An Interactive Tutoring System for Training Software Refactoring

Thorsten Haendler, Gustaf Neumann and Fiodor Smirnov

Institute for Information Systems and New Media,
Vienna University of Economics and Business (WU), Austria

Keywords: Intelligent Tutoring System, Software Refactoring, Software Design, Code Visualization, Software-engineering Education and Training, Unified Modeling Language (UML2), Interactive Training Environment.

Abstract: Although considered useful and important, software refactoring is often neglected in practice because of the perceived risks and difficulties of performing it. A way to address these challenges can be seen in promoting developers’ practical competences. In this paper, we propose an approach for an interactive training environment for addressing practical competences in software refactoring. In particular, we present a tutoring system that provides interactive feedback to the users (e.g., university students or software developers) regarding the software-design quality and the functional correctness of the (modified) source code. After each code modification (refactoring step), the user can review the results of run-time regression tests and compare the actual software design (as-is) with the targeted design (to-be) in order to check quality improvement. For this purpose, structural and behavioral diagrams of the Unified Modeling Language (UML2) representing the as-is software design are automatically reverse-engineered from source code. The to-be design diagrams (also in UML) can be pre-specified by the instructor. We illustrate the usability of the approach for training competences in refactoring via short application scenarios and describe exemplary learning paths. Moreover, we provide a web-based software-technical implementation in Java (called refacTutor) to demonstrate the technical feasibility of the approach. Finally, limitations and further potential of the approach are discussed.

1 INTRODUCTION

As a result of time pressure in software projects, priority is often given to implementing new features rather than ensuring code quality (Martini et al., 2014). In the long run, this leads to software aging (Parnas, 1994) and increased technical debt with the consequence of increased maintenance costs on the software system (i.e., debt interest) (Kruchten et al., 2012). A popular technique for code-quality assurance is software refactoring, which aims at improving code quality by restructuring the source code while preserving the external system behavior (Opdyke, 1992; Fowler et al., 1999). Several kinds of flaws (such as code smells) can negatively impact code quality and are thus candidates for software refactoring (Alves et al., 2016). Besides kinds of smells that are relatively simple to identify and to refactor (e.g., stylistic code smells), others are more complex and difficult to identify, to assess and to refactor, such as smells in software design and architecture (Fowler et al., 1999; Suryanarayana et al., 2014; Nord et al., 2012) Although refactoring is considered useful and important, it is often neglected in practice due to several barriers, among which are (perceived by software developers) the difficulty of performing refactoring, the risk of introducing an error into a previously correctly working software system, and a lack of adequate tool support (Tempero et al., 2017). These barriers pose challenges with regard to improving refactoring tools and promoting the skills of developers.

On the one hand, in last years, several tools have been proposed that apply static analysis for identifying smells via certain metrics and benchmarks (e.g., level of coupling between system elements) or supporting in planning and performing the actual refactoring (steps), such as JDedodrant (Tsantalis et al., 2008) and DECOR (Moha et al., 2010), as well as for measuring and quantifying their impact in terms of technical debt, such as SonarQube (Campbell and Papapetrou, 2013) or JArchitect (CoderGears, 2018). However, for more complex kinds of smells (e.g., on the level of software design and architecture) the smell detection and refactoring is difficult. Thus, these tools only cover a modest amount of smells (Fernandes et al., 2016; Fontana et al., 2012) and tend to produce false positives (Fontana et al., 2016) (which, i.a., represent constructs intentionally used by
the developer with symptoms similar to smells, such as certain design patterns). In addition, the decision what and how to refactor also depends on domain knowledge (e.g., regarding design rationale) provided by human experts such as software architects. Due to these issues, the refactoring process is still challenging and requires human expertise.

