Serious Refactoring Games

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

Abstract

Software design issues can severely impede software development and maintenance. Thus, it is important for the success of software projects that developers are aware of bad smells in code artifacts and improve their skills to reduce these issues via refactoring. However, software refactoring is a complex activity and involves multiple tasks and aspects. Therefore, imparting competences for identifying bad smells and refactoring code efficiently is challenging for software engineering education and training. The approaches proposed for teaching software refactoring in recent years mostly concentrate on small and artificial tasks and fall short in terms of higher level competences, such as analysis and evaluation. In this paper, we investigate the possibilities and challenges of designing serious games for software refactoring on real-world code artifacts. In particular, we propose a game design, where students can compete either against a predefined benchmark (technical debt) or against each other. In addition, we describe a lightweight architecture as the technical foundation for the game design that integrates pre-existing analysis tools such as test frameworks and software-quality analyzers. Finally, we provide an exemplary game scenario to illustrate the application of serious games in a learning setting.

1. Introduction

Refactoring is the process of improving a system's internal technical quality by modifying and restructuring its source code without changing its external behavior [1]. Code smells are programming constructs that indicate a violation of design or coding principles [2]. Anti-patterns, in addition, represent commonly used programming solutions for recurring problems, but are proven to have negative consequences [3]. Both code smells and instances of anti-patterns negatively impact the quality of the software system regarding its maintainability and extensibility and thus are targets of code refactoring. But smells can be found at different levels (e.g., source code, software design and architecture) and also in artifacts other than the source code and its design, such as in the requirements specification, documentation, or test specification [4].

In recent years, several tools and techniques have been proposed to support software developers in refactoring-related tasks. So far, these tools cover only a modest amount of smells and corresponding refactorings (for an overview, see, e.g., [5, 6]). Since smell detection is complex, tools tend to produce false positives [7], which share aspects with bad smells but represent consciously implemented constructs (e.g., certain design patterns). In addition, because refactoring always carries the risk of introducing errors or other digressions into a previously correct artifact, automated refactoring based on these tools is quite limited. Therefore, the task of refactoring demands software developers competent in identifying and assessing refactoring candidates as well as in planning and performing the corresponding refactorings.

Since refactoring requires a deeper understanding of programming artifacts, addressing refactoring in software-engineering education and training is desirable, but triggers as well many challenges on how to impart these competences to novice or even experienced software developers, and how to assess them. Providing students with small and artificial code examples does not transport the underlying concepts in a satisfying manner. In this paper, we investigate the possibilities of applying serious games in order to address these challenges. The basic idea is to confront a learner with larger executable coding artifacts with functionally correct behavior but containing multiple bad smells. The functional correctness is validated by regression tests, and the smell metrics are provided by quality analyzers. In these games, moves of a player are characterized by a transition of a functional correct code to another functional correct code but with smells removed. This approach has the advantage that learners

