RefacTutor: An Interactive Tutoring System for Software Refactoring

Thorsten Haendler(✉), Gustaf Neumann, and Fiodor Smirnov

Institute for Information Systems and New Media,
Vienna University of Economics and Business (WU Vienna), Vienna, Austria
{thorsten.haendler,gustaf.neumann,fiodor.smirnov}@wu.ac.at

Abstract. While software refactoring is considered important to manage software complexity, it is often perceived as difficult and risky by software developers and thus neglected in practice. In this article, we present refacTutor, an interactive tutoring system for promoting software developers' practical competences in software refactoring. The tutoring system provides immediate feedback to the users regarding the quality of the software design and the functional correctness of the (modified) source code. In particular, 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* UML design diagrams can be pre-specified by the instructor. To demonstrate the technical feasibility of the approach, we provide a browser-based software prototype in *Java* accompanied by a collection of exercise examples. Moreover, we specify a viewpoint model for software refactoring, allocate exercises to competence levels and describe an exemplary path for teaching and training.

Keywords: Intelligent tutoring system · Software refactoring · Software design · Code visualization · Unified Modeling Language (UML2) · Software-engineering education and training · Interactive training environment

1 Introduction

As a result of time pressure in software projects, priority is often given to implementing new features rather than ensuring code quality [40]. In the long run, this leads to *software aging* [51] and increased *technical debt* with the consequence of increased maintenance costs on the software system (i.e. *debt interest*) [36]. As studies found out, these costs for software maintenance often account to 80% or more than 90% of software project costs [16,57]. A popular means for repaying this debt is software refactoring, which aims at improving code quality by restructuring the source code while preserving the external system behavior [21,49]. Several kinds of flaws (such as code smells) can negatively impact code quality and are thus candidates for software refactoring [3]. 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 [21,46,66]. Although refactoring is considered important to guarantee that a system remains maintainable and extensible, it is often neglected in practice due to several barriers, among which are perceived by software developers the difficulty of performing refactoring activities as well as the risk of introducing errors into a previously correctly working software system, and a lack of adequate tool support [67]. These barriers pose challenges with regard to improving refactoring tools as well as promoting the skills of software developers.

In the last years, several tools have been proposed for identifying and assessing bad smells in code via certain metrics and benchmarks mostly based on static analysis (e.g. level of coupling between system elements), for supporting in planning and applying the (sequences of) refactoring techniques, such as *JDeodorant* [69] and *DECOR* [43], as well as for measuring and quantifying the impact of bad smells on software quality and project costs in terms of technical debt, such as *SonarQube* [10] or *JArchitect* [13]. Despite these advantages, the refactoring process is still challenging (also see [27]). For instance, more complex kinds of smells (such as smells on the level of software design and architecture) are covered only moderately by detection tools [17,18]. Moreover, these tools to produce *false positives* [19] (representing constructs intentionally used by the developer with symptoms similar to smells, such as certain design patterns), which need to be assessed and discarded by developers manually. In addition, the decision what and how to refactor also depends on the domain knowledge (e.g. regarding design rationale) or the project schedule for which human expertise is required such as provided by software architects and project managers [52]. Due to these issues, the refactoring process is still mostly performed without tool support [45] and thus requires developers to be competent. In turn, 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. [21], there are only a few approaches (see Sect. 5) that aim at mediating or improving software developers' practical and higher-level competences such as *application*, *analysis* and *evaluation* according to Bloom's taxonomy of cognitive learning objectives; see, e.g. [8,35].

In the conference paper presented at *CSEDU-2019* [30], we have proposed an approach for an interactive learning environment for mediating and improving practical competences in software refactoring. The basic idea of the developed tutoring system is to foster active learning by providing instant feedback to user activities (cf. [9,22]). In particular, the tutoring system provides immediate feedback and decision-support to the user regarding relevant aspects for software refactoring, i.e. both the quality of the software design as well as the functional correctness of the source code modified by the user (see Fig. 1). After each refactoring step, the user can review the results of run-time regression tests (e.g.

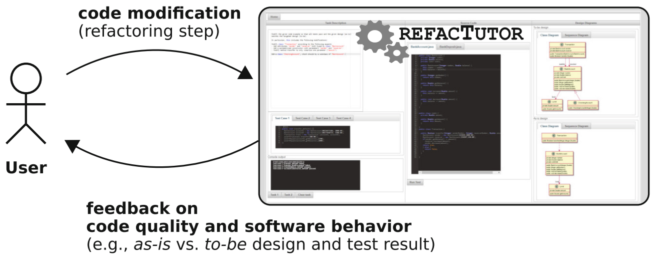

code modification
(refactoring step)
User
REFACTUTOR
feedback on
code quality and software behavior
(e.g., *as-is* vs. *to-be* design and test result)

Fig. 1. Exercise interaction supported by REFACTUTOR [30].

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) [47] 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) [5,25,56]. 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. [11,24]. In general, so the approach aims to support learners to actively apply knowledge (previously consumed on conceptual level) and to reflect their own problem-solving skills [42].

This post-conference revision of the *CSEDU-2019* publication [30] incorporates important extensions, also in response to feedback by reviewers and conference attendees. In particular, this article includes the following additional contributions:

(1) We elaborate on the conceptual foundations of the tutoring system, e.g. by specifying a viewpoint model for software refactoring (see Sect. 2).
(2) We provide a set of examples for refactoring exercises (available for download; e.g. to be used with the developed software prototype; see Sect. 3).
(3) We analyze and allocate the competence levels (in terms of prerequisite and target competences) addressed by training exercises according to Bloom's revised taxonomy of educational objectives (see Sect. 4).
(4) We extend the discussion of related work, e.g. by including recently published approaches (see Sect. 5).

The applicability of the proposed tutoring system is demonstrated in the following two ways. On the one hand, we introduce a proof-of-concept implementation in *Java*[1] in terms of a browser-based development environment. Its

[1] See Sect. 3. The software prototype is available for download from http://refactoringgames.com/refactutor.

functionality is illustrated via a detailed exercise example. On the other hand, we reflect on the usability for teaching and training refactoring. For this purpose, we describe exemplary types of exercises, allocate addressed prerequisite and target competences and sketch a path for learning and training software refactoring.

Structure. The remainder of this article is structured as follows. In Sect. 2, the applied conceptual framework of the interactive tutoring system (including interactive learning and the applied views for software refactoring) is elaborated in detail. In Sect. 3, we introduce refacTutor. In particular, after an overview of components and activities, we present a software prototype in *Java* including a detailed exercise example as well as a collection of further examples. Then, Sect. 4 illustrates the usability of the tutoring system in the learning context, i.e. by describing two exercise scenarios (i.e. design refactoring and test-driven development), allocating addressed competence levels and drawing an exemplary learning and training path. In Sect. 5, related approaches (e.g. approaches for teaching software refactoring and tutoring systems) are discussed. Section 6 reflects on limitations and further potential of the approach and Sect. 7 concludes the paper.

2 Interactions in (Training) Software Refactoring

In this section, we describe the applied conceptual framework of the interactive tutoring system in terms of interactive learning (see Sect. 2.1) and views on the software system relevant for software refactoring (see Sect. 2.2).

2.1 Interactive Learning

Active learning focuses on mediating practical competences by actively involving and engaging learners [22]. This way, it strongly differs from teacher-centered approaches, where learners mainly have to consume passively. This active participation, however, requires the learner to reflect on their problem-solving skills and at the same time to apply the knowledge previously imparted. [42]. In general, a challenge can be seen in stimulating learners to engage actively in the learning process. One key to this is to provide intelligent interactions in the form of immediate feedback, which can either be based on social interaction (e.g., with other learners) or computer-aided and automated as provided by intelligent tutoring systems [38,59], which enable the learner to immediately reflect on and, if necessary, improve the consequences and effects of a performed action. Figure 2 depicts a generic cyclic workflow of decision making including double-loop learning (left-hand side) [4,27] applied to software refactoring exercises (right-hand side).[2] The exercise interaction represents a cycle of analysis and actions performed by the user which is built on immediate feedback provided by the tutoring system. Driven by the specific task (*problem recognition*) and

[2] This workflow is supported by refacTutor; see Sect. 3 and Fig. 4 in particular.