On the other hand, so far only little attention has been paid in research to education and training in the field of software refactoring. Besides textbooks with best practices and rules on how to identify and to remove smells via refactoring, e.g., (Fowler et al., 1999), there are only a few approaches (see Section 6) for supporting developers in accomplishing or improving practical and higher-level competences such as application, analysis and evaluation according to Bloom’s taxonomy of cognitive learning objectives; see, e.g., (Bloom et al., 1956; Krathwohl, 2002).

In this paper, we propose an approach for an interactive learning environment for training practical competences in software refactoring. In particular, we present a tutoring system that provides interactive feedback and decision-support regarding both the software-design quality and the functional correctness of the source code modified by the user (see Fig. 1).

Driven by a refactoring task defined by the instructor, after each refactoring step, the user can review the results of run-time regression tests (e.g., pre-specified by the instructor; in order to check that no error has been introduced) and compare the actual software design (as-is) with the intended design (to-be; in order to check quality improvement). For this purpose, structural and behavioral software-design diagrams of the Unified Modeling Language (UML2) (Object Management Group, 2015) representing the as-is software design are automatically reverse-engineered from source code. The to-be design diagrams (also in UML) can be pre-specified by the instructor. UML is the de-facto standard for modeling and documenting structural and behavioral aspects of software design. There is evidence that UML design diagrams are beneficial for understanding software design issues (Arisholm et al., 2006; Scanniello et al., 2018; Haendler, 2018). This way, the approach primarily addresses the refactoring of issues on the level of software design and architecture. Moreover, the code visualization supports in building and questioning users’ mental models, see, e.g., (Cañas et al., 1994; George, 2000). In addition, the system optionally reports on other quality aspects provided by quality-analysis tools (Campbell and Papapetrou, 2013).

In order to illustrate the usability for teaching and training refactoring competences, we describe exemplary learning paths, specify the corresponding exercise-interaction workflow and draw exemplary application scenarios (based on different learning objectives and tasks and with corresponding settings (configuration options) for decision support and feedback. We also introduce a proof-of-concept implementation in Java (called refactutor) in terms of a web-based development environment. Its application is demonstrated via a short exercise example. In the following Section 2, a short conceptual overview of the proposed approach is given.

2 CONCEPTUAL OVERVIEW

Fig. 2 depicts an overview of the tutoring-system infrastructure in terms of technical components and artifacts used, created or modified by instructor and/or user respectively.

---

1 See Section 5. The software prototype is available for download from http://refactoringgames.com/refactutor
Focused on certain learning objectives (for details, also see Sections 3 and 4), the instructor at first specifies a task, prepares the initial source code for exercise, specifies (or re-uses an existing) test script, and specifies the to-be software design (see step 1 in Fig. 2). The user’s exercise then starts by analyzing the task (step 2). After each code modification (step 3), the user can review several kinds of feedback created by different analysis tools that analyze the code (see steps 4 to 6). These analyzers consist of a test framework for checking the functional correctness (test result), a quality analyzer that examines the code using pre-configured quality metrics (for providing hints on quality issues), and a diagram builder that derives (reverse-engineers) diagrams from source code and reflects the current as-is software design. This way, the tutoring system supports a cyclic exercise workflow of steps 3 to 5.

**Structure.** The remainder of this paper is structured as follows. In Section 3, the applied conceptual framework of the interactive tutoring system (including competences and learning paths, exercises, code modification as well as feedback mechanisms) are elaborated in detail. Section 4 illustrates the usability of the training environment for two exercise scenarios (i.e., design refactoring and test-driven development). Then, in Section 5, we present a proof-of-concept implementation in Java; including a detailed exercise example. In Section 6, related approaches are discussed. Section 7 reflects limitations and further potential of the approach and Section 8 concludes the paper.

## 3 INTERACTIVE TUTORING ENVIRONMENT

Fig. 3 gives an overview of important conceptual aspects for training software refactoring. The model is structured into four layers that concretize from an exemplary learning and training path (A) (built on exemplary environments) via the exercises performed by the user (B) to refactoring paths and system states (C) and finally views on system states (D) (feedback and decision support). The corresponding aspects are described below in detail.

