Clean Code Tutoring: Makings of a Foundation

Nikola Luburić[a], Dragan Vidaković[b], Jelena Slivka[c], Simona Prokić[d], Katarina-Glorija Grujić[e], Aleksandar Kovačević[f] and Goran Sladić[g]
Faculty of Technical Sciences, University of Novi Sad, Trg Dositeja Obradovića 6, Novi Sad, Serbia

Keywords: Intelligent Tutoring Systems, e-Learning, Clean Code, Refactoring, Code Readability, Software Engineering.

Abstract: High-quality code enables sustainable software development, which is a prerequisite of a healthy digital society. To train software engineers to write higher-quality code, we developed an intelligent tutoring system (ITS) grounded in recent advances in ITS design. Its hallmark feature is the refactoring challenge subsystem, which enables engineers to develop procedural knowledge for analyzing code quality and improving it through refactoring. We conducted a focus group discussion with five working software engineers to get feedback for our system. We further conducted a controlled experiment with 51 software engineering learners, where we compared learning outcomes from using our ITS with educational pages offered by a learning management system. We examined the correctness of knowledge, level of knowledge retention after one week, and the learners' perceived engagement. We found no statistically significant difference between the two groups, establishing that our system does not lead to worse learning outcomes. Additionally, instructors can analyze challenge submissions to identify common incorrect coding patterns and unexpected correct solutions to improve the challenges and related hints. We discuss how our instructors benefited from the challenge subsystem, shed light on the need for a specialized ITS design grounded in contemporary theory, and examine the broader educational potential.

1 INTRODUCTION

Software code is written to answer specific functional requirements and enable use cases required of the complete software solution. These requirements state *what* the code must do and do not care for *how* it is designed. Consequently, a requirement can be fulfilled by a near-infinite set of different code configurations. While many code solutions can fulfill a requirement, not all of them are acceptable. Many of the possible solutions cause severe but non-obvious problems. Code that is hard to understand and modify harms the software's maintainability, evolvability, and reliability (Sharma and Spinellis, 2018), introducing technical debt.

On the other hand, *clean code* is easy to understand and maintain, imposing minor cognitive strain on the programmer (Fowler, 2018), thereby increasing their productivity and reducing the chance of introducing bugs (Tom et al., 2013). Such code is a prerequisite for sustainable software development.

Unfortunately, there is ambiguity regarding what constitutes clean code. While functional requirements are easy to test, a code's cleanliness is challenging to evaluate, and software engineers disagree on what code is clean (Hozano et al., 2018) based on their awareness, knowledge, and familiarity with the domain and coding style (Luburić et al., 2021a).

The significance of clean code and the challenges concerning its development produce a need for effective and scalable training of software engineers and their procedural knowledge required to analyze

[a] https://orcid.org/0000-0002-2436-7881
[b] https://orcid.org/0000-0003-3983-7249
[c] https://orcid.org/0000-0003-0351-1183
[d] https://orcid.org/0000-0002-2852-9219
[e] https://orcid.org/0000-0003-2816-7980
[f] https://orcid.org/0000-0002-8342-9333
[g] https://orcid.org/0000-0002-0691-7392

Luburić, N., Vidaković, D., Slivka, J., Prokić, S., Grujić, K., Kovačević, A. and Sladić, G.
Clean Code Tutoring: Makings of a Foundation.
DOI: 10.5220/0010800900003182
In *Proceedings of the 14th International Conference on Computer Supported Education (CSEDU 2022) - Volume 1*, pages 137-148
ISBN: 978-989-758-562-3; ISSN: 2184-5026

the code's cleanliness and enhance it through refactoring (Fowler, 2018). Luckily, e-learning and intelligent tutoring system (ITS) advances can help solve this problem by facilitating effective learning at scale.

An ITS helps educators deliver an efficient and effective learning experience by tailoring instruction to a specific learner and their interaction with the subject matter. At their best, they facilitate *robust learning* for each learner – learning which achieves (1) high knowledge retention, (2) is generalizable, and (3) accelerates future learning opportunities (Koedinger et al., 2012). They discover hidden concepts in the domain knowledge (Piech et al., 2015), identify ineffective instructional principles (Aleven et al., 2016b; Gervet et al., 2020), and reduce the learner's over- and under-practice (Huang et al., 2021). ITSs transform aspects of the educator's work, freeing them from being a database of facts or a hint machine.

Our Clean CaDET[8] ITS is based on contemporary learning theory for ITS design (Shute & Towle, 2003; Koedinger et al., 2012; Aleven et al., 2016a, Huang et al., 2021) and specializes in the clean code and refactoring domain (Fowler, 2018).

In Section 2, we examine the advances in education theories and ITS design that found widespread success in math, languages, and basic programming (Koedinger et al., 2012; Aleven et al., 2016a) and explore how existing tutoring systems targeting clean code overlook this body of literature.

While our Clean CaDET ITS presents traditional lectures to learners, combining learning objects and delivering a tailored lecture with text, video, and multiple-choice questions, we do not bring any novelty to this area of ITS development. Instead, we focus on our novel refactoring challenges subsystem and describe how it builds on contemporary ITS advances in Section 3.

To evaluate the usefulness of our ITS, we organized a focus group discussion with five software engineers, who evaluated our tool and gave feedback for its improvement. We then conducted a controlled experiment with 51 software engineering learners to evaluate the initial version of our ITS. We found that the Clean CaDET ITS produces satisfactory learning outcomes. Notably, the challenge subsystem, this paper's focal point, received high praise from working engineers and learners. We describe our empirical evaluations and the experiment's design and results in Section 4.

We see significant advances in the field of ITSs and the disciplines that make up its foundation, such as cognitive theory, educational psychology, and computational modeling (Koedinger et al., 2012). We discuss how these advances benefited our ITS, the limitations of our design, and call for a more systematic foundation for specialized ITS design in Section 5.

Finally, in Section 6, we conclude the paper and note ideas for further work.