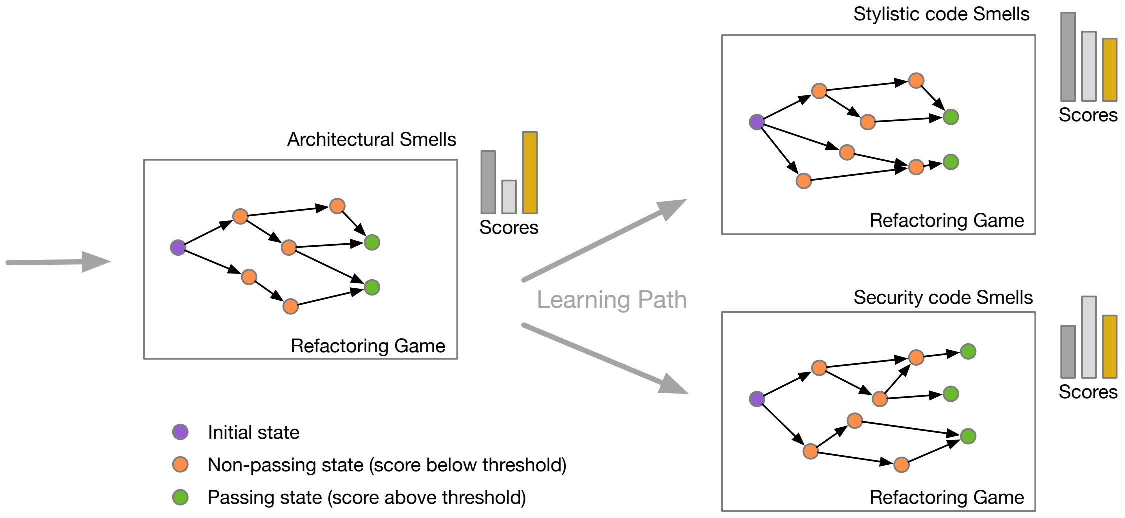


Figure 1. Exemplary learning paths for serious refactoring games distinguished according to smell types.

are confronted with real-world situations, in which they have to analyze the artifacts and to evaluate alternatives, which are higher goals in Bloom's taxonomy [8].

The remainder of this paper is structured as follows. Section 2 describes the background on challenges in the refactoring process for software engineers (Section 2.1) and gives an overview of techniques for game-based learning in software-engineering education (Section 2.2). Section 3 defines refactoring-related competences according to Bloom's taxonomy (Section 3.1) and describes kinds of learning objects in terms of different smell categories (Section 3.2). Section 4 introduces a game design for serious games in refactoring consisting of basic game mechanics (Section 4.1) and presents exemplary game modes (Section 4.2). Section 5 describes a component architecture using pre-existing tools. In Section 6, we provide an exemplary game scenario for a concrete learning setting. Section 7 discusses research related to our approach. Finally, Section 8 reflects on limitations and further potential, and Section 9 concludes the paper.

2. Background

2.1. Challenges in the Software Refactoring Process

In recent years, multiple tools and techniques have been proposed for automatically detecting refactoring candidates (such as bad smells), [6, 5], and for automatically executing the corresponding refactoring steps, see, e.g. [9, 10]. However, these tools only cover a modest amount of smells and refactorings, see, e.g., [6]. For instance, the popular smell detection and refactoring recommendation tools *JDeodorant* [9] and *DECOR* [10] only cover 5 and 9 kinds of smells respectively.[1] In addition, such tools also produce *false positives* [7] which, e.g., represent consciously implemented design constructs with symptoms similar to bad smells.

For these reasons, and due to the general ambivalence of smell detection and refactoring (e.g., regarding the design rationale), software engineers need to investigate the source code and design documentation via code and design reviews [11]. Design-critique and code-quality tools such as *SonarQube* [12] *JArchitect* [13] or *NDepend* [14] can provide decision support for software engineers using different metrics (such as dependencies). Via rules and thresholds, the design quality of the system under analysis can be measured and quantified in terms of the system's technical debt (TD) [15]. The TD score then represents the estimated person-hours to fix all identified debt items. Moreover, due to time pressure and development plans, resources in terms of time and man-power available for refactoring are limited, see, e.g., [16]. Thus, the candidates for refactoring need to be prioritized based on a certain paradigm, such as the level of relevance for the new release version or the risk level etc., see, e.g., [17]. After that, the different options for refactoring the candidates need to be assessed, planned and are finally performed, for example according to rules provided by [1, 2].

This way, the refactoring process can be roughly subdivided into the following two steps performed by software engineers:

(A) *identify and assess refactoring candidates*, and
(B) *plan and perform refactoring steps.*

[1]For example, Fowler et al. [1] list 22 different kinds of bad smells.

Table 1. Levels in Bloom's taxonomy with corresponding exemplary competences in software refactoring.

Level	Competences related to identifying and assessing refactoring candidates (**A**)	Competences related to planning and performing refactoring steps (**B**)
(1) Knowledge	*Reading* and *remembering* the documented knowledge on rules/symptoms for identifying bad smells.	*Reading* and *remembering* the documented knowledge on rules for planning and performing refactoring (steps).
(2) Comprehension	*Understanding* the rules/symptoms for identifying bad smells.	*Understanding* the rules for planning and performing refactoring (steps)
(3) Application	*Identifying* refactoring candidates (e.g., bad smells) according to rules/symptoms (in small synthetic examples).	*Performing* corresponding refactoring steps (in small synthetic examples).
(4) Analysis	*Analyzing* the system's source code and design as well as the candidate's structural and behavioral dependencies while identifying refactoring candidates (in larger code base).	*Analyzing* the system's source code and design as well as the candidate's structural and behavioral dependencies while performing corresponding sequences of refactoring steps (in larger code base).
(5) Evaluation	*Comparing* and *prioritizing* refactoring candidates (according to applied paradigm, such as risk or relevance).	*Comparing* and *selecting* options/paths for performing the refactoring (steps).
(6) Creation	*Developing* and/or *improving* tools for assisting in smell detection and refactoring, or *designing* and/or *revising* (company's) strategies for refactoring or managing technical debt.	