### 3.1 Learning Objectives and Paths

The learning activity sequence (IMS Global Consortium, 2003) (depicted on the top layer A in Fig. 3) illustrates an exemplary learning path for software refactoring by sequence multiple exercise units, which base on different learning environments. The path can be driven by certain learning objectives, e.g. oriented to Bloom’s taxonomy of cognitive learning objectives (Krathwohl, 2002). For instance, via **lecture** based on best practices and rules provided by a refactoring textbook, e.g., (Suryanarayana et al., 2014), basic competences (i.e., knowledge and comprehension) may be addressed, e.g., the (theoretical) understanding of rules for identifying refactoring candidates and for performing refactoring steps. For applying the comprehended techniques, e.g., units with **paper-based exercises** (PBE) can follow. In order to actually apply and improve practical competences on executable code (via editor), a **tutoring system** then can be used (see below). Moreover, via playing **serious games** (providing or simulating real-world conditions, such as a large code base) (Haendler and Neumann, 2019), further higher-level competences can be acquired such as analysis and evaluation (e.g., while developing and applying refactoring strategies for prioritizing smell candidates and refactoring options). This exemplary path along different training environments demonstrates that the proposed tutoring-system approach has to be seen as a complement to other approaches, each focusing on certain competence levels.

### 3.2 Structure of Exercises

A unit might generally consist of several exercises. Each exercise of the proposed tutoring system (see the second layer B in Fig. 3) has a certain sequence consisting of the **task** (instruction) defined by the instructor, the exercise **interaction** of the user with the tutoring system and the exercise **assessment**, which will be explained in detail below.
The task defines the actual instructions to be performed by the user. The instructions might be oriented to e.g., a challenge to be solved (e.g., to remove a certain smell), a goal to be achieved (e.g., to realize a certain design structure), or instructions for concrete modification actions (e.g., to perform an EXTRACT METHOD refactoring). For examples with further details, also see Section 4).

The exercise interaction represents a cycle of analysis and actions performed by the user which is built on feedback provided by the tutoring system. Fig. 4 represents the workflow of the exercise interaction in terms of a double-loop learning and decision-making cycle supported by the tutoring system at the beginning (e.g., as prepared by the instructor). Each actual or possible modification to be performed on a certain state then leads to a certain following state (TRANSITIONAL). Depending on the kind of task and smell, there are different ways of achieving the defined exercise goal. As mentioned above, for more complex smells several options and sequences for achieving a desired design can exist, such as for some types of software-design smells (Suryanarayana et al., 2014). This way, the options and steps of possible modifications can be represented via a graph structure (i.e., modifications as edges, states as nodes). The performance of one user (via several refactoring steps starting from and/or leading to a TRANSITIONAL) then draws a concrete path through this graph. The final state can either be passing (FINAL(PASSING)) (i.e., fulfilling the quality standards, e.g., defined by to-be design diagrams, and passing the tests) or failing (FINAL(NON-PASSING)). For each state, several aspects are important for refactoring which are reflected by selected views (see below).

3.3 Code Modifications and System States

During exercise interaction, the system under analysis (SUA) can take different system states (see (2) in Fig. 3). The states can be classified into the initial state (INITIAL), the final state(s) (i.e., FINAL(PASSING) and FINAL(NON-PASSING)) and multiple transitional states TRANSITIONAL. In particular, the initial state INITIAL represents the system at the beginning (e.g., as prepared by the instructor). Each actual or possible modification to be performed on a certain state then leads to a certain following state (TRANSITIONAL). Depending on the kind of task and smell, there are different ways of achieving the defined exercise goal. As mentioned above, for more complex smells several options and sequences for achieving a desired design can exist, such as for some types of software-design smells (Suryanarayana et al., 2014). This way, the options and steps of possible modifications can be represented via a graph structure (i.e., modifications as edges, states as nodes). The performance of one user (via several refactoring steps starting from and/or leading to a TRANSITIONAL) then draws a concrete path through this graph. The final state can either be passing (FINAL(PASSING)) (i.e., fulfilling the quality standards, e.g., defined by to-be design diagrams, and passing the tests) or failing (FINAL(NON-PASSING)). For each state, several aspects are important for refactoring which are reflected by selected views (see below).