2 BACKGROUND

In Section 2.1, we explore the background for our work. Here we examine the terminology and components of contemporary ITSs, which provide the foundation for our ITS and challenge subsystem. Section 2.2 examines related refactoring educational tools. Here we discuss their strong points, limitations, and underlying learning theory.

2.1 Intelligent Tutoring System Foundations

An ITS is a computerized learning environment that models learning, implements principles of efficient instruction, and contains intelligent algorithms that adapt instruction to the specificities of the learner (Graesser et al., 2012). An ITS aims to develop robust learning outcomes (Koedinger et al., 2012) for each learner by adapting the instruction to their individual needs (Shute & Towle, 2003).

Aleven et al. (2016a) differentiate three levels of instruction adaptivity. *Step-loop adaptivity* is common in ITSs and entails data-driven decisions the ITS makes in response to a learner's actions within an instructional task. An ITS exhibits step-loop adaptivity when it offers hints to a learner that is working on an exercise. *Task-loop adaptivity* includes decisions the ITSs make to select which instructional task the learner should view next. Finally, *design-loop adaptivity* entails decisions made by course designers to improve the ITS or course content based on data collected by the ITS or the environment in which it is used. To enable all levels of adaptivity, the ITS collects an extensive set of data to support learning analytics (Siemens, 2013). The captured data

[8] Clean CaDET (Prokić et al., 2021) is our platform for clean code analysis, which includes the ITS. It is found at https://github.com/Clean-CaDET/platform#readme.

can include time spent in the learning environment, clickstreams, task submissions, and other behavioral data. By tracking behavior, the instructors identify common misconceptions, overly challenging tasks, and ineffective content (Siemens, 2013; Holstein et al., 2017). In this paper, we call this analytics system the *progress model*.

Adaptive e-learning systems such as ITSs are often conceptualized as consisting of three components: the content model, the learner model, and the instructional model (Shute & Towle, 2003; Imhof et al., 2020).

The *content model* represents the domain knowledge and skills covered by the ITS and the course it serves. The model's structure is designed to support adapting to the learner's needs (Shute & Towle, 2003). A notable problem is determining the grain size of the educational content (Shute & Towle, 2003; Koedinger et al., 2012), where step-loop and task-loop adaptivity requires fine-grained content (Aleven et al., 2016a). The IEEE Computer Society (2020) standardized the metadata schema for learning objects, often used as the smallest unit of educational content in adaptive e-learning systems (Shute & Towle, 2003; Imhof et al., 2020). Examples of learning objects include instructional multimedia (e.g., a short text or video that introduces new or refines existing knowledge), assessments (e.g., tests or tasks that evaluate a learner's knowledge), or more complex structures (e.g., case studies or simulations). Koedinger et al. (2012) provide a different view of learning objects in their Knowledge-Learning-Instructional (KLI) framework. They define *instructional events* (e.g., image, text, example) and *assessment events* (e.g., test, task) as observable events in the learning environment controlled by an instructor or ITS. Considering these perspectives, we can say that the ITS selects which instructional or assessment events to trigger (Koedinger et al., 2012) or learning objects to present (Shute & Towle, 2003) to best fulfill the learner's needs.

The *learner model* contains information about the learner that effectively supports the adaptivity of the ITS. The scope of this information varies in the literature. It can include the learner's assessed knowledge (i.e., domain-dependent information), general cognitive abilities, personality traits, and emotional state (i.e., domain-independent information) (Shute & Towle, 2003; Aleven et al., 2016a; Normadhi et al., 2019). Regarding the learner's assessed knowledge, Koedinger et al. (2012) define *knowledge components* as an acquired unit of cognitive function that can be inferred from a learner's performance on a set of related tasks. Aleven et al. (2016a) explored how instructional design adapts to: the learner's prior knowledge, the learner's path through a problem (i.e., their study strategy and common errors), their affective and motivational state, their self-regulation and metacognition efforts, and their learning styles. They found strong evidence that considering prior knowledge or the learner's path through a problem leads to robust learning outcomes. They also found weak evidence that adhering to a learner's learning style affects learning.

The *instructional model* merges the information about the learner with the available content and applies *instructional principles* to offer the correct instructional events or learning objects and achieve robust learning outcomes (Imhof et al., 2020). The instructional principles can be based on general guidelines for delivering effective instruction (Gagne et al., 2005; Koedinger et al., 2012), enhancing the learners' learning processes (Fiorella and Mayer, 2016), or targeting a specific characteristic of the learner, like motivation (Mayer, 2014). The instructional model can be viewed as an expert system, performing the instructor's job by knowing the domain (content model), the learners (learner model), and effective ways to facilitate knowledge development for each learner (instructional principles). This expertise is often implemented using recommender systems (Khanal et al., 2019). A subset of these systems includes black-box machine learning models that might enhance learning but do not offer the necessary insight about learning that can aid instructors in improving the overall ITS and course (i.e., design-loop adaptivity) (Rosé et al., 2019). A different set of recommenders are knowledge-based, where instructors embed their expertise into a rule-based system. Such recommenders are easy to reason about and improve (Rosé et al., 2019). However, they are time-consuming to develop.

2.2 Refactoring Tutors

Wiese et al. (2017) conducted a controlled experiment with 103 students to evaluate AutoStyle, a code style tutor with an automated feedback system. A code's style is a low-level aspect of code cleanliness that often considers single-line changes and simplifications of standalone statements. A basic improvement of code style is illustrated in Figure 1.

```
if(isAccepted == true) {
    // send notification
}
```

```
if(isAccepted) {
    // send notification
}
```

Figure 1: The conditional expression on the left can be simplified, improving the code style as seen on the right.

Wiese et al. clustered similar submissions using historical data from 500 prior submissions. They wrote hints that would move the submission from the current cluster to a cluster with a better coding style. The authors performed an experiment where they randomly divided 103 students into a control group and the AutoStyle group for the experiment, which got hints for improving their submission. They found that the AutoStyle group improved their recognition of code that follows a good style but did not perform better than the control when tasked with writing clean code without the support of AutoStyle.