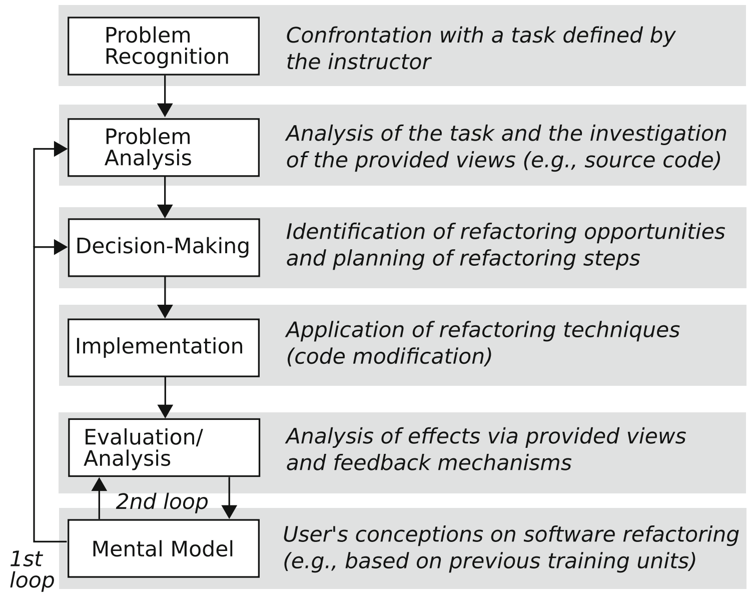

Problem Recognition
Confrontation with a task defined by the instructor
Problem Analysis
Analysis of the task and the investigation of the provided views (e.g., source code)
Decision-Making
Identification of refactoring opportunities and planning of refactoring steps
Implementation
Application of refactoring techniques (code modification)
Evaluation/ Analysis
Analysis of effects via provided views and feedback mechanisms
2nd loop
Mental Model
User's conceptions on software refactoring (e.g., based on previous training units)
1st loop

Fig. 2. Generic cycle for decision making including double-loop learning (left-hand side; see [4,27]) applied to refactoring exercises (right-hand side; oriented to [30]).

based on the user's *mental model*, the user analyses the task and investigates the provided views on the software system. After *analysis*, the user identifies options and plans for refactoring (*decision making*). In this course, the user selects a certain option for code modification and plans the particular steps for performing them (*implementation*). After the code has been modified, the user analyzes the feedback given by the tutoring system (*evaluation*). This workflow supports a double-loop learning [4]. Through previous learning and practice experiences, the users have built a certain mental model [11] on performing software refactoring, on which the analysis and the decisions for planned modifications are built (*first loop*), e.g. how to modify the code structure in order to remove the code or design smell. After modification, the feedback (e.g. in terms of test result or the delta of the *as-is* and the *to-be* design diagrams) impacts the user's mental model and allows for scrutinizing and adapting the decision-making rules (*second loop*). The following section investigates the views and feedback mechanisms relevant to software refactoring in particular.

2.2 Views on Software Refactoring

Here we investigate the refactoring process as well as the relevant feedback for decision-making and learning in software refactoring.

Refactoring Process. Performing software refactoring is generally driven by the goal to improve the internal quality of the source code, e.g. regarding its maintainability or extensibility [49]. For this purpose, the code is modified while the external behavior of the system should remain unaffected. A typical risk, which often prevents refactoring activities, is to introduce errors into a correctly working software system [45]. However, the process of software refactoring can be roughly divided into the two following activities (see [27]):

(1) Identifying and assessing opportunities for refactoring (such as bad code smells and other technical-debt items [36]), and
(2) Planning and applying refactoring techniques to remove the bad smells [21].

In the following, we focus on the actual process of removing the smelly structures by modifying the source code (i.e. activity 2). During software refactoring, the system under analysis (SUA) can take different system states (see top diagram in Fig. 3). The states can be classified into the *Initial State* (before refactoring), *Transitional States* (during the refactoring) and a *Final State(s)* (representing the system after refactoring). In particular, the *Initial State* represents the software system before refactoring, i.e. with correct functionality (tests passing), but including a certain identified refactoring opportunity (bad smell), such as a `CyclicDependency` [21,66] (see Fig. 3). In order to remove this smell, multiple refactoring options exist, which also depend on the characteristics of the concrete smell instance. In general, as already mentioned above, for more complex bad smells multiple options and sequences for achieving a desired design can exist, such as for some types of software-design smells [66]. 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 by applying several refactoring techniques (steps) then draws a concrete path through this graph. However, in [66] the following four options are distinguished for removing a `CyclicDependency` smell:

(a) `ExtractClass` in terms of introducing a new class containing the methods and field that introduce the dependency.
(b) Multiple `MoveMethod` (and/or `MoveField`) refactorings to move the methods or fields that introduce the cyclic dependency to one of the other participating classes.
(c) `InlineClass` to merge the two dependent classes into one class.
(d) Multiple `ExtractMethod` and `MoveMethod` (as refinement to (a)) to remove only the code fragment from including methods that introduces the dependency and then move it accordingly.

For further details on the refactoring techniques or bad smells, please see e.g. [21,66]. After each code modification (e.g. `MoveMethod` refactoring), the software system changes its state (see Fig. 3). Besides the impact on the smelly code structure, the code modification can also introduce errors resulting in failing tests. However, every actual or possible modification to be performed on a certain state then leads to a certain following state. In case that it is planned to apply further refactoring techniques, this state is described as a *Transitional State* (see e.g. the bottom row in Fig. 3). After the planned modifications are performed, the *Final State* of the system is reached. In particular, we distinguish between failing (`F`) and passing (`P`) final states. Failing states represent states that do not conform to other quality requirements (especially functional requirements; i.e. the tests fail). Passing states (`P`), in turn, fulfill the defined quality standards. For example, when performing the `InlineClass` refactoring for removing a `CyclicDependency` (see third row in top diagram in Fig. 3), the methods calling the moved methods or fields of the merged classes might fail addressing the target. In order to pass the tests, these dependencies need to be updated by applying another code modification (e.g. adapting the target class).

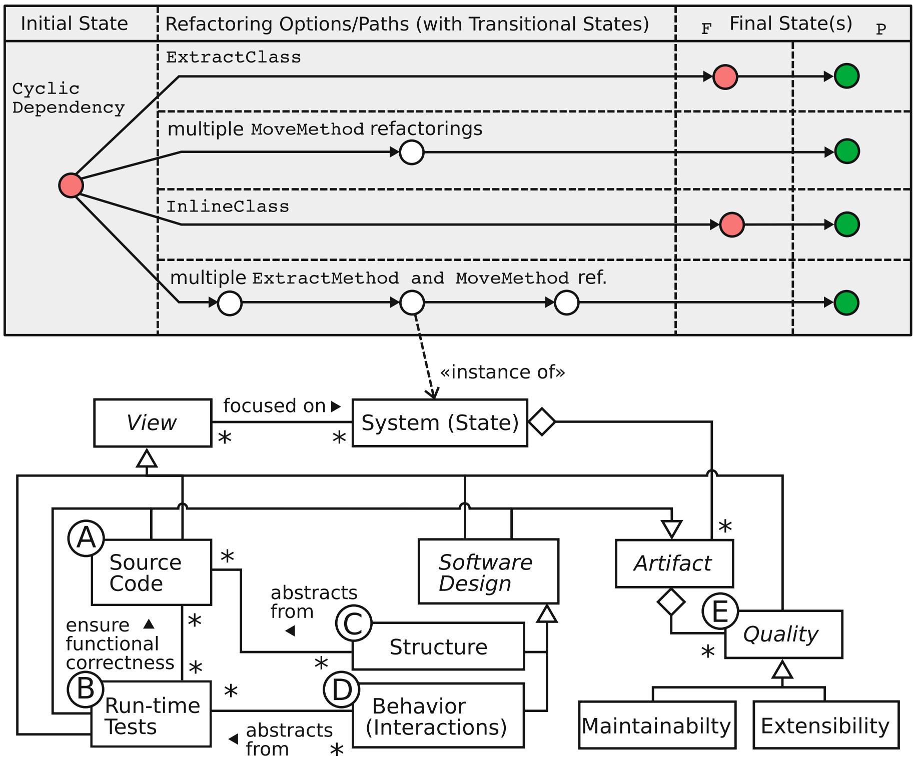


Fig. 3. Options and paths of refactoring an exemplary bad smell (i.e. `CyclicDependency` (top; based on the refactoring options described in [66]) and the applied viewpoint model for software refactoring represented as a UML class diagram (bottom; extending the viewpoint model in [30]).