3.4 Views on System States

Oriented to viewpoint models applied in software architecture such as (Kruchten, 1995), we apply a viewpoint model for refactoring that includes the following four views on system states, i.e. code, test be-
behavior as well as design structure and behavior (see \( \Phi \) in Fig. 2). The views conform to the viewpoint model depicted in Fig. 5 and serve as feedback and decision support for the user on each system state. In particular, there is evidence that for identifying and assessing software design smells (such as ABSTRACTION or MODULARIZATION smells), a combination of UML class diagrams (providing inter-class couplings evoked by method-call dependencies) and sequence diagrams (representing run-time scenarios) is appropriate (Haendler, 2018). The viewpoint model depicted in Fig. 5 reflects the views on system states.

![Figure 5: Applied viewpoint model for refactoring.](image)

In addition to the source code (modified by the user), the other views are provided by applying the following tools and techniques.

- **The test behavior** is specified (in terms of a test script) using a certain test framework (e.g., XUnit or scenario-based tests). By performing the tests after each (important) modification (regression testing) the functional correctness of the code is ensured. Feedback is returned to the user in terms of the test result.

- For automatically deriving (a.k.a. reverse-engineering) the design diagrams representing the as-is software design of the source code, (automatic) static and dynamic analysis techniques are applied, see, e.g., (Richner and Ducasse, 1999; Kollmann et al., 2002; Haendler et al., 2015). The Unified Modeling Language (UML)(Object Management Group, 2015) is the de-facto standard for documenting software design. For representing design structure, UML class diagrams are extracted from source code; for design behavior, UML sequence diagrams are derived from execution traces. For further details on the derivation techniques and the resulting diagrams, see Section 5.

- In addition, also software-quality analyzers (e.g., SonarQube (Campbell and Papapetrou, 2013)) can be used to identify quality issues (such as smells) as well as to measure and quantify the technical debt score, which both then can be presented as hints or feedback to the user.

## 4 APPLICATION SCENARIOS

In order to illustrate the feasibility of the proposed tutoring-system approach for teaching and training software refactoring, two possible application scenarios are described. In Table 1, exemplary tasks with corresponding values of views and other exercise aspects are specified for two scenarios, i.e., (a) design refactoring and (b) test-driven development (TDD).

Table 1: Exemplary exercise scenarios: (a) design refactoring and (b) test-driven development.

<table>
<thead>
<tr>
<th>Aspects and Views</th>
<th>Design Refactoring</th>
<th>Test-driven Development (TDD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>Refactor the given (smelly) source code in order to achieve the defined-to-be design</td>
<td>Apply the red-green-refactor steps to realize the defined-to-be design and test behavior</td>
</tr>
<tr>
<td>Target Competence</td>
<td>Analyze software design and plan &amp; perform refactoring steps for removing a given smell</td>
<td>Knowledge on refactoring options for given smell type as well as on notations and meaning of UML diagrams</td>
</tr>
<tr>
<td>Prerequisite Competence</td>
<td>Knowledge on refactoring options for given smell type as well as on notations and meaning of UML diagrams</td>
<td>Knowledge on refactoring options for given smell type as well as on notations and meaning of UML diagrams</td>
</tr>
<tr>
<td>Code</td>
<td>Code with design smells</td>
<td>No code given</td>
</tr>
<tr>
<td>As-is Design</td>
<td>Representing design smells</td>
<td>No as-is design available</td>
</tr>
<tr>
<td>As-is Design</td>
<td>Conforming to to-be design</td>
<td>Conforming to to-be design</td>
</tr>
<tr>
<td>Tests</td>
<td>Tests pass in ((S_{\text{INITIAL}})) and in ((S_{\text{FINAL}}))</td>
<td>Tests fail in ((S_{\text{INITIAL}})) but pass in ((S_{\text{FINAL}}))</td>
</tr>
<tr>
<td>Assessment</td>
<td>As-is and to-be design are identical and all tests pass</td>
<td>As-is and to-be design are identical and all tests pass</td>
</tr>
</tbody>
</table>