Keuning et al. (2021) developed Refactor Tutor, an educational tool meant to teach novice programmers to improve their coding style. It provides exercises with structured hints, enabling students to ask for more specific help until they finally reveal the solution. While such hint structures are common in contemporary ITS, they often transform the exercise into a worked example (Aleven et al., 2016b) when the student avoids solving the issue and clicks through the hints. The authors perform focus group discussions with teachers to define rules for evaluating student submissions and hints to help them improve the coding style. They compare the teachers' insight with outputs of code quality analysis tools (e.g., IntelliJ, SonarQube) and conclude that the code quality analysis tools are not suitable as educational tools. In a related paper (Keuning et al., 2020), they use the tool with 133 students to understand how students solve refactoring challenges and perceive their Tutor.

Haendler et al. (2019) developed RefacTutor, a tutoring system for software refactoring. The tool is grounded in Bloom's taxonomy (Haendler and Neumann, 2019) and aims to improve software engineers' procedural knowledge for refactoring at the application, analysis, and evaluation level. The tool introduces novel features, such as UML class visualization of the refactored code. However, the authors did not employ the tool in an educational setting, nor did they evaluate the learning outcomes it produces.

Sandalski et al. (2011) developed a refactoring e-learning environment that plugs into the Eclipse integrated development environment (IDE). From here, the learner submits code for analysis, which runs against rules that determine if the code's cleanliness can be improved. Notably, no evaluation to determine the effectiveness of the learning tool was employed.

The examined educational tools explore exciting ideas like: clustering student submissions to identify trends (Wiese et al., 2017), analyzing hint use and submissions to understand learner behavior during refactoring (Keuning et al., 2021), providing several views on the refactored code (Haendler et al., 2019), and integrating the educational tool into an IDE, enabling learners to utilize the power of their code editor and become more familiar with the toolset they will use in their careers (Sandalski et al., 2011).

Notably, the reviewed work has two sets of limitations worth discussing: a lack of evaluation of learning outcomes and a lack of foundation in contemporary ITS design theory.

Most of the refactoring educational tools did not evaluate the effectiveness of the learning facilitated by the tool. Wiese et al. (2017) present the sole empirical evaluation that tests the learning outcomes of their tool. However, their experiment design presents a threat to validity, as they randomly separated students into a control and experiment group. Consequently, the experiment group using their educational tool might have included more capable students, which could influence their conclusions. This threat could be mitigated by using *randomized block design*, where students are blocked by skill level before being equally distributed to the two groups.

Most of the examined studies do not ground their educational tool in proven ITS design practices. Haendler et al. (2019) present the sole study that uses Bloom's taxonomy, which structures knowledge and cognitive processes, to design their ITS on this foundation. While a good starting point, Bloom's taxonomy primarily classifies learning objectives in a broader context. Advances in education theory and ITS design, such as those presented by Koedinger et al. (2012) and Aleven et al. (2016a), provide more concrete guidance for ITS development. Given the maturity of these contributions, we argue that specialized ITSs should be founded on these advances and develop our ITS accordingly.

3 CLEAN CaDET TUTOR

Our Clean CaDET ITS presents traditional lectures to learners, combining learning objects and delivering a tailored lecture using a basic learner model. It is grounded in the theory described in Section 2.1 and maintains a content, learner, progress, and instructional model. Our novelty lies in utilizing these models to provide refactoring challenges to develop the learners' procedural knowledge for clean code analysis and refactoring. Figure 2 illustrates the interaction flow between the learner and the challenge subsystem of our ITS.

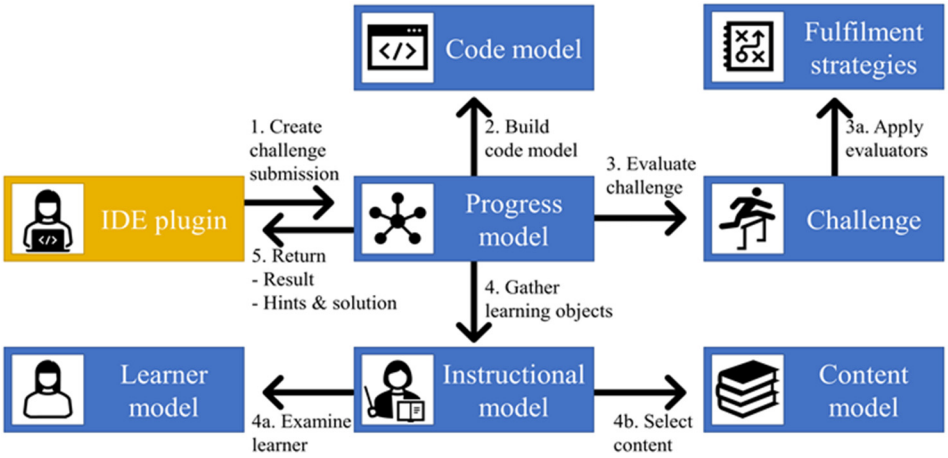

Figure 2: Challenge submission and evaluation data flow.

Once the learner arrives at the challenge learning object, they load the related code (i.e., the starting challenge state) into their IDE[9]. The learner analyses the code, refactors it, and creates a submission through our ITS plugin.

The progress model coordinates the submission processing by:

1. Receiving the challenge submission, including the source code, challenge, and learner identifier.
2. Parsing the code into a model that contains the code graph and calculated source code metrics.
3. Sending the model to the Challenge. The challenge runs available unit tests to check functional correctness. If they pass, it employs evaluation strategies, where:
   a. Each strategy analyses some part of the code graph or metric values to determine if the related clean code criteria is satisfied. A challenge submission is correct if all strategies are satisfied.
4. Asking the instructional model for suitable learning objects for each unsatisfied strategy and the final solution, where:
   a. The learner model is consulted to determine the relevant characteristics of the learner.
   b. The content model is queried for the most suitable available learning objects.
5. Persisting the submission and its correctness and returning the selected set of learning objects.