A Viewpoint Model. In addition to the procedural aspects reflected above, certain views on (the states of) a software system are important for software refactoring, which can be derived from the definition of software refactoring;

i.e. refactoring as a restructuring of the *source code* driven by the objective to improve the system's internal *quality* (e.g. in terms of maintainability or extensibility) without changing the *external system's behavior* [21,49]. Oriented to this definition, certain views on the software system can be considered relevant for software refactoring, which can be structured in a viewpoint model. Viewpoint models (also regarded as view models or viewpoint frameworks [41]) are means to structure a set of coherent (often concurrent) views on a software system. Each view represents the system from the perspective of one or multiple concerns held by stakeholders of the system [12]. They are especially popular in software architecture for constructing and maintaining a software system [37,61]. In addition, viewpoint models are also applied in other activities related to software engineering such as requirements engineering [14,63] and process management and modeling [62].

Oriented to and as a complement to these viewpoint models, we propose and apply a *viewpoint model for software refactoring* that includes the following five views on (states of) a software system: *source code*, *run-time tests*, *structural software design* and *behavioral software design* as well as *artifact quality*. The bottom diagram in Fig. 3 (see above) depicts the structure of the views and the relations between them in terms of viewpoint model represented as a class diagram of the Unified Modeling Language [47]. The included views with motivating concerns (and optionally analysis tools to obtain the views) are described below (also see Fig. 3).

- Ⓐ The `Source Code` represents the basis for other views and is the target of refactoring activities, i.e. it is analyzed and modified by the software developer.
- Ⓑ `Run-Time Tests` are a popular means to specify the functional requirements for the source code. Thus, they are often used in software projects in terms of *regression tests* to ensure the functional correctness of the source code, i.e. that no error has been introduced by the code modifications. For automating the tests certain test frameworks such as *XUnit* or scenario-based tests are available. Feedback is then returned to the user in terms of the test result.
- Ⓒ `Design Structure` reflects the structure of the source code. A popular means for documenting the design structure are UML class diagrams [47], which can either be created manually or derived (a.k.a. reverse-engineered) automatically by using tools or techniques based on static-code analysis (cf. [70]). The diagrams can support software engineers (and other stakeholders) in getting a better overview of the logical structure of the software system. As studies found out, UML design diagrams are especially beneficial for understanding software design (issues) [5,25,56]. In general, the diagrams can either represent the *as-is* design (reflecting the current state) or the *to-be* design (defining an intended or targeted software design).
- Ⓓ `Design Behavior` represents behavioral aspects of the software design (e.g. interactions or processes). Here, also UML diagrams represent the *de-facto* standard for specification. For describing interactions on a level of

software design, UML sequence diagrams [47] are popular. For automatically deriving (a.k.a. reverse-engineering) the behavioral design diagrams representing the *as-is* software design of the source code, (automatic) static and dynamic analysis techniques can be leveraged; see, e.g. [31,34,53]. 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 helpful [25].

Ⓔ Orthogonally to the views above, which also represent concrete artifacts, the `Quality` view reflects on further quality aspects such as the `Maintainability` and the `Extensibility` of the software artifacts, which can e.g. concretely manifest via the level of code complexity. For instance, analysis tools such as *software-quality analyzers* (e.g. *SonarQube* [10]) apply certain code metrics to measure the software quality. Thus, they can also be applied to identify quality issues (such as smells) in the source code or software design. Moreover, these metrics can be used to measure and quantify the technical debt score [36] (e.g. as person hours required to repay the debt).

The proposed tutoring system provides these views conforming to the viewpoint model depicted in Fig. 3 (bottom) as immediate feedback (and decision support) to code modifications performed by the user. Further details on the feedback mechanisms and automatic-analysis techniques are provided in the next Sect. 3.

3 RefacTutor

In this section, we introduce our tutoring system REFACTUTOR. At first, a conceptual overview of components and activities is given (see Sect. 3.1). Then we present a software prototype in *Java* and detail on used technologies, diagram-derivation techniques and GUI and also illustrate a detailed exercise example (see Sect. 3.2). Finally, we provide a collection of exercise examples, which are available for download (see Sect. 3.3).

3.1 Conceptual Overview of REFACTUTOR

Figure 4 depicts a conceptual overview of the tutoring system in terms of the technical components and artifacts as well as of the activities performed by instructor and user. At first, the instructor prepares the exercise. For this purpose, she specifies a task, provides (or re-uses existing) source code and test script, and specifies the expected (*to-be*) software design (see step ① in Fig. 4). The user's exercise workflow then starts by analyzing the task description (step ②). After each code modification (step ③), REFACTUTOR provides automated feedback in terms of several kinds of views on the software system (see steps

④ to ⑥) conforming to the proposed viewpoint model specified in Fig. 3. For this purpose, REFACTUTOR integrates several analysis tools, which automatically analyze the modified source code and produce corresponding views. In particular, these analyzers consist of a test framework for checking the functional correctness (providing the test result), a quality analyzer that examines the code using pre-configured quality metrics (providing hints on quality issues), and a diagram builder that automatically derives (reverse-engineers) diagrams from source code and reflects the current *as-is* software design. This way, the tutoring system supports the cyclic-exercise workflow as defined in Fig. 2 by providing immediate feedback on user's code modifications (see steps ③ to ⑥).

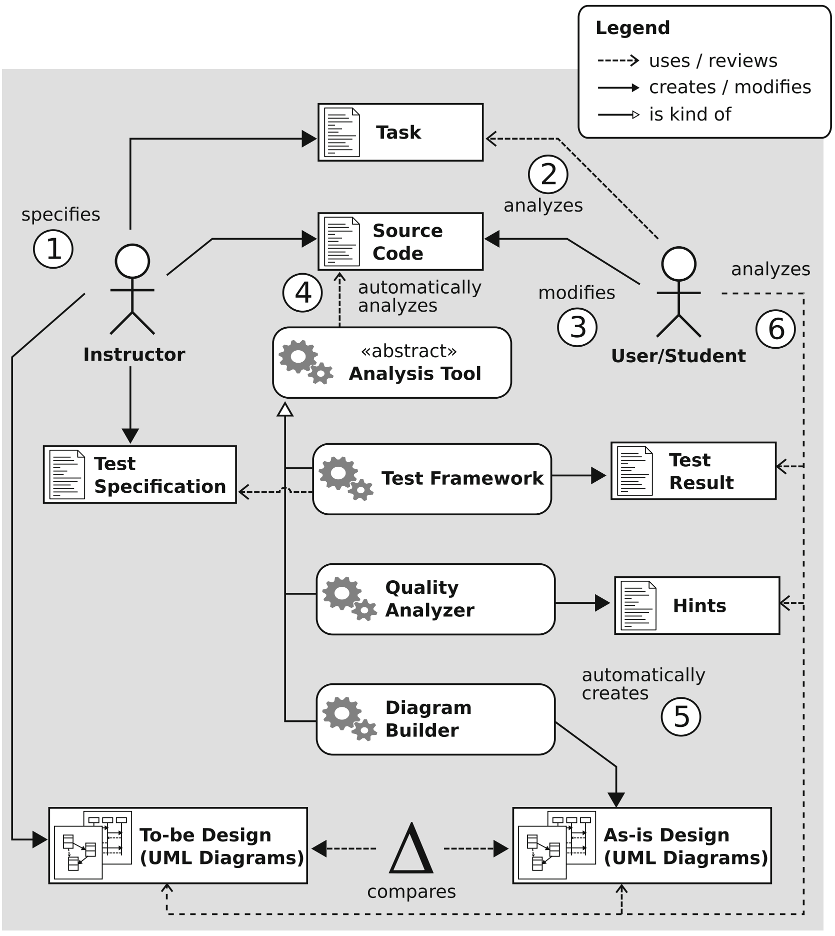


Fig. 4. Overview of components and activities provided by REFACTUTOR [30].