2.2. Gamification in Software Engineering Education

Teaching and learning software development provides multiple difficulties [18] (for instance, with regard to comprehending control or data structures), which are addressed by multiple teaching techniques [19], e.g. by tutoring systems applying program visualization [20]. Moreover, games can foster motivation; the objective is that the learner has fun in doing difficult activities while acquiring relevant competences. Thus, in recent years, multiple gamification approaches in software-engineering education have been proposed (see, e.g., [21, 22]); for an overview, see [23, 24]. In particular, serious games (see, e.g., [25]) aim at simulating or providing real-world conditions, authentic regarding the field of application, while including motivational (and fun) aspects. Here we investigate the possibilities for serious refactoring games, which foster competences regarding the refactoring steps **A** and **B** described above (in Section 2.1).

3. Learning Objectives and Objects for Software Refactoring

3.1. Bloom's Taxonomy for Software Refactoring

For classifying learning objectives and competences the taxonomy provided by Bloom is very popular, also in software engineering education, see, e.g., [8, 26]. Table 1 provides Bloom's six (revised) levels of cognitive competences with corresponding tasks/competences in software refactoring (with regard to the two steps of identifying/assessing candidates and planning/performing refactoring steps).

Different catalogs are established which provide knowledge in terms of rules on how to detect refactoring candidates (e.g., bad smells) and how to perform the corresponding refactorings (see, e.g., [1, 2]). In Bloom's taxonomy, the aforementioned catalogs only address the first two competence levels, i.e. *knowledge* and *comprehension* (see Tab. 1). The *application* competence can be demonstrated by using the knowledge from steps (1) and (2) for the candidate identification and modification of source code in an exemplary software system or fragment of it. For example, editor-based tutoring systems (see Section 2.2) require the learner to identify candidates and perform refactoring in small synthetic examples based on particular instructions. This learning technique addresses the *application* level (third). *Analysis* in the context of learning software refactoring can signify to investigating the source code and design of the given software system for analyzing the candidate's structural and behavioral dependencies in the context of candidate identification and performing the refactoring. The fifth step of *evaluation* is represented by comparing and weighting the candidates for refactoring according to a given or selected prioritization paradigm. Moreover, also comparing the different possible options and paths for refactoring and finally selecting one, can be seen as an evaluation competence. Finally, the *creation* competence can be expressed by developing/improving tools for smell detection or refactoring, or by designing/revising company's strategies for managing the refactoring process.

With focus on the development of learning systems, the question arises, how to foster (and assess) higher levels of competences in software refactoring (levels 3 to 6). By providing or simulating real-world conditions, i.e. an infrastructure comprising a larger software system, higher problem level and complexity of tasks, serious-games represent a promising way to impart these higher-level competences.

3.2. Learning Objects

Orthogonally to the learning objectives, different learning objects in refactoring can be distinguished