### 4.1 Design Refactoring

First, consider the instructor aims at fostering the practical competences of analyzing the software design and performing refactoring steps (i.e., application and analysis, see target competences in Table 1). In this case, a possible training scenario can be realized by presenting a piece of source code that behaves as intended (i.e., runtime tests pass), but with smelly design structure (i.e., as-is design and to-be design are different). The user’s task then is to refactor the given code in order to realize the specified targeted design. As important prerequisite, the user needs to bring along the competences to already (theoretically) know the rules for refactoring and to analyze UML diagrams. Fig. 6 depicts the values of different system states while performing the exercise (also see Section 3.3 and 3.4). At the beginning (\(S_{\text{INITIAL}}\)), the tests...
pass, but code and design are smelly by containing, e.g., a MULTIFACETED ABSTRACTION smell (Suryanarayana et al., 2014). During the (path of) refactorings (e.g., by applying corresponding EXTRACT-CLASS and/or MOVE-METHOD/MOVE-FIELD refactorings (Fowler et al., 1999)), the values of the views can differ (box in the middle). Characteristic for a non-passing final state or the transitional states is that at least one view (e.g., test, design structure or behavior) does not meet the requirements. In turn, a passing final state fulfills the requirements for all view values (box on the right-hand side).

Besides these two exemplary scenarios, multiple other scenarios can be specified by varying the values of the exercise aspects and views (see Table 1).

5 SOFTWARE PROTOTYPE

In order to demonstrate the technical feasibility, we introduce our software prototype refaTutor2 that realizes key aspects of the infrastructure presented in Fig. 2. In the following, we focus on applied technologies, the derivation of UML class diagrams reflecting the as-is design, the graphical user interface (GUI), and a concrete exercise example.

5.1 Applied Technologies

We implemented a software prototype as a web-based application using Java Enterprise Edition (EE) to demonstrate the feasibility of the proposed concept. Java is also the supported language for the refactoring exercises. As editor for code modification the Ace Editor (Ajax.org, 2019) has been integrated, which is a flexible browser-based code editor written in JavaScript. After test run, the entered code is then passed to the server for further processing. For compiling the Java code, we applied InMemoryJavaCompiler (Trung, 2017), a GitHub repository that provides a sample of utility classes allowing compilation of Java sources in memory. The compiled classes are then analyzed using Java Reflection (Forman and For- man, 2004) in order to extract information on code structure. The correct behavior is verified via JUnit tests (Gamma et al., 1999). Relevant exercise information are stored in XML files including task description, source code, test script, and information for UML diagrams (see below).

5.2 Design-diagram Derivation

For automatically creating the as-is design diagrams, the extracted information is transferred to an integrated diagram editor. PlantUML (Roques, 2017) is an open-source and Java-based UML diagram editor that allows for passing plain text in terms of a simple DSL for creating graphical UML diagrams (such as class and sequence diagrams), i.a. in PNG or SVG format. PlantUML also manages the efficient and aesthetic composition of diagram elements.

Fig. 8 depicts an application example of source code to be refactored (left-hand side, Listing 1) with

---

2 The software prototype is available for download from http://refactoringgames.com/refactutor
Figure 8: Listing with exemplary Java code fragment for a refactoring task (left-hand side) and UML class diagram (right-hand side) representing the as-is design automatically derived from source code and visualized using PlantUML.