3.2 Software Prototype

In order to demonstrate the technical feasibility, we introduce a prototype implementation of refacTutor[3] that realizes the key aspects of the infrastructure presented in Fig. 4. In the following, the used technologies, the derivation of UML class diagrams reflecting the *as-is* design, the graphical user interface (GUI), and an exercise example are explained in detail.

Used Technologies. We implemented a software prototype as a browser-based application using *Java Enterprise Edition (EE)*, where the source code can be modified by a client without the need of installing any additional software locally. *Java* is also the supported language for the software-refactoring exercises. As editor for code modification the *Ace Editor* [2] has been integrated, which is a flexible browser-based code editor written in *JavaScript*. After each test run, the entered code is then passed to the server for further processing. For compiling the *Java* code, we applied *InMemoryJavaCompiler* [68], 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* [20] in order to extract information on the ode structure. The correct behavior is verified via *JUnit* tests [23]. Relevant exercise information is stored in *XML* files including task description, source code, test script, and information on UML diagrams (see below).

Derivation of Design Diagrams. For automatically creating the *as-is* design diagrams, the extracted information is transferred to an integrated diagram editor. *PlantUML* [54] 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), which then can be exported as *PNG* or *SVG* files. *PlantUML* also already manages the efficient and aesthetic composition of diagram elements. Listing 1 presents exemplary source code to be refactored. The corresponding *as-is* design diagram (reflecting the current state of code) in terms of a **UML class diagrams** is depicted in Fig. 5. In general, the automatically 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. 5). *Generalizations* represent *is-a* relationships (between subclasses and super-classes); see ②) in Fig. 5 and line 11 in Listing 1. Moreover, the derived class diagrams also provide *dependencies* between classes (see ③). Dependencies can express as call or usage dependencies representing inter-class calls of attributes or methods from within a method of another class (see lines 19 and 20 in the listing in Fig. 5). For deriving **UML sequence diagrams**, we apply dynamic reverse-engineering techniques based on the execution traces triggered by the run-time tests, already described and demonstrated e.g. in [31,32]. As already explained above (see Sect. 2.2), the identification and assessment of

[3] The software prototype is available for download from http://refactoringgames.com/refactutor.

issues in software design or architecture can be improved by consulting a combination of UML class and sequence diagrams [25].

Listing 1. Exemplary Java code fragment for a refactoring task.

```java
public class BankAccount {
  private Integer number;
  private Double balance;
  private Limit limit; //(1)
  public BankAccount(Integer number
      , Double balance) {
    this.number = number;
    this.balance = balance;
  }
  [...]
}
public class CheckingAccount
     extends BankAccount { //(2)
  [...]
}
public class Limit {
  [...]
}
public class Transaction {
  public Boolean transfer(Integer
      senderNumber, Integer
      receiverNumber, Double
      amount) {
    BankAccount sender = new
        BankAccount(123456); //(3)
    BankAccount receiver = new
        BankAccount(234567);
    if(sender.getBalance() >=
        amount){
      receiver.increase(amount);
      sender.decrease(amount);
      return true;
    } else {return false;}
  }
}
```

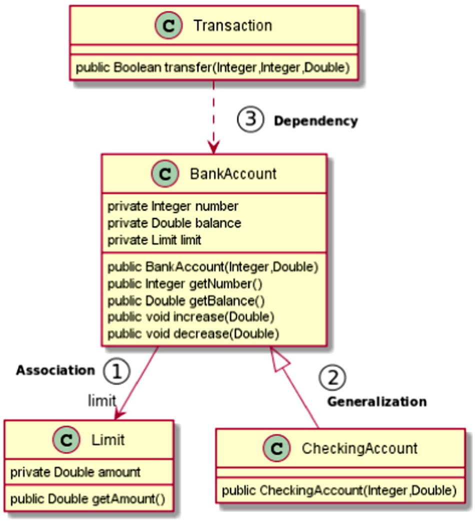

C Transaction
public Boolean transfer(Integer,Integer,Double)
③ Dependency
C BankAccount
private Integer number
private Double balance
private Limit limit
public BankAccount(Integer,Double)
public Integer getNumber()
public Double getBalance()
public void increase(Double)
public void decrease(Double)
Association ①
limit
② Generalization
C Limit
private Double amount
public Double getAmount()
C CheckingAccount
public CheckingAccount(Integer,Double)

Fig. 5. Exemplary UML class diagram representing the *as-is* design automatically derived from source code (in Listing 1) and visualized using PlantUML.

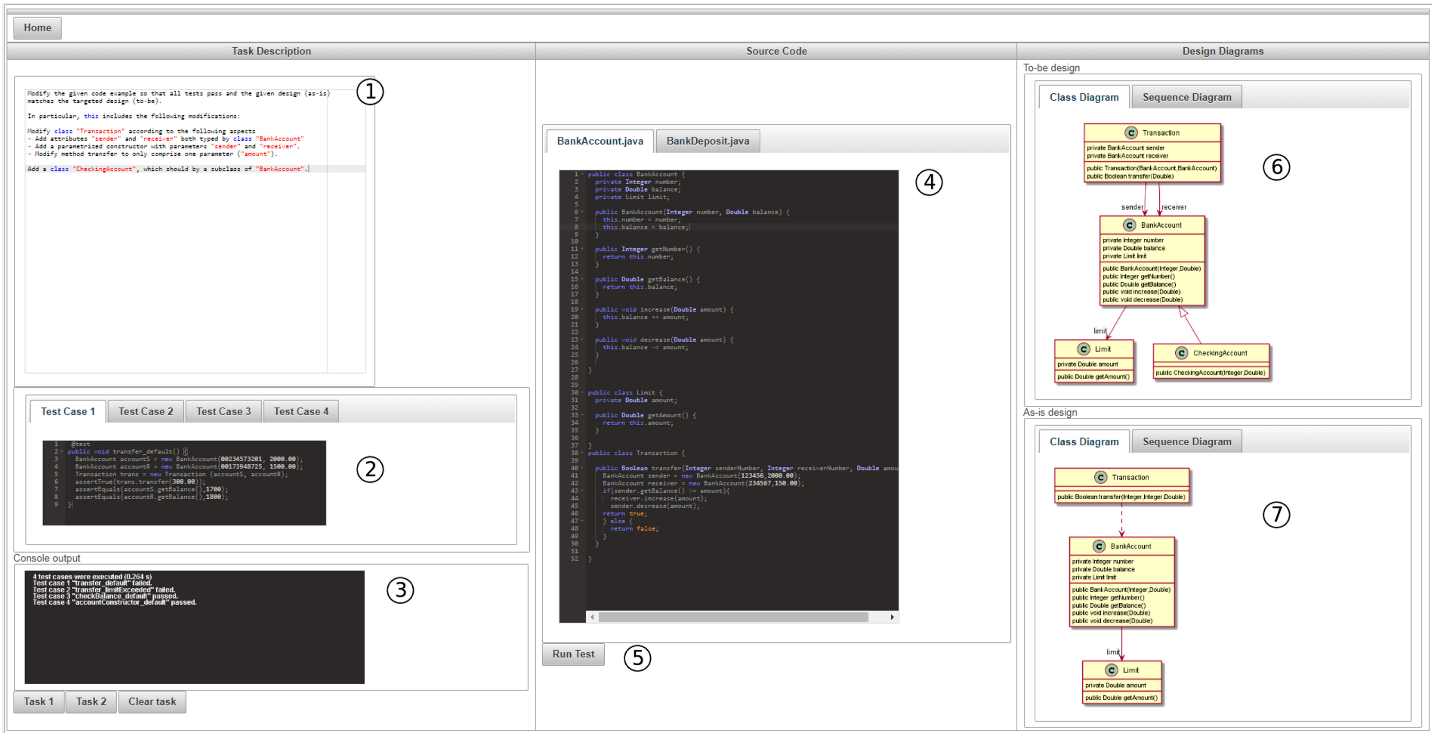
Graphical User Interface. As outlined in the overview depicted in Fig. 4, the prototype provides GUI perspectives both for instructors to configure and prepare exercises and for users to actually perform the exercises. Each role has a specific browser-based dashboard with multiple views (a.k.a. widgets) on different artifacts (as described in Fig. 4). Figure 6 depicts the user's perspective with provided views and an exercise example (which is detailed in the following Sect. 3.2). In particular, the user's perspective comprises a view on the task description (as defined by the instructor; see ① in Fig. 6). 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 as prepared by the instructor or provided by additional analysis tools, e.g. technical-debt analyzers [10]). The code editor (see ④ in Fig. 6) provides the source code to be modified by the user (with tabs for multiple source files to deal with larger source-code examples). Via the button in ⑤, the user can trigger the execution of the run-time 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 respectively 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 to the user's perspective, it provides an editor for specifying the *to-be* diagrams as well as means to save or load exercises.