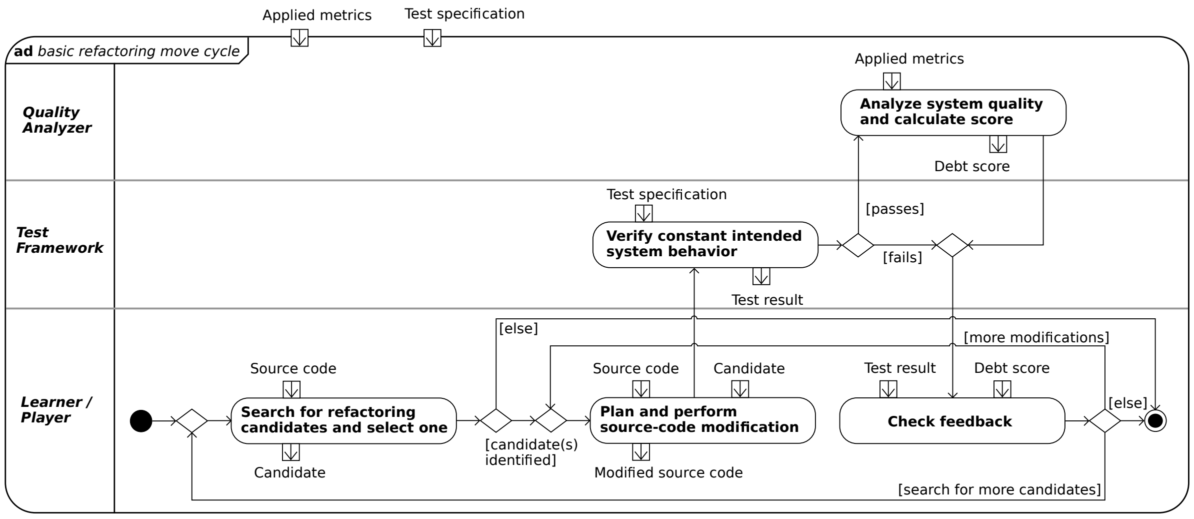


Figure 2. Basic move cycle for serious refactoring games from player perspective (bottom) with behavioral and quality assessment (medium and top layer).

according to the smell categories (or types of debt items; see, e.g., [4]), such as, for example: (stylistic) code smells (see, e.g., [1]), design smells (aka architectural smells; see, e.g., [2]), security code smells, test smells (see, e.g., [27]), documentation smells, requirements smells (see, e.g., [28]), and process smells. The proposed approach is generic in the sense that it can be variably applied to any smell type, provided that quality analyzers with corresponding smell metrics are available.

4. Game Design

For designing serious refactoring games, we can take up tools and techniques established in the software engineering practice. Software refactoring can be defined as a behavior preserving source-code transformation for the purpose of improving the system's internal quality (see, e.g., [1]). For assessing whether the behavior has not changed after refactoring, in practice test frameworks are applied in terms of runtime regression tests. For assessing the quality of the system, so called quality analyzer tools are very popular nowadays (see, e.g., Section 2.1). For the game design presented in this paper, we will combine these pre-existing available kinds of software tools. The following game design is structured into basic game mechanics (see, e.g., [29]) for describing fundamental game rules, e.g., in terms of basic move cycle and goal of the game. Then several game modes are explained which build on these mechanics. Afterwards also the technical foundation of the game design will be explained in terms of a component architecture including the aforementioned assessment tools and providing requirements for technical implementation.

4.1. Basic Game Mechanics

Move: A move consists of the following three steps:
(1) Search for and select refactoring candidates
(2) Plan and perform source-code modification
(3) Check the feedback/result in terms of system's behavior and quality

Steps (1) and (2) correspond to the steps A and B described in Section 2.1. For one move, the steps can be performed iteratively with multiple source-code modifications, e.g., in case of a refactoring step sequence or a failing test. Fig. 2 depicts the move cycle in terms of a UML activity diagram [30].
Score: The quality is expressed in terms of the system's technical debt score measured by a quality analyzer after the tests have passed. A high debt score correlates with low system quality.
Goal: The primary goal is defined as maximizing the system's internal quality (i.e., minimizing the debt score), while preserving the system's external behavior.
Correct Move: A correct move does not change the external behavior of the system, which is assessed by passing runtime regression tests.
Successful Move: A successful move is a correct move with the addition that the system's debt score is reduced compared to the state before refactoring.