As-is design diagram (reflecting the current state of code) in terms of a UML class diagrams (right-hand side). In particular, the (derived) class diagrams provide associations in terms of references to other classes. For example, see (1) in the diagram and line 4 in the listing in Fig. 8. Generalizations represent is-a relationships (between sub-classes and super-classes); see (2) in the diagram and line 11 in the listing. Moreover, the derived class diagrams also provide call (or usage) dependencies between classes (see (3)), which, e.g., represent inter-class calls of attributes or methods from methods within a method (see lines 19 and 20 in the listing in Fig. 8). For deriving UML sequence diagrams, we apply dynamic reverse-engineering techniques based on the execution traces triggered by the runtime tests, already described and demonstrated in (Haendler et al., 2015) and (Haendler et al., 2017). As explained in Section 3.4, a combination of UML class and sequence diagrams can support in identifying and assessing issues in software design or architecture (Haendler, 2018).

5.3 GUI

As outlined in the overview depicted in Fig. 2, the prototype provides GUI perspectives for instructors configuring and preparing exercises and for users performing the exercises. Each role has a specific browser-based dashboard with multiple views (a.k.a. widgets) on different artifacts (as described in Fig. 2). Fig. 9 depicts the user’s perspective with provided views and an exercise example (which is detailed in Section 5.4). In particular, the user’s perspective comprises a view on the task description (as defined by the instructor; see (1) in Fig. 9). In (2), the tests scripts of the (tabbed) JUnit test cases are presented. The console output in (3) reports on the test result and additional hints. The code editor (see (4) in 9) provides the source code to be modified by the user (with tabs for multiple source files). Via the button in (5), the user can trigger the execution of the runtime tests. For both the to-be design (defined by the instructor beforehand; see (6)) and the actual as-is design (automatically derived after and while test execution and reflecting the state of the source code; see (7)), also tabs for class and sequence diagrams are provided. The teacher’s perspective is quite similar; in addition, it provides an editor for specifying the to-be diagrams.

5.4 Exercise Example

In the following, a short exercise example applying refactTutor is illustrated in detail. In particular, the source code, runtime tests, the task to be performed by the user and the corresponding as-is and to-be design diagrams are presented.

Source Code. For this exemplary exercise, the following Java code fragment (containing basic classes and functionality for a simple banking application) has been prepared by the instructor (see Listing 2 and also the editor (4) in Fig. 9).
Tests In this example, four JUnit test cases have been specified by the instructor (see Listing 3 and also ③) and the views ② in Fig. 9). Two out of these cases fail at the beginning (SINITIAL).

Listing 2: Java source code.

```java
public class BankAccount {
    private Integer number;
    private Double balance;
    private Limit limit;

    public BankAccount (Integer number, Double balance) {
        this.number = number;
        this.balance = balance;
    }

    public Integer getNumber () {
        return this.number;
    }

    public Double getBalance () {
        return this.balance;
    }

    public void increase (Double amount) {
        this.balance += amount;
    }

    public void decrease (Double amount) {
        this.balance -= amount;
    }
}

public class Limit {
    private Double amount;

    public Double getAmount () {
        return this.amount;
    }
}

public class Transaction {
    public Boolean transfer (Integer senderNumber, Integer receiverNumber, Double amount) {
        BankAccount sender = new BankAccount(123456,2000.00);
        BankAccount receiver = new BankAccount(234567,150.00);
        if(sender.getBalance() >= amount){
            receiver.increase(amount);
            sender.decrease(amount);
            return true;
        }else{
            return false;
        }
    }
}
```

Listing 3: Test result.

```
A test cases were executed (0.264 s)
Test case 1 "transfer_default" failed.
Test case 2 "transfer_limitExceeded" failed.
Test case 3 "checkBalance_default" passed.
Test case 4 "accountConstructor_default" passed.
```

One of these failing cases is transfer_default, which is specified in the script in Listing 4.