The learner can make multiple submissions, receiving tailored hints for aspects of their submission that require improvement. The hints are learning objects that present themselves as instructional events (Koedinger et al., 2012). As learning objects, they can be reused among challenges or integrated into a traditional lecture offered by the ITS. As instructional events, they help the learner complete the assessment event (i.e., the challenge). Finally, the challenge solution is a learning object and instructional event that transforms the challenge into a worked example (Aleven et al., 2016b).

Out of the many possible code configurations that fulfill a requirement, we can find a subset of solutions considered clean by most software engineers. Notably, this is different from elementary algebra (a domain for which we see many ITS developments), where problems have a single correct solution. Consequently, the evaluation needs to be flexible to include the acceptable subset of possible solutions. For example, we might define that all methods in the submitted code must have below ten lines of code (mapping to the LOC metric) instead of having precisely five lines.

While fulfillment strategies are flexible, we recommend that learning engineers map one fulfillment strategy to a single knowledge component (Koedinger et al., 2012). Adhering to this recommendation allows educators to examine which knowledge components are under-developed and perform a design-loop adaptation of the content. To better explain this point, we present two challenges, their fulfillment strategies, and the related knowledge components in Table 1.

[9] We made a point to integrate our ITS with industry-relevant IDEs to enable learners to acquaint themselves with refactoring tools supported by the environment

Table 1: Examples of challenges, their fulfillment strategies, and the related knowledge components.

Challenge description	Fulfillment strategies	Related knowledge component
Refactor the class so that all identifiers have meaningful names.	Find expected words or their synonyms at the appropriate place in the code.	Use meaningful words in identifier names
	Detect if any class, method, field, parameter, or variable name contains a meaningless noise word.	Avoid noise words in identifier names
Refactor the class so that all methods are simple, short, and focus on a single, clearly defined task.	Find expected words or their synonyms at the appropriate place in the code.	Use meaningful words in identifier names
	Detect if any method has cyclomatic complexity higher than 10 or maximum nested blocks more than four.	Write simple methods
	Detect if any method has more than 20 unique words in its body (excluding comments and language keywords).	Write methods focused on a single task
	Detect if the containing class has more than 15 methods.	Avoid classes with many micro methods
	Detect if any method has more than 20 lines of code.	Write short methods

We point to two significant elements present in Table 1. First, our challenges assess multiple knowledge components, which is typical for sophisticated tasks involving procedural knowledge. Second, multiple challenges can have the same fulfillment strategy to test the same knowledge component (in the example, the first strategy for both challenges is the same). Testing the same knowledge component through multiple challenges is desirable as multiple assessment events are needed to infer whether a learner has robustly acquired a knowledge component (Koedinger et al., 2012).

4 TUTOR EVALUATION

Here we present our evaluations of the initial version of our ITS. Section 4.1 describes the focus group discussion we held with five working software engineers to review their perceptions on the usefulness of our tool. Next, we performed a controlled experiment with 51 students to ensure that, given the same amount of time, our tool does not lead to worse learning outcomes when compared to a traditional learning management system (LMS). Section 4.2 describes our experiment design, while Section 4.3 presents the achieved results.

4.1 Focus Group Discussion

We conducted the focus group discussion (Templeton, 1994) with five working software engineers of medium seniority (2-5 years of experience) in three phases:

1. We discussed the tool's purpose and features and showed the engineers how to install and use it.
2. The engineers took two hours to go over the lectures and try out the challenge system. During this time, we observed their interaction, answered any questions, and noted any usability issues.
3. We discussed the tool's usefulness and examined areas for improving its features and the presented content.

The main takeaways from the focus group discussion included a set of features, technical improvements, and content enhancements. Furthermore, we received unanimous praise for the challenge subsystem, particularly how it integrated into the IDE, allowing the engineers to use familiar code refactoring tools.

4.2 Controlled Experiment Design

We designed our experiment based on the body of research that performed a controlled experiment to evaluate their software solutions. Wettel et al. (2011) performed an exhaustive survey of experimental validation research in software engineering to compile an experimental design wish list. Our controlled experiment is very similar to the one they designed.

The purpose of our experiment is to quantitatively evaluate the effectiveness of our Clean CaDET ITS compared to the traditional learning approach. To this aim, we formulated the following research questions and their corresponding hypothesis (Table 2):

RQ1. Does the use of our ITS increase the *correctness* of the solutions to code design test questions compared to conventional consumption of the static content?

RQ2. Is the use of our ITS more *engaging* for learners than the conventional consumption of static content?

RQ3. Does the use of our ITS increase *knowledge retention* compared to the conventional consumption of static content?

RQ4. Which properties of our ITS appeal to learners and which should be improved?

Table 2: The null and alternative hypothesis corresponding to our research questions RQ1-RQ3.

Null hypothesis	Alternative hypothesis
($H1_0$) Our ITS does not impact the correctness of the solutions to code design test questions.	($H1$) Our ITS impacts the correctness of the solutions to code design test questions.
($H2_0$) Our ITS does not impact knowledge retention.	($H2$) Our ITS impacts knowledge retention.
($H3_0$) Our ITS does not impact engagement with the studied material.	($H3$) Our ITS impacts engagement with the studied material.

To answer RQ1 – RQ3, we have *three dependent variables* in our experiment: (1) post-test correctness, (2) engagement, and (3) knowledge retention. Our experiment has *one independent variable*: the means of learning about clean code design practices. Thus, we have two treatments in our experiment:

- The experimental group: learners learning about clean code design through our ITS.
- The control group: learners learning about clean code design via traditional learning techniques. To facilitate traditional learning, we offer static learning content through an LMS.

