (eds.), New York: ACM Press, pp. 89–92.