4.2. Game Modes

Based on the base game mechanics, several game modes can be derived. Fig. 3 depicts different exemplary game

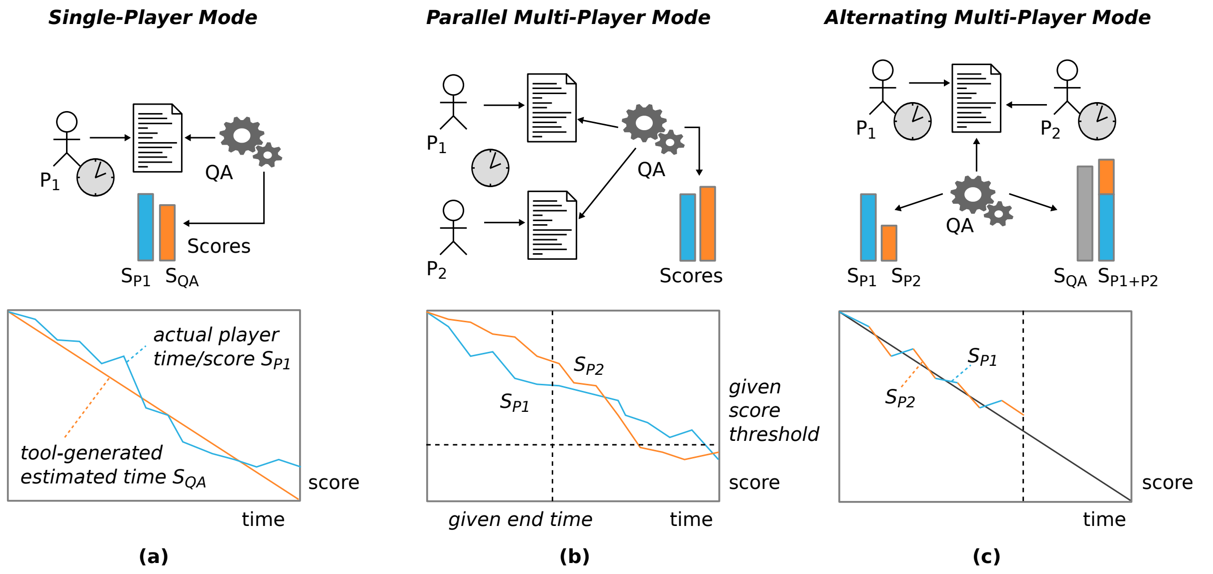


Figure 3. Exemplary refactoring game modes with game constellations (top) and corresponding leader-boards with score-point progression (bottom); for details, see Section 4.2.

modes, which are explained in detail with regard to game mechanics and dynamics. Orthogonally to the game modes, game variants can be distinguished with regard to smell categories. Fig. 1 depicts leader-boards for these game variants. Each node represents a system state, the edges alternative refactoring paths and sequences leading to different following system states. The scores (at right-hand of each leader-board) indicate the technical debt scores of the participating and competing players. The smell types (which are also described in 3.2) can be combined with the following game modes to design game scenarios.

4.2.1. Single-Player Mode The single-player mode represents the most basic game mechanics (see (a) in Fig. 3). One player searches for refactoring candidates in the code base of the system under analysis (SUA) and performs corresponding refactorings. The time for each refactoring move is captured. As already stated above, most popular quality-analyzer tools (such as *SonarQube* [12], *JArchitect* [13], and *NDepend* [14]) quantify the system's technical debt in terms of a score which represents the person-hours needed to fix the debt (items). This value can be applied as benchmark not just for measuring the quality of the system, but for measuring refactoring efficiency. This way, we can define rules for a competition between the quality analyzer (QA) with its tool-generated expected time to fix the debt (item) and the player P_1 with the actual time needed for performing the corresponding refactorings of the debt items. A successful move for P_1 in this game mode consists in being faster in refactoring one debt item (bad smell) than the tool-based estimation. The ultimate goal for P_1 is being in total faster in reducing the debt than estimated by the quality-analyzer tool.

4.2.2. Parallel Multi-Player Mode Another game mode is represented by two (or more) players (e.g., P_1 and P_2) competing against each other while working *in parallel* on different images of the system under analysis (SUA) with an identical state at game start, i.e. identical source code and debt score (see Fig. 3 (b)). As in the single-player mode (in Section 4.2.1), the goal is to efficiently minimize the debt score. In this mode with the difference that two (or more) human players compete against each other. For this mode, we can differ two exemplary variants:

- **A game end-time is set** In this case, the player with the lowest technical-debt score after a previously defined time period (e.g., after 40 minutes) is the game winner. For an example, see P_1 in Fig. 3 (b).
- **A score threshold is set** Here, the player wins who first falls below the defined score threshold. For an example, see P_2 in Fig. 3 (b).