To avoid the possibility of the educational materials influencing the experiment outcome, we offer the same educational materials to both groups[10]. The design of our experiment is a *between-subjects design*, i.e., a subject is part of either the control group or the experimental group.

Our *controlled variable*, i.e., a factor that could influence the participants' performance, is the skill level obtained after listening to several software engineering courses. The participants in our experiments are third-year undergraduate software engineering students with roughly the same knowledge and experience in software engineering. However, we use the *randomized block design* to reduce the possibility of differences in skill levels affecting the experiment. That is, we first divide our subjects into groups (*blocks*) according to their skill level. Then, we randomly assign subjects from each block to the two treatments for a between-subjects design.

The flow of our experiment execution was as follows:

1. The participants applied for the experiment by answering a pre-experiment survey. We used the answers to assess the participants' skill level and divided them into treatment groups accordingly.
2. The experiment was held in a single day and took three hours: participants had 2.5 hours to learn the subject matter (i.e., the learning phase) and 0.5 hours to complete the post-test.
3. A day after the experiment, we asked the participants to answer a post-experiment survey to collect their experience with the experiment and feedback to improve our ITS.
4. A week after the experiment, we organized a knowledge retention test.

We constructed the pre-experiment and post-experiment surveys following Punter et al. (2003) recommendations for effective online surveys.

We estimated the participants' skill level (controlled variable) according to their:

- Years of programming experience: we asked the learners to specify the years of their programming experience in general and their experience with the C# programming language that we used in our experiment.
- Grades: we asked the learners to specify their average grade and the grades they achieved on the courses related to software engineering they completed.

To measure the *post-test correctness* (dependent variable), we constructed a test designed to evaluate the learners' conceptual knowledge (through carefully constructed multiple-choice questions) and procedural knowledge (where they had to analyze code and refactor it using the learned principles). We administrated this test after the learning phase and a short break. We added the scores of individual questions to obtain the value of the post-test correctness variable. Participants were asked not to share their post-text experience with other participants to avoid influencing the knowledge retention test.

To measure participants' *engagement* with the educational materials (dependent variable), we used their answers from the post-experiment survey to a question: "I was focused and engaged during the experiment." This question was a 5‑point response

[10] The offered educational material is based on the guidelines for creating engaging digital educational content from our earlier work (Luburić et al., 2021b).

scale Likert item (Subedi, 2016), ranging from 1 = strongly disagree to 5 = strongly agree. To confirm the participants' self-assessment, we observed them during the learning phase following guidelines on conducting empirical research (Taylor et al., 2015) to estimate their engagement with the educational materials.

To measure *knowledge retention* (dependent variable), we asked the learners to solve the identical post-test a week after their learning experience and measured the correctness of their solutions. Karaci et al. (2018) and Abu-Naser (2009) measured knowledge retention similarly but applied the knowledge retention test a month after the post-test. Due to organizational constraints[11], we had to shorten this period to a week. We derived the knowledge retention variable the same way as the post-test correctness variable.

To answer RQ4, we constructed the post-experiment survey questions we administered to the experimental group participants (learners using our ITS). We constructed the following structured Likert item questions:

Q1. "I would integrate the Clean CaDET ITS into everyday learning."

Q2. "The Clean CaDET ITS has a user-friendly UI and is easy to use."

Q3. "The challenges are too demanding."

Q4. "The hints and solutions to the challenges are clear and useful."

Additionally, we asked the experiment group learners to answer the following two multiple-choice questions:

Q5. "I wish there were LESS,"

Q6. "I wish there were MORE,"

where the choices were different types of learning objects: (1) text-based explanations, (2) images, (3) videos, (4) multiple-choice questions, (5) challenges.

4.3 Controlled Experiment Results

A total of 51 learners participated in our experiment, where we assigned 33 learners to the experiment group and 18 learners to the control group. Our experiment is a between-subjects, unbalanced experiment as the groups are of unequal size. As we have one independent variable, the appropriate parametric test for hypothesis testing is the one-way Analysis Of Variance (ANOVA) (Norman, 2010; Wettel et al., 2011). Before the analysis, we ensured that our data met the test's assumptions:

1. The independence of observations assumption is met due to the between-subject design of the experiment.
2. We tested the homogeneity of variances of the dependent variables' assumption of all our dependent variables' using Levene's test (Levene, 1961) and found that the assumption was met in all cases.
3. We tested the normality of the dependent variable across levels of the independent variables using the Shapiro-Wilk test for normality (Shaphiro and Wilk, 1965). We found that the normality assumption was met for the post-test correctness and knowledge retention variables. However, the assumption was violated for the engagement variable.

Therefore, we used the one-way ANOVA parametric test for the post-test correctness and knowledge retention variables. We used the nonparametric Kruskal-Wallis H Test for the engagement variable. Kruskal-Wallis H Test does not require data normality and is well-suited for ordinal Likert item questions (MacFarland and Yates, 2016). We performed all statistical tests using the IBM SPSS statistical package. We used the significance level α of .05; if the p-value is less than 0.05 ($p \leq .05$), we consider the result significant.

Table 3 shows the mean and standard deviation for the post-test correctness and knowledge retention variables and ANOVA test results for the related null hypothesis ($H1_0$ and $H2_0$). We observe that the mean scores fall near the center of the score range (15). Furthermore, the Shapiro-Wilk test for normality showed that the scores are normally distributed. This result indicates that our test was well-constructed – not too hard and not too easy to solve. We can see that the mean value for the post-test correctness and the knowledge retention test is higher for the experiment than the control group. However, the ANOVA test shows that we do not have sufficient evidence to reject the null hypothesis (i.e., to claim these means are not equal).

Figure 3 shows the distributions of the self-assessed participant engagement for the experiment and control group. The Kruskal-Wallis H test showed that there was not a statistically significant difference in engagement between the control and experiment group, $\chi2(2) = 2.177$, $p = 0.140 > 0.05$, with a mean rank engagement score of 22.17 for the control group, and 28.09 for the experiment group. Our observations

[11] If we delayed the knowledge retention test any longer, learners would learn the same subject matter on their course, influencing the results.

of the participants during the learning phase concur with this result.