Listing 4: Test case 1 transfer_default.

```java
@Test
public void transfer_default () {
    BankAccount accountS = new BankAccount(00234573201, 2000.00);
    BankAccount accountR = new BankAccount(00173948725, 1500.00);
    Transaction trans = new Transaction(accountS, accountR);
    assertTrue(trans.transfer(300.00));
    assertEquals(accountS.getBalance(), 1700);
    assertEquals(accountR.getBalance(), 1800);
}
```

Task For the exercises, the user is confronted with the task presented in Listing 5 (also see ① in Fig. 9).

Listing 5: Task description.

```
Modify the given code example so that all tests pass and the given design (as-is) matches the targeted design (to-be).
In particular, this includes the following modifications:
(1) Modify class "Transaction" according to the following aspects
(a) Add attributes "sender" and "receiver" both typed by class "BankAccount" to class "Transaction".
(b) Add a parametrized constructor with parameters "sender" and "receiver".
(c) Modify method transfer to only comprise one parameter (amount of type Double).
(2) Add a class "CheckingAccount", which should be a subclass of "BankAccount".
```
Design Diagrams. In the course of the exercise (after each code modification), the user can review the diagrams reflecting the actual design derived from code and compare them with the targeted design diagrams (see Fig. 10; also see (6) and (7) in Fig. 9).

Figure 10: Contrasting as-is (automatically derived from source code; left-hand side) and to-be software design (targeted design specified by the instructor; right-hand side) both represented as UML class diagrams.

6 RELATED WORK

Related work can be roughly divided into the following two groups: (1) interactive tutoring systems for programming, especially those leveraging program visualization (in terms of UML diagrams) and (2) approaches for teaching refactoring, especially with focus on software design.

6.1 Tutoring Systems for Programming

Interactive learning environments in terms of editor-based web-applications such as Codecademy (Sims and Bubinski, 2018) are popular nowadays for learning programming. These tutoring systems provide learning paths for accomplishing practical competences in selected programming aspects. They motivate learners via rewards and document the achieved learning progress. Only very few tutoring systems can be identified that address software refactoring; see, e.g., (Sorva et al., 2011). In particular, Sandalski et al. present an analysis assistant that provides intelligent decision-support and feedback very similar to a refactoring recommendation system, see, e.g., (Tsantalis et al., 2008). It identifies and highlights simple code-smell candidates. After the user’s code modification, it reacts by proposing (better) refactoring options.

Related are also tutoring environments that present interactive feedback in terms of code and design visualization; for an overview, see, e.g., (Sorva et al., 2013). Only a few of these approaches provide the reverse-engineering of UML diagrams, such as JAVAVIS (Oechsle and Schmitt, 2002) or BlueJ (Kölling et al., 2003). However, existing approaches do not target refactoring exercises. For instance, they not allow for comparing the actual design (as-is) and the targeted design (to-be), e.g., as defined by the instructor, especially not in terms of UML class diagrams. Moreover, tutoring systems barely provide integrated software-behavior evaluation in terms of regression tests (pre-specified by the instructor).

We complement these approaches by presenting an approach that integrates interactive feedback on code modifications in terms of software-design quality and software behavior.

6.2 Approaches for Teaching Refactoring

Besides tutoring systems (discussed above) other kinds of teaching refactoring are related to our approach. In addition to tutoring systems based on instructional learning design, a few other learning approaches in terms of editor-based refactoring games can be identified that also integrate automated feedback. In contrast to tutoring systems which normally apply small examples, serious games such as (Elezi et al., 2016; Haendler and Neumann, 2019) are based on real-world code base. These approaches also include means for rewarding successful refactorings by increasing the game score (or reducing the technical-debt score) and partially provide competitive and/or collaborative game variants. However, so far these games do not include visual feedback that is particularly important for the training of design-related refactoring.