4.2.3. Alternating Multi-Player Mode The third exemplary mode is also represented by two (or more) interacting players (e.g., P_1 and P_2; as in the example above, Section 4.2.2). But in this mode, the players work on the same instance of the source code and alternately (or in turn, for more than two players) perform refactorings, see Fig. 3 (c). For this mode, two exemplary variants are described:

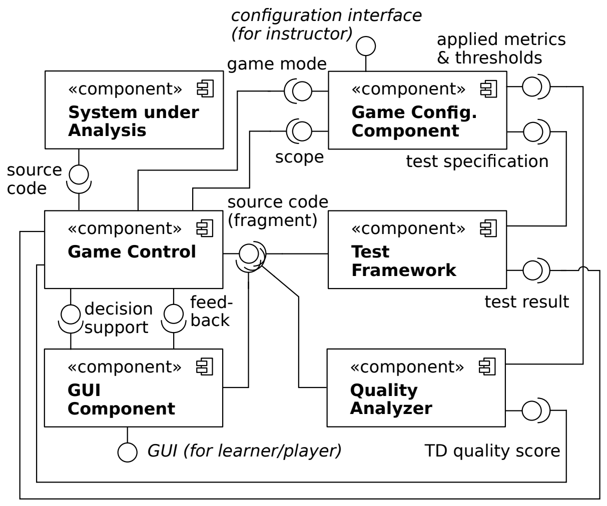
Competing Players In this variant, a player selects a candidate and performs source-code modification in terms of refactoring steps, but without testing the behavioral correctness after each step. Only at the end, as soon as she believes that the refactoring move is finished, the system behavior is tested. For each refactoring move (i.e. here the modification cycle until testing), only a certain amount of time is set. In case the time is up, also the runtime tests are triggered. Anyway, in case all tests pass, the player's score is reduced by the debt estimation of the refactored debt item; see the scores S_{P1} and S_{P2} at the left-hand and leader-board below in Fig. 3 (c). If a test fails, the player's score, for example, increases by the debt estimation of the vainly refactored item.

Collaborating Players In the second variant of this mode, the players work as a team that strives to minimize the debt score of the system under analysis. For this reason, they alternately perform refactoring moves. In this case, the player switch could take place after each modification step or after each completed refactoring (i.e. move cycle). A suitable opponent could be again the quality analyzer with its tool-generated debt estimation (see the score bars at right-hand in Fig. 3 (c). Alternatively, teams could compete against each other.

With these different variants, the game play of this third mode is comparable to a refactoring *Jenga* game. There, in turn, a wooden block is pulled by a player out of the tower and placed on top of the tower, with which the tower is constantly raised (comparable to raising the system quality). Loser of the game is the one who brings down the tower (comparable to failing tests).

5. Component Architecture

Now we describe a software architecture that can supply the game-design functionality defined in Section 4. Fig. 4 provides an overview of the architecture in terms of a UML component diagram [30] representing components and interfaces. The component architecture is organized by the components' functional responsibilities (also in order to reduce the coupling between components) and is independent from a particular programming language. In the following, the components' responsibilities as well as the interfaces to other components are explained in detail. For each component, we also state technical (and educational) requirements for implementation and provide exemplary tools and techniques where appropriate.



configuration interface (for instructor)
applied metrics & thresholds
game mode
«component» System under Analysis
«component» Game Config. Component
scope
test specification
source code
source code (fragment)
«component» Game Control
«component» Test Framework
test result
decision support
feed-back
«component» GUI Component
«component» Quality Analyzer
GUI (for learner/player)
TD quality score

Figure 4. Architectural overview with components and interfaces provided and used by the components.

Configuration The configuration component allows the instructor to configure the refactoring games. In particular, the (functional) requirements for the system under analysis (SUA) in terms of a test specification for the test framework are configured as well as the metrics and thresholds to be applied by the quality analyzer. Moreover, the instructor selects the game mode (e.g., single-player, multi-player in parallel or alternating way) as well as the game-termination mode (e.g., a certain end time or a score threshold) and defines the scope of the SUA (i.e. the view on the SUA's