Exercise Example. In the following, a example of a refactoring exercise applying the REFACTUTOR prototype is illustrated in detail. In particular, the source code, run-time tests, the task to be performed by the user and *to-be* design diagrams (as prepared or reused by the instructor) as well as the the *as-is* 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. 6).

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;}
  }
}
```


Home
Task Description
Source Code
Design Diagrams
①
Test Case 1
Test Case 2
Test Case 3
Test Case 4
②
Console output
③
Task 1
Task 2
Clear task
BankAccount.java
BankDeposit.java
④
Run Test
⑤
To-be design
Class Diagram
Sequence Diagram
⑥
Transaction
private BankAccount sender
private BankAccount receiver
public Transaction(BankAccount,BankAccount)
public Boolean transfer(Double)
sender
receiver
BankAccount
private Integer number
private Double balance
private Limit limit
public BankAccount(Integer,Double)
public Integer getNumber()
public Double getBalance()
public void increase(Double)
public void decrease(Double)
limit
Limit
private Double amount
public Double getAmount()
CheckingAccount
public CheckingAccount(Integer,Double)
As-is design
Class Diagram
Sequence Diagram
⑦
Transaction
public Boolean transfer(Integer,Integer,Double)
BankAccount
private Integer number
private Double balance
private Limit limit
public BankAccount(Integer,Double)
public Integer getNumber()
public Double getBalance()
public void increase(Double)
public void decrease(Double)
limit
Limit
private Double amount
public Double getAmount()

```
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:

Modify class "Transaction" according to the following aspects
- Add attributes "sender" and "receiver" both typed by class "BankAccount"
- Add a parametrized constructor with parameters "sender" and "receiver".
- Modify method transfer to only comprise one parameter ("amount").

Add a class "CheckingAccount", which should by a subclass of "BankAccount".
```

```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);
}
```

```
4 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.
```

```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 amou
    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;
    }
  }

}
```

Fig. 6. Screenshot of user's perspective with views provided by REFACTUTOR [30].

Tests. In this example, four *JUnit* test cases have been specified by the instructor (see Listing 3 and also the views ② and ③ in Fig. 6). Two out of these cases fail at the beginning (*initial state*).

Listing 3. Test result.

```
4 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, for example.

Listing 4. Test case 1 *transfer_ default*.

```
@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 this exemplary exercise, the user is confronted with the task presented in Listing 5 (also see ① in Fig. 6).

Listing 5. Task description.

```
Modify the given code example so that all tests pass and the given design (\
     textit{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 (i.e. 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. 7; also see ⑥ and ⑦ in Fig. 6).

3.3 Example Collection

From a didactic perspective, instructors are faced with the challenge in the process of planning a training unit to put together appropriate refactoring exercises consisting of code fragments that manifest as smells in corresponding UML diagrams. For this purpose, we have prepared a set of exercise examples in *Java*,

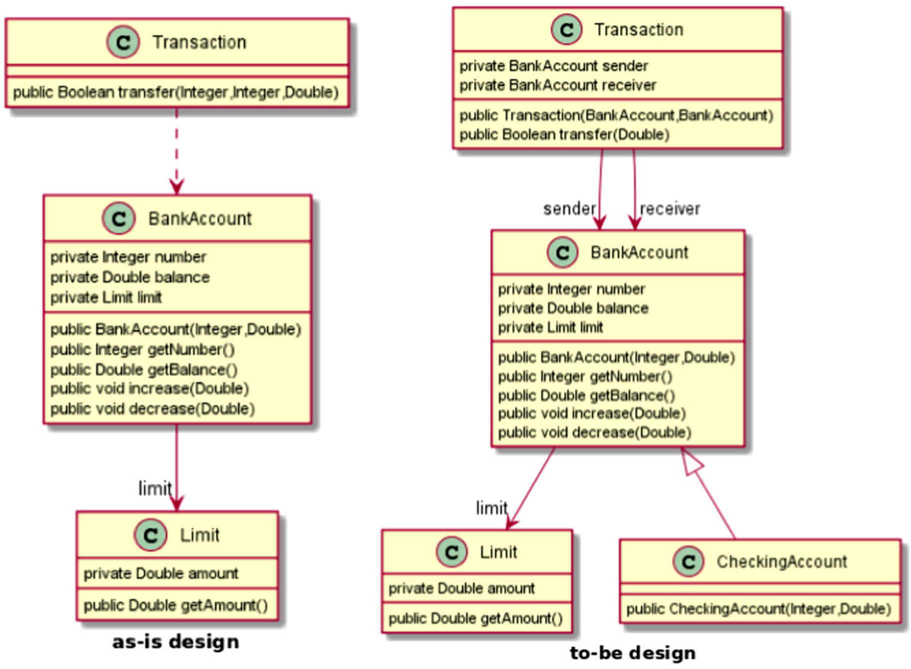

Transaction
public Boolean transfer(Integer,Integer,Double)
BankAccount
private Integer number
private Double balance
private Limit limit
public BankAccount(Integer,Double)
public Integer getNumber()
public Double getBalance()
public void increase(Double)
public void decrease(Double)
limit
Limit
private Double amount
public Double getAmount()
as-is design
Transaction
private BankAccount sender
private BankAccount receiver
public Transaction(BankAccount,BankAccount)
public Boolean transfer(Double)
sender
receiver
BankAccount
private Integer number
private Double balance
private Limit limit
public BankAccount(Integer,Double)
public Integer getNumber()
public Double getBalance()
public void increase(Double)
public void decrease(Double)
limit
Limit
private Double amount
public Double getAmount()
CheckingAccount
public CheckingAccount(Integer,Double)
to-be design

Fig. 7. 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.

which can be used with the REFACTUTOR prototype. The examples are available for download from our website[4]. In particular, the examples are oriented to the catalog of software-design smells investigated in [25], where we analyzed how 14 kinds of software-design smells can be identified (and represented) in UML class and sequence diagrams. The provided set of examples can be extended/adapted or combined for creating individual refactoring exercises, e.g. for a certain application domain.

4 Application Scenarios and Addressed Competences

In this section, we illustrate scenarios for applying the tutoring system for teaching and training software refactoring. In particular, two exemplary exercise types are described in detail (see Sect. 4.1). Then, we allocate competences addressed by the exercises to levels and categories according to Bloom's revised taxonomy and draw an exemplary path for teaching and training software refactoring (see Sect. 4.2).

[4] http://refactoringgames.com/refactutor.

4.1 Exemplary Exercise Types

In order to illustrate the applicability of the proposed tutoring system for teaching and training software refactoring, two possible exercise types are described in detail.[5] In Table 1, exemplary tasks with corresponding selected options of views and other exercise aspects are specified for two exercise types, i.e. *(a) design refactoring* and *(b) test-driven development (TDD)*.

Table 1. Exemplary exercise type: (a) design refactoring and (b) test-driven development (extends the exercise specification provided in [30]).