Moreover other approaches without integrated automated feedback are established. For instance, Smith et al. propose an incremental approach for teaching different refactoring types on college level in terms of learning lessons (Smith et al., 2006; Stoecklin et al., 2007). The tasks are also designed in an instructional way with exemplary solutions that have to be transferred to the current synthetic refactoring candidate (i.e. code smell). Within this, they provide an exemplary learning path for refactorings. Furthermore, Abid et al. conducted an experiment for contrasting two students groups, of which one performed pre-enhancement (refactoring first, then code extension) and the second post-enhancement (code extension first, then refactoring) (Abid et al., 2015).
Then they compared the quality of the resulting code. López et al. report on a study for teaching refactoring (López et al., 2014). They propose exemplary refactoring task categories and classify them according to Bloom’s taxonomy and learning types. They also describe learning settings, which aim at simulating real-world conditions by providing, e.g., an IDE and revision control systems.

In addition to these teaching and training approaches, we present a tutoring system that can be seen as a link in the learning path between lectures and lessons (that mediate basic knowledge on refactoring) on the one hand and environments such as serious games that already demand practical competences on the other.

7 DISCUSSION

In this paper, a new approach has been presented for teaching and training practical competences in software refactoring. As shown above (Section 6), the tutoring system is considered as complement to other training environments and techniques, such as lectures for mediating basic understanding (before) and serious games (afterwards) for consolidating and extending the practical competences in direction of higher-level competences such as evaluating refactoring options and developing refactoring strategies. The provided decision support in terms of UML design diagrams (representing the as-is and to-be software design) especially addresses the refactoring of quality issues on the level of software design and architecture, which are not directly visible by reviewing the source code alone; also see architectural debt (Kruchten et al., 2012).

The feasibility of the proposed approach has been illustrated in two respects. On the one hand, in terms of the usability for training practical competences in software refactoring via two exemplary application scenarios, i.e. design refactoring and test-driven development, for which both the exercise workflows and other aspects have been described. On the other hand, by demonstrating the technical feasibility via a proof-of-concept implementation in Java that realizes the core functions. In this course, also a more detailed exercise example has been presented.

As a next step, it is necessary to investigate the appropriateness of the approach in an actual training setting (empirical evaluation) and gather feedback from system users. From a didactic perspective, a challenge in creating a training unit is to put together a set of appropriate refactoring exercises consisting of code fragments that manifest as smells in corresponding UML diagrams. An orientation for this can be seen in rules and techniques for identifying and refactoring smells on the level of software design (Suryanarayana et al., 2014). In addition, in (Haendler, 2018) we explored how 14 kinds of software-design smells can be identified (and represented) in UML class and sequence diagrams. This catalog can serve as a systematic guideline for creating exercises targeted on different design smells. For applying the prototype in a training setting, also further technical extensions and refinements are planned; for example, by including views for assessment and progress management (e.g., gamification elements such as leaderboards or activity charts).

8 CONCLUSION

In this paper, we presented a novel approach for teaching software refactoring, especially with focus software design issues, via an interactive development and learning environment that provides feedback to the user in terms of reverse-engineered UML diagrams, test behavior and an extensible array of quality aspects. The paper contributes to the state-of-the-art via the following aspects by:

- presenting a training environment that supports users, e.g., university students or (more) experienced software developers, in accomplishing practical competences for software refactoring;
- illustrating exemplary application scenarios that show the feasibility of the approach for training practical competences in refactoring and test-driven development;
- providing a web-based software-technical implementation in Java (refacTutor) as proof-of-concept that demonstrates the technical feasibility of the proposed approach. Here also a short exercise example is illustrated in detail;
- specifying a generic concept and a lightweight architecture that integrates already existing analysis tools such as test frameworks, diagram editors (and quality analyzers), which allows one to implement the tutoring system for other programming languages as well.

As already discussed in Section 7, in the next step an empirical evaluation of the prototype implementation will be done. For this purpose, it is planned to integrate a training unit on refactoring for design smells into a university course on software design and modeling. The key questions here are how the training via the tutoring system is perceived by the users and to what extent it supports in acquiring practical refactoring competences.
REFERENCES