To summarize, for RQ1 – RQ3, we cannot reject the null hypothesis. Thus, we conclude that the use of our ITS does not impact the correctness of the solutions to code design test questions, knowledge retention, or engagement with the studied material. We consider this result positive as it establishes that our system does not lead to worse learning outcomes than the traditional learning approach.

Table 3: Participant scores (mean ± standard deviation) on the immediate post-test and the identical knowledge retention test administered a week later. The range of test scores is [0, 30].

Treatment group	Post-test correctness	Knowledge retention
Experiment	17.4 ± 2.76	17.8 ± 2.64
Control	16.5 ± 2.85	16.4 ± 3.22
ANOVA	$F_{1,49} = 1.274$ $p = 0.264 > 0.05$	$F_{1,49} = 2.822$ $p = 0.099 > 0.05$

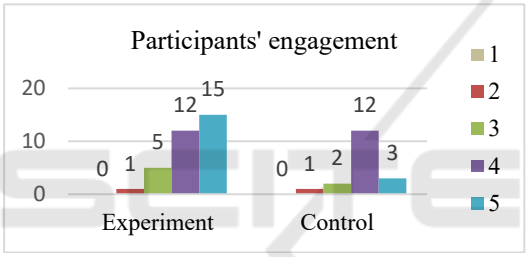

Figure 3: Self-assessed participant engagement (5-point response scale Likert item question).

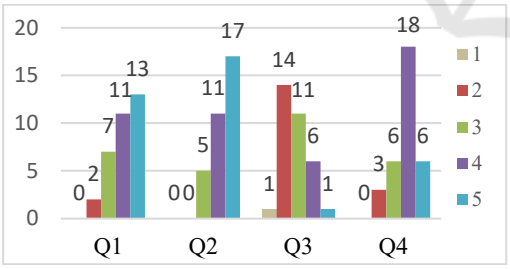

Figure 4: Participants' answers to the 5-point Likert item questions Q1-Q4.

Next, we analyze the post-experiment survey questions constructed to answer RQ4. Figure 4 shows the participants' answers to the 5-point Likert item questions Q1-Q4. We can see that most of the participants would integrate our Clean CaDET ITS into everyday learning (median 4) and thought that its user interface (UI) is user-friendly and easy to use (median 5). The participants did not find the challenges we offered too easy or demanding (median 3) and found the hints and solutions to these challenges clear and useful (median 4). We conclude that the participants were satisfied with our ITS.

To answer RQ4, we can look at Table 4 that shows which learning objects participants found engaging and helpful (and wished more were offered) and which learning objects were overrepresented. We also show the number of learning object types offered through our ITS to put these answers into context. From Table 4, we see that our participants mostly wished for more challenges out of the offered learning object types. The multiple-choice questions closely followed this. However, judging by the number of multiple-choice questions offered through our platform, we can see that this may be due to these learning objects being underrepresented compared to other learning object types. We can also conclude that the number of video learning objects is balanced and that the participants wished there were more images and fewer text-based explanations.

Table 4: Participants' answers to questions Q5 and Q6: number of learning object types offered to the participants through our ITS and the number of participants that wish there were more or less of these object types.

Learning object type	No. of served objects	"I wish there were LESS"	"I wish there were MORE"
Text-based explanations	11	8	4
Images	7	1	7
Videos	4	4	5
Multi-choice questions	2	5	12
Challenges	5	3	13

5 DISCUSSION

In Section 2.2, we presented related clean code tutors and listed two categories of limitations from which most suffer. In this section, we examine how our approach stands against these limitations.

Section 5.1 examines the limitations of performed empirical evaluations for ITSs, discusses how our empirical study addresses these limitations and lists the related threats to our evaluations' validity. Section 5.2 reconsiders the problem of rooting ITS development in education theory advances. We use our ITS as a case study to examine the benefits of adhering to these advances. Finally, Section 5.3 discusses related challenges, including ambiguous knowledge components and laborious content authoring.

5.1 Evaluation Limitations

Software engineering is intensely people-oriented. Consequently, empirical studies and evaluations have become a required element of mature software engineering research (Shull et al., 2007). While it is helpful to utilize the many empirical methods to evaluate a tool's usability, gather ideas for features, or explore the users' engagement, the priority should be to evaluate the extent to which the tool fulfills its purpose. For an ITS, this is its ability to facilitate robust learning, given a reasonable amount of time (Koedinger et al., 2012). We have seen that most existing ITSs specialized in clean code do not perform such evaluation.

While we employed two empirical methods to evaluate our ITS, both introduce threats to our study's validity.

First, we conducted a focus group discussion to gather early insight into our tool, assess its usability and overall usefulness. This evaluation was a valuable stepping stone, but it did not examine learning outcomes. Furthermore, it included a small number of participants from a single software vendor, limiting the gathered insight.

Second, the controlled experiment included a modest number of participants and content, including 51 students and two lectures, examined over three hours. At this scale, our results may be misleading, and further experimentation in the classroom is required to verify the achieved learning outcomes. Furthermore, our evaluation is limited to software engineering students, and it is unclear how effective our ITS is for professional software engineers. Finally, our knowledge retention test introduces a validity threat, as it was conducted only a week after the initial learning.

5.2 Benefits of Anchoring ITS Research in Broader ITS Design Advances

Anchoring the design and development of an ITS to a common framework brings several benefits.

Firstly, it improves communication by bringing consistency to the used terminology. The shared concepts are present in discussions among research team members, the ITS code, documentation, and scientific papers. Without a baseline framework, we can state that our instructors examined the challenge submissions to find patterns of incorrect submissions, to improve the educational materials, or add new challenges. By anchoring the previous statement in the KLI framework (Koedinger et al., 2012), we can state that our instructors examined the assessment events (i.e., challenge submissions) to analyze which knowledge components were underdeveloped. Our instructors then conducted design-loop adaptivity (Aleven et al., 2016a) by enhancing existing instructional events or introducing new assessment events that target the underdeveloped knowledge components. We can better organize and exchange ideas, design experiments, and communicate novelty by utilizing these established concepts.