Aspects and Views	*Design Refactoring*	*Test-driven Development (TDD)*
Task	Refactor the given smelly code fragment in order to achieve the defined *to-be* design	Implement the given *to-be* design in code so that also the given run-time tests pass
Prerequisite Competence	*Understanding*, *applying* and *analyzing onceptual knowledge* on refactoring options for given smell types (as well as on notations and meaning of UML diagrams)	*Understanding*, *applying* and *analyzing conceptual knowledge* on refactoring options for given smell type as well (as on notations and meaning of UML diagrams)
Target Competence	*Analyze* the software design and plan & perform refactoring steps for removing a given smell (*applying* and *analyzing procedural knowledge*)	Apply the *red–green-refactor* steps (to realize the defined *to-be* design and test behavior), which includes the *application* and *analysis* of *procedural knowledge*
Code (*initial state*)	Code including software-design smells [66]	No code given
As-is-Design (*initial state*)	Representing software-design smells	No *as-is*-design available
As-is-Design (*passing final state*)	Conforming to *to-be* design	Conforming to *to-be* design
Tests	Tests pass in (*initial state*) and in (*passing final state*)	Tests fail in (*initial state*), but pass in (*passing final state*)
Assessment	*as-is* and *to-be* design are identical and all tests pass	*as-is* and *to-be* design are identical and all tests pass

Design Refactoring. First, consider an instructor who 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 exercise can be realized by presenting a piece of source code that behaves as intended (i.e. run-time 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.

At the beginning (*initial state*), the tests pass, but code and design are smelly by containing, e.g. a `MultifacetedAbstraction` smell [66]. During the (path of) refactorings (e.g. by applying corresponding `ExtractClass` and/or `MoveMethod`/`MoveField` refactorings [21]), the values of the views

[5] Besides the two described types of exercises, multiple other scenarios can be designed by varying the values of the exercise aspects and views (see Table 1).

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.

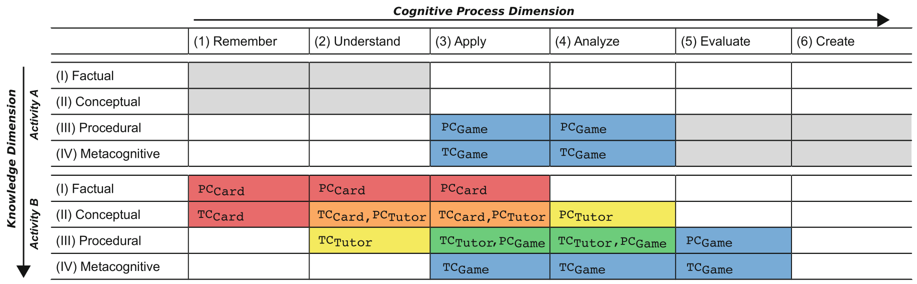
Test-Driven Development. Another exercise type is represented by test-driven development. Consider the situation that the user shall improve her competences in performing the *red–green–refactor* cycle typical for test-driven development [7], which also has been identified as challenging for teaching and training [44]. This cycle includes the activities to specify and run tests that reflect the intended behavior (which first do not pass; *red*), then implement (extend) the code to pass the tests (i.e. *green*), and finally modify the code in order to improve design quality (i.e. *refactor*). For this purpose, corresponding run-time tests and the intended *to-be* design are prepared by the instructor. The user's task then is to realize the intended behavior and design by implementing and modifying the source code.

4.2 Competence-Based Learning Path

In the following, we allocate the competence levels addressed by exercises of selected teaching and training environments and describe an exemplary competence-based path for teaching and training software refactoring.

Addressed Competences. In [28], we have applied Bloom's revised taxonomy of educational objectives [35] to software refactoring. In this course, we have specified the corresponding knowledge categories and cognitive-process levels for the two main activities of software refactoring (i.e. A: identifying refactoring candidates and B: planning and performing refactoring steps; see above), which then allows to precisely allocate the competence levels addressed by refactoring exercises. In particular, the corresponding *prerequisite* (required to perform an activity) and *target competences* (a.k.a. learning objectives; see [50]) can be allocated. Figure 8 depicts a two-dimensional matrix correlating the four knowledge categories (vertical) and six cognitive-process levels (horizontal) according to Bloom's revised taxonomy [35]. For example, the exercise types presented above (see Sect. 4.1) can be classified as follows:

- *Prerequisite Competences*: For both exercises *conceptual knowledge* on applying refactoring techniques is required. In particular: *understanding*, *applying* and *analyzing* the *concepts* for performing refactoring steps (such as refactoring options and paths on a *conceptual level*; i.e. knowledge category II and process levels 2/3/4 for activity B; also see `PC_Tutor` in Fig. 8).
- *Target Competences*: In turn, the aim of the exercises is to mediate *procedural knowledge* on how to actually perform refactoring techniques (on the levels of understanding, application and analysis; i.e. knowledge category III and process levels 2/3/4 for activity B; also see `TC_Tutor` in Fig. 8).



Knowledge Dimension		Cognitive Process Dimension: (1) Remember	(2) Understand	(3) Apply	(4) Analyze	(5) Evaluate	(6) Create
Activity A	(I) Factual						
	(II) Conceptual						
	(III) Procedural			^{PC}Game	^{PC}Game		
	(IV) Metacognitive			^{TC}Game	^{TC}Game		
Activity B	(I) Factual	^{PC}Card	^{PC}Card	^{PC}Card			
	(II) Conceptual	^{TC}Card	^{TC}Card, ^{PC}Tutor	^{TC}Card, ^{PC}Tutor	^{PC}Tutor		
	(III) Procedural		^{TC}Tutor	^{TC}Tutor, ^{PC}Game	^{TC}Tutor, ^{PC}Game	^{PC}Game	
	(IV) Metacognitive			^{TC}Game	^{TC}Game	^{TC}Game	

Fig. 8. Prerequisite (PC) and target competences (TC) of exercises provided by selected teaching and training environments, i.e. a card game (referred as Card) [26], REFACTUTOR (referred as Tutor) and a serious game (referred as Game) [29] allocated to knowledge categories (I–IV; vertical) and cognitive-process levels (1–6; horizontal) of Bloom's revised taxonomy of educational objectives [35] applied to *identification and assessment of candidates* (activity A; top rows) as well as to *planning and applying refactoring techniques* (activity B; bottom rows). For further details on the competence framework, see [28].

Learning and Training Path. In addition to the competences addressed by the tutoring system, the matrix in Fig. 8 also presents the competences addressed by exercises provided by other learning and training environments, i.e. a non-digital card game (referred as Card) [26] as well as a digital serious game (referred as Game) [29]. The allocation of competences shows that by combining the three training environments, a large part of competences for activity B is covered. Moreover, these environments could by used to describe a path from lower level competences addressed by the card game (e.g. understanding of conceptual knowledge) via the tutoring system covering practical aspects of refactoring (e.g. applying procedural knowledge) to more reflexive and higher-level competences (e.g. analysis of meta-cognitive knowledge, such as strategies for refactoring) addressed by the serious game.

In turn, it can also been seen that activity A (i.e. the identification of refactoring opportunities) is barely addressed by these three environments. For this purpose, for example, other environments such as *quizzes* (e.g. with multiple-choice questions) addressing the lower levels of factual and conceptual knowledge for activity A or *capstone projects* [6] focusing on the development of a smell-detection tool to address the levels of evaluation and creation of meta-cognitive knowledge (see the cells colored grey in Fig. 8) could by applied.

5 Related Work

Research related to our tutoring system for software refactoring can be roughly divided into (1) interactive tutoring systems for software development, especially those leveraging program visualization (in terms of UML diagrams) and (2) approaches for teaching software refactoring, especially with focus on software design.

5.1 Tutoring Systems for Software Development

Interactive learning environments in terms of editor-based web-applications such as *Codecademy* [58] 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. [55]. 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. [69]. 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. [64]. Only a few of these approaches provide the reverse-engineering of UML diagrams, such as *JAVAVIS* [48] or *BlueJ* [33]. 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). In addition, Krusche and Seitz propose an approach for a tutoring system providing immediate feedback and applied in the framework of MOOCs for teaching and training software engineering [38]. In particular, the environment visualizes errors by highlighting the affected parts of the code in a reverse-engineered UML class diagram. By providing the results of run-time tests and program visualization as decision support the approach is very similar to ours. However, while it focuses on reporting errors triggered by run-time tests, our approach focuses on supporting the removal of design flaws.

In general, we complement these approaches by presenting an approach that integrates immediate feedback on code modifications in terms of software-design quality and software behavior.