Secondly, by anchoring our specialized ITS to a tried and tested framework, we benefit from innovations that build upon the same framework. For example, when designing the challenge hint feature, we can directly integrate Aleven et al. (2016b) insights for delivering effective instructional events in the form of hints. Likewise, we can enhance our design-loop adaptivity by expanding our progress model with advances from Huang et al. (2021). As a final example, by understanding the structure of the knowledge component (e.g., application conditions, nature of response) targeted by our challenges, we can benefit from the advances from any field that targets a knowledge component of the same structure.

5.3 Challenges with ITS Design

We explore two challenges that plague ITS design, which are very much present in our work – the ambiguity of knowledge and the difficulty of authoring ITS content.

Knowledge and knowledge structures, in general, suffer from ambiguity (Law, 2014). One aspect of this problem is determining the appropriate granularity of a knowledge component or learning objective. For example, Haendler et al. (2019) use Bloom's taxonomy as a foundation and point out that their Tutor develops higher cognitive processes like *analysis* and *evaluation*. However, the assignments in their Tutor often decompose the complex problem of refactoring a class into simple steps the learner should perform (e.g., "add an attribute to a class"). The presence of these steps reduces the cognitive level required to complete the task (e.g., adding an attribute requires *remembering*, the first level of Bloom's taxonomy). We do not intend to criticize the approach provided by Haendler et al. We point to the broader problem of knowledge ambiguity and how introducing a subtask, or a hint can degrade the cognitive process required to complete the task. Knowledge components are not exempt from this ambiguity, which is why Koedinger et al. (2012) provide many guidelines for their use. A common challenge is defining the granularity of a knowledge component, where "write clean code" is significantly

different from “Use meaningful words in identifier names.” Despite our best efforts, we still struggle with knowledge engineering, and we expect to refine our knowledge components, like those presented in Table 1, for many more iterations.

A second limitation present in many ITSs, including ours, is the time it takes to develop the content. Creating a refactoring challenge for our platform entails:

- Writing the starting challenge code,
- Defining one or more rules for each knowledge component the challenge assesses,
- Creating one or more learning objects to act as hints for each rule,
- Writing the assignment text, and
- Optionally developing a test suite to ensure the code is functionally correct.

To help mitigate this somewhat inherent issue, we plan to develop high-usability authoring tools to support instructors.

6 CONCLUSION

We have developed and empirically evaluated an ITS specialized for clean code analysis and refactoring. Our novelty lies in the refactoring challenge subsystem, anchored in advances from broader ITS design advances. We have empirically evaluated our tool to collect ideas for improvement and ensure it produces satisfactory learning outcomes.

Throughout the paper, we have emphasized the need to develop novel solutions in specialized fields by grounding our work in advances from broader and well-tested domains. While we have chosen Koedinger et al. (2012), Aleven et al. (2016a), and the related body of literature as our anchors, we do not claim this work to be the only option. Mature research groups have produced many advances in intelligent tutoring system design and educational data mining. It is at the intersection of these general advances and the intricacies of specialized fields like clean code where novelty can be found.

As further work, we will work on the authoring problem to simplify challenge creation. We will also monitor advances that spring from our anchors to incorporate helpful solutions into our ITS and continuously evaluate our advances to ensure they contribute to the ultimate goal of producing robust learning outcomes.

ACKNOWLEDGEMENTS

Funding provided by Science Fund of the Republic of Serbia, Grant No 6521051, AI-Clean CaDET.

REFERENCES

Abu-Naser, S.S., 2009. Evaluating the Effectiveness of the CPP-Tutor, an Intelligent Tutoring System for Learners Learning to Program in C+. Journal of Applied Sciences Research, 5(1), pp.109-114.

Aleven, V., McLaughlin, E.A., Glenn, R.A. and Koedinger, K.R., 2016, Instruction based on adaptive learning technologies. Handbook of research on learning and instruction, pp.522-560.

Aleven, V., Roll, I., McLaren, B.M. and Koedinger, K.R., 2016. Help helps, but only so much: Research on help seeking with intelligent tutoring systems. International Journal of Artificial Intelligence in Education, 26(1), pp.205-223.

Fiorella, L. and Mayer, R.E., 2016. Eight ways to promote generative learning. Educational Psychology Review, 28(4), pp.717-741.

Fowler, M., 2018. Refactoring: improving the design of existing code. Addison-Wesley Professional.

Gagne, R.M., Wager, W.W., Golas, K.C., Keller, J.M. and Russell, J.D., 2005. Principles of instructional design.

Gervet, T., Koedinger, K., Schneider, J. and Mitchell, T., 2020. When is Deep Learning the Best Approach to Knowledge Tracing?. JEDM| Journal of Educational Data Mining, 12(3), pp.31-54.

Graesser, A.C., Conley, M.W. and Olney, A., 2012, Intelligent tutoring systems. APA educational psychology handbook, Vol 3: Application to learning and teaching., pp.451-473.

Haendler, T., Neumann, G. and Smirnov, F., 2019, May. RefacTutor: an interactive tutoring system for software refactoring. In International Conference on Computer Supported Education (pp. 236-261). Springer, Cham.

Haendler, T. and Neumann, G., 2019. A Framework for the Assessment and Training of Software Refactoring Competences. In KMIS (pp. 307-316).

Holstein, K., McLaren, B.M. and Aleven, V., 2017, March. Intelligent tutors as teachers' aides: exploring teacher needs for real-time analytics in blended classrooms. In Proceedings of the seventh international learning analytics & knowledge conference (pp. 257-266).

Hozano, M., Garcia, A., Fonseca, B. and Costa, E., 2018. Are you smelling it? Investigating how similar developers detect code smells. Information and Software Technology, 93, pp.130-146.