5.2 Teaching Software 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 [15,29] 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 [60,65]. 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) [1]. Then they compared the quality of the resulting code. López et al. report on a study for teaching refactoring [39]. 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 require practical competences on the other.

6 Discussion

In this article, we have presented a novel approach for teaching and training practical competences in software refactoring. As shown above (Sect. 5), the tutoring system is considered as complement to other 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* [36].

The applicability of the proposed approach has been shown in two ways. First, by demonstrating the technical feasibility via a proof-of-concept implementation in *Java*, which realizes the most important functionalities. In this course, also a more detailed exercise example has been presented and a set of further examples is provided. Second, in terms of the usability for teaching and training practical competences in software refactoring. This has been illustrated by two exemplary exercise types (i.e. design refactoring and test-driven development), for which the addressed competence have been allocated according to a competence framework [28]. Moreover, an exemplary path for training software refactoring has been drawn including the exercises provided by the tutoring system.

As a next step, it is important to investigate, how and to what extent the tutoring system can contribute to competence acquisition and training in the framework of an actual educational setting, such as a university course or an open online course. In addition to collecting feedback from users, the challenge here is to measure the competence levels of users and in particular whether and to what extent the levels have risen due to the use of the environment.

7 Conclusion

In this article, we have presented an interactive training and development environment for improving practical competences in software refactoring. refacTutor represents a novel approach for actively learning and training software refactoring. Key to our approach is providing immediate feedback to the user's code modifications (i.e. refactoring steps) on the quality of the software design (in terms of reverse-engineered UML diagrams), the functional correctness of the (modified) source code (via presenting the result of integrated run-time regression tests) as well as an extensible array of other quality aspects (depending on the kind of integrated analysis tools).

In particular, this article extends the previous conference publication [30] by providing an elaboration of the conceptual foundations of the tutoring system (e.g. in terms of the viewpoint model for software refactoring), a set of further examples of refactoring exercises for download as well as a competence-based analysis and an exemplary teaching path.

As already discussed in Sect. 6, an interesting future research direction is to measure, how and to what extent refacTutor can contribute to acquiring and training competences for software refactoring. 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.

References

1. Abid, S., Abdul Basit, H., Arshad, N.: Reflections on teaching refactoring: a tale of two projects. In: Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, pp. 225–230. ACM (2015)
2. Ajax.org: AceEditor (2019). https://ace.c9.io/. Accessed 7 Aug 2019
3. Alves, N.S., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Seaman, C.: Identification and management of technical debt: a systematic mapping study. Inf. Softw. Technol. **70**, 100–121 (2016). https://doi.org/10.1016/j.infsof.2015.10.008
4. Argyris, C.: Double loop learning in organizations. Harvard Bus. Rev. **55**(5), 115–125 (1977)
5. Arisholm, E., Briand, L.C., Hove, S.E., Labiche, Y.: The impact of UML documentation on software maintenance: an experimental evaluation. IEEE Trans. Softw. Eng. **32**(6), 365–381 (2006). https://doi.org/10.1109/TSE.2006.59
6. Bastarrica, M.C., Perovich, D., Samary, M.M.: What can students get from a software engineering capstone course? In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET), pp. 137–145. IEEE (2017)
7. Beck, K.: Test-Driven Development: By Example. Addison-Wesley Professional (2003)

8. Bloom, B.S., et al.: Taxonomy of Educational Objectives, vol. 1: Cognitive Domain, pp. 20–24. McKay, New York (1956)
9. Bonwell, C.C., Eison, J.A.: Active Learning: Creating Excitement in the Classroom. 1991 ASHE-ERIC Higher Education Reports. ERIC (1991)
10. Campbell, G., Papapetrou, P.P.: SonarQube in Action. Manning Publications Co. (2013). https://www.sonarqube.org/. Accessed 7 Aug 2019
11. Cañas, J.J., Bajo, M.T., Gonzalvo, P.: Mental models and computer programming. Int. J. Hum.-Comput. Stud. **40**(5), 795–811 (1994). https://doi.org/10.1006/ijhc.1994.1038
12. Clements, P., et al.: Documenting Software Architectures: Views and Beyond. Pearson Education (2002)
13. CoderGears: JArchitect (2018). http://www.jarchitect.com/. Accessed 7 Aug 2019
14. Daun, M., Tenbergen, B., Weyer, T.: Requirements viewpoint. In: Pohl, K., Hönninger, H., Achatz, R., Broy, M. (eds.) Model-Based Engineering of Embedded Systems, pp. 51–68. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34614-9_4
15. Elezi, L., Sali, S., Demeyer, S., Murgia, A., Pérez, J.: A game of refactoring: studying the impact of gamification in software refactoring. In: Proceedings of the Scientific Workshops of XP2016, pp. 23:1–23:6. ACM (2016). https://doi.org/10.1145/2962695.2962718
16. Erlikh, L.: Leveraging legacy system dollars for e-business. IT Prof. **2**, 17–23 (2000)
17. Fernandes, E., Oliveira, J., Vale, G., Paiva, T., Figueiredo, E.: A review-based comparative study of bad smell detection tools. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, pp. 18:1–18:12. ACM (2016). https://doi.org/10.1145/2915970.2915984
18. Fontana, F.A., Braione, P., Zanoni, M.: Automatic detection of bad smells in code: an experimental assessment. J. Object Technol. **11**(2), 5-1 (2012). https://doi.org/10.5381/jot.2012.11.2.a5
19. Fontana, F.A., Dietrich, J., Walter, B., Yamashita, A., Zanoni, M.: Antipattern and code smell false positives: preliminary conceptualization and classification. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 609–613. IEEE (2016). https://doi.org/10.1109/SANER.2016.84
20. Forman, I.R., Forman, N.: Java Reflection in Action (In Action Series). Manning Publications Co. (2004). https://www.oracle.com/technetwork/articles/java/javareflection-1536171.html. Accessed 7 Aug 2019
21. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (1999). http://martinfowler.com/books/refactoring.html. Accessed 7 Aug 2019
22. Freeman, S., et al.: Active learning increases student performance in science, engineering, and mathematics. Proc. Nat. Acad. Sci. **111**(23), 8410–8415 (2014)
23. Gamma, E., Beck, K., et al.: JUnit: a cook's tour. Java Rep. **4**(5), 27–38 (1999). http://junit.sourceforge.net/doc/cookstour/cookstour.htm. Accessed 7 Aug 2019
24. George, C.E.: Experiences with novices: the importance of graphical representations in supporting mental mode. In: PPIG, p. 3 (2000)
25. Haendler, T.: On using UML diagrams to identify and assess software design smells. In: Proceedings of the 13th International Conference on Software Technologies, pp. 413–421. SciTePress (2018). https://doi.org/10.5220/0006938504470455
26. Haendler, T.: A card game for learning software-refactoring principles. In: Proceedings of the 3rd International Symposium on Gamification and Games for Learning (GamiLearn@CHIPLAY) (2019)

27. Haendler, T., Frysak, J.: Deconstructing the refactoring process from a problem-solving and decision-making perspective. In: Proceedings of the 13th International Conference on Software Technologies (ICSOFT), pp. 363–372. SciTePress (2018). https://doi.org/10.5220/0006915903970406
28. Haendler, T., Neumann, G.: A framework for the assessment and training of software refactoring competences. In: Proceedings of 11th International Conference on Knowledge Management and Information Systems (KMIS). SciTePress (2019)
29. Haendler, T., Neumann, G.: Serious refactoring games. In: Proceedings of the 52nd Hawaii International Conference on System Sciences (HICSS), pp. 7691–7700 (2019). https://doi.org/10.24251/HICSS.2019.927
30. Haendler, T., Neumann, G., Smirnov, F.: An interactive tutoring system for training software refactoring. In: Proceedings of the 11th International Conference on Computer Supported Education (CSEDU), vol. 2, pp. 177–188. SciTePress (2019). https://doi.org/10.5220/0007801101770188
31. Haendler, T., Sobernig, S., Strembeck, M.: Deriving tailored UML interaction models from scenario-based runtime tests. In: Lorenz, P., Cardoso, J., Maciaszek, L.A., van Sinderen, M. (eds.) ICSOFT 2015. CCIS, vol. 586, pp. 326–348. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30142-6_18
32. Haendler, T., Sobernig, S., Strembeck, M.: Towards triaging code-smell candidates via runtime scenarios and method-call dependencies. In: Proceedings of the XP2017 Scientific Workshops, pp. 8:1–8:9. ACM (2017). https://doi.org/10.1145/3120459.3120468
33. Kölling, M., Quig, B., Patterson, A., Rosenberg, J.: The BlueJ system and its pedagogy. Comput. Sci. Educ. **13**(4), 249–268 (2003). https://doi.org/10.1076/csed.13.4.249.17496
34. Kollmann, R., Selonen, P., Stroulia, E., Systa, T., Zundorf, A.: A study on the current state of the art in tool-supported UML-based static reverse engineering. In: Proceedings of the Ninth Working Conference on Reverse Engineering, pp. 22–32. IEEE (2002). https://doi.org/10.1109/WCRE.2002.1173061
35. Krathwohl, D.R.: A revision of Bloom's taxonomy: an overview. Theory Pract. **41**(4), 212–218 (2002). https://doi.org/10.1207/s15430421tip4104_2
36. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: from metaphor to theory and practice. IEEE Softw. **29**(6), 18–21 (2012). https://doi.org/10.1109/MS.2012.167
37. Kruchten, P.B.: The 4+1 view model of architecture. IEEE Softw. **12**(6), 42–50 (1995). https://doi.org/10.1109/52.469759
38. Krusche, S., Seitz, A.: Increasing the interactivity in software engineering MOOCs - a case study. In: 52nd Hawaii International Conference on System Sciences, HICSS 2019, pp. 1–10 (2019)
39. López, C., Alonso, J.M., Marticorena, R., Maudes, J.M.: Design of e-activities for the learning of code refactoring tasks. In: 2014 International Symposium on Computers in Education (SIIE), pp. 35–40. IEEE (2014). https://doi.org/10.1109/SIIE.2014.7017701
40. Martini, A., Bosch, J., Chaudron, M.: Architecture technical debt: understanding causes and a qualitative model. In: 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 85–92. IEEE (2014). https://doi.org/10.1109/SEAA.2014.65
41. May, N.: A survey of software architecture viewpoint models. In: Proceedings of the Sixth Australasian Workshop on Software and System Architectures, pp. 13–24 (2005)
42. Michael, J.: Where's the evidence that active learning works? Adv. Physiol. Educ. **30**(4), 159–167 (2006)

43. Moha, N., Gueheneuc, Y.G., Duchien, L., Le Meur, A.F.: DECOR: a method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. **36**(1), 20–36 (2010). https://doi.org/10.1109/TSE.2009.50
44. Mugridge, R.: Challenges in teaching test driven development. In: Marchesi, M., Succi, G. (eds.) XP 2003. LNCS, vol. 2675, pp. 410–413. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44870-5_63
45. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Trans. Softw. Eng. **38**(1), 5–18 (2012). https://doi.org/10.1109/TSE.2011.41
46. Nord, R.L., Ozkaya, I., Kruchten, P., Gonzalez-Rojas, M.: In search of a metric for managing architectural technical debt. In: 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, pp. 91–100. IEEE (2012). https://doi.org/10.1109/WICSA-ECSA.212.17
47. Object Management Group: Unified Modeling Language (UML), Superstructure, Version 2.5.1, June 2017. https://www.omg.org/spec/UML/2.5.1. Accessed 7 Aug 2019
48. Oechsle, R., Schmitt, T.: JAVAVIS: automatic program visualization with object and sequence diagrams using the Java Debug Interface (JDI). In: Diehl, S. (ed.) Software Visualization. LNCS, vol. 2269, pp. 176–190. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45875-1_14
49. Opdyke, W.F.: Refactoring object-oriented frameworks. University of Illinois at Urbana-Champaign Champaign, IL, USA (1992). https://dl.acm.org/citation.cfm?id=169783
50. Paquette, G.: An ontology and a software framework for competency modeling and management. Educ. Technol. Soc. **10**(3), 1–21 (2007). https://www.jstor.org/stable/jeductechsoci.10.3.1?seq=1
51. Parnas, D.L.: Software aging. In: Proceedings of 16th International Conference on Software Engineering, pp. 279–287. IEEE (1994). http://portal.acm.org/citation.cfm?id=257734.257788
52. Ribeiro, L.F., de Freitas Farias, M.A., Mendonça, M.G., Spínola, R.O.: Decision criteria for the payment of technical debt in software projects: a systematic mapping study. In: ICEIS (1), pp. 572–579 (2016)
53. Richner, T., Ducasse, S.: Recovering high-level views of object-oriented applications from static and dynamic information. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 13–22. IEEE Computer Society (1999). https://doi.org/10.1109/ICSM.1999.792487
54. Roques, A.: PlantUml: UML diagram editor (2017). https://plantuml.com/. Accessed 7 Aug 2019
55. Sandalski, M., Stoyanova-Doycheva, A., Popchev, I., Stoyanov, S.: Development of a refactoring learning environment. Cybern. Inf. Technol. (CIT) **11**(2) (2011). http://www.cit.iit.bas.bg/CIT_2011/v11-2/46-64.pdf. Accessed 7 Aug 2019
56. Scanniello, G., et al.: Do software models based on the UML aid in source-code comprehensibility? Aggregating evidence from 12 controlled experiments. Empirical Softw. Eng. **23**(5), 2695–2733 (2018). https://doi.org/10.1007/s10664-017-9591-4
57. Schach, S.R.: Object-Oriented and Classical Software Engineering, vol. 6. McGraw-Hill, New York (2007)
58. Sims, Z., Bubinski, C.: Codecademy (2018). http://www.codecademy.com. Accessed 7 Aug 2019
59. Sleeman, D., Brown, J.S.: Intelligent tutoring systems (1982)

60. Smith, S., Stoecklin, S., Serino, C.: An innovative approach to teaching refactoring. In: ACM SIGCSE Bulletin, vol. 38, pp. 349–353. ACM (2006). https://doi.org/10.1145/1121341.1121451
61. Software Engineering Standards Committee of the IEEE Computer Society: IEEE recommended practice for architectural description of software-intensive systems. IEEE Std 1471–2000, pp. 1–29, September 2000
62. Sommerville, I., Kotonya, G., Viller, S., Sawyer, P.: Process viewpoints. In: Schäfer, W. (ed.) EWSPT 1995. LNCS, vol. 913, pp. 2–8. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59205-9_35
63. Sommerville, I., Sawyer, P.: Viewpoints: principles, problems and a practical approach to requirements engineering. Ann. Softw. Eng. **3**(1), 101–130 (1997)
64. Sorva, J., Karavirta, V., Malmi, L.: A review of generic program visualization systems for introductory programming education. ACM Trans. Comput. Educ. (TOCE) **13**(4), 15 (2013). https://doi.org/10.1145/2490822
65. Stoecklin, S., Smith, S., Serino, C.: Teaching students to build well formed object-oriented methods through refactoring. ACM SIGCSE Bull. **39**(1), 145–149 (2007). https://doi.org/10.1145/1227310.1227364
66. Suryanarayana, G., Samarthyam, G., Sharma, T.: Refactoring for Software Design Smells: Managing Technical Debt. Morgan Kaufmann (2014). https://dl.acm.org/citation.cfm?id=2755629
67. Tempero, E., Gorschek, T., Angelis, L.: Barriers to refactoring. Commun. ACM **60**(10), 54–61 (2017). https://doi.org/10.1145/3131873
68. Trung, N.K.: InMemoryJavaCompiler (2017). https://github.com/trung/InMemoryJavaCompiler. Accessed 7 Aug 2019
69. Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A.: JDeodorant: identification and removal of type-checking bad smells. In: Proceedings of 12th European Conference on Software Maintenance and Reengineering (CSMR 2008), pp. 329–331. IEEE (2008). https://doi.org/10.1109/CSMR.2008.4493342
70. Wichmann, B., Canning, A., Clutterbuck, D., Winsborrow, L., Ward, N., Marsh, D.: Industrial perspective on static analysis. Softw. Eng. J. **10**(2), 69–75 (1995)