Huang, Y., Lobczowski, N.G., Richey, J.E., McLaughlin, E.A., Asher, M.W., Harackiewicz, J.M., Aleven, V. and Koedinger, K.R., 2021, April. A general multi-method approach to data-driven redesign of tutoring systems. In LAK21: 11th International Learning Analytics and Knowledge Conference (pp. 161-172).

IEEE Computer Society, 2020, IEEE Standard for Learning Object Metadata (1484.12.1-2020). Standard, IEEE, New York, NY, USA.

Imhof, C., Bergamin, P. and McGarrity, S., 2020, Implementation of adaptive learning systems: Current state and potential. In Online Teaching and Learning in Higher Education (pp. 93-115). Springer, Cham.

Karaci, A., Akyüz, H.I., Bilgici, G. and Arici, N., 2018. Effects of Web-Based Intelligent Tutoring Systems on Academic Achievement and Retention. International Journal of Computer Applications, 181(16), pp.35-41.

Keuning, H., Heeren, B. and Jeuring, J., 2020, November. Student refactoring behaviour in a programming tutor. In Koli Calling'20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research (pp. 1-10).

Keuning, H., Heeren, B. and Jeuring, J., 2021, March. A tutoring system to learn code refactoring. In Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (pp. 562-568).

Khanal, S.S., Prasad, P.W.C., Alsadoon, A. and Maag, A., 2019. A systematic review: machine learning based recommendation systems for e-learning. Education and Information Technologies, pp.1-30.

Koedinger, K.R., Corbett, A.T. and Perfetti, C., 2012, The Knowledge‐Learning‐Instruction framework: Bridging the science‐practice chasm to enhance robust learner learning. Cognitive science, 36(5), pp.757-798.

Law, K.K., 2014. The problem with knowledge ambiguity. European Management Journal, 32(3), pp.444-450.

Levene, H., 1961. Robust tests for equality of variances. Contributions to probability and statistics. Essays in honor of Harold Hotelling, pp.279-292.

Luburić, N., Prokić, S., Grujić, K.G., Slivka, J., Kovačević, A., Sladić, G. and Vidaković, D., 2021. Towards a systematic approach to manual annotation of code smells.

Luburić, N., Slivka, J., Sladić, G. and Milosavljević, G., 2021. The challenges of migrating an active learning classroom online in a crisis. Computer Applications in Engineering Education.

MacFarland, T.W. and Yates, J.M., 2016. Kruskal–Wallis H-test for oneway analysis of variance (ANOVA) by ranks. In Introduction to nonparametric statistics for the biological sciences using R (pp. 177-211). Springer, Cham.

Mayer, R.E., 2014. Incorporating motivation into multimedia learning. Learning and instruction, 29, pp.171-173.

Normadhi, N.B.A., Shuib, L., Nasir, H.N.M., Bimba, A., Idris, N. and Balakrishnan, V., 2019. Identification of personal traits in adaptive learning environment: Systematic literature review. Computers & Education, 130, pp.168-190.

Norman, G., 2010. Likert scales, levels of measurement and the “laws” of statistics. Advances in health sciences education, 15(5), pp.625-632.

Piech, C., Bassen, J., Huang, J., Ganguli, S., Sahami, M., Guibas, L.J. and Sohl-Dickstein, J., 2015. Deep Knowledge Tracing. Advances in Neural Information Processing Systems, 28, pp.505-513.

Prokić, S., Grujić, K.G. Luburić, N., Slivka, J., Kovačević, A., Vidaković, D., Sladić, G., 2021. Clean Code and Design Educational Tool. In 2021 44th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO) (In Press). IEEE.

Punter, T., Ciolkowski, M., Freimut, B. and John, I., 2003, September. Conducting on-line surveys in software engineering. In 2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings. (pp. 80-88). IEEE.

Rosé, C.P., McLaughlin, E.A., Liu, R. and Koedinger, K.R., 2019. Explanatory learner models: Why machine learning (alone) is not the answer. British Journal of Educational Technology, 50(6), pp.2943-2958.

Sandalski, M., Stoyanova-Doycheva, A., Popchev, I. and Stoyanov, S., 2011. Development of a refactoring learning environment. Cybernetics and Information Technologies (CIT), 11(2).

Shaphiro, S. and Wilk, M., 1965. An analysis of variance test for normality. Biometrika, 52(3), pp.591-611.

Sharma, T. and Spinellis, D., 2018. A survey on software smells. Journal of Systems and Software, 138, pp.158-173.

Shull, F., Singer, J. and Sjøberg, D.I. eds., 2007. Guide to advanced empirical software engineering. Springer Science & Business Media.

Shute, V. and Towle, B., 2003, Adaptive e-learning. Educational psychologist, 38(2), pp.105-114.

Siemens, G., 2013. Learning analytics: The emergence of a discipline. American Behavioral Scientist, 57(10), pp.1380-1400.

Subedi, B.P., 2016. Using Likert type data in social science research: Confusion, issues and challenges. International journal of contemporary applied sciences, 3(2), pp.36-49.

Taylor, S.J., Bogdan, R. and DeVault, M., 2015. Introduction to qualitative research methods: A guidebook and resource. John Wiley & Sons.

Templeton, J.F., 1994. The focus group: A strategic guide to organizing, conducting and analyzing the focus group interview. Probus Publishing Company.

Tom, E., Aurum, A. and Vidgen, R., 2013. An exploration of technical debt. Journal of Systems and Software, 86(6), pp.1498-1516.

Wettel, R., Lanza, M. and Robbes, R., 2011, May. Software systems as cities: A controlled experiment. In Proceedings of the 33rd International Conference on Software Engineering (pp. 551-560).

Wiese, E.S., Yen, M., Chen, A., Santos, L.A. and Fox, A., 2017, April. Teaching students to recognize and implement good coding style. In Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale (pp. 41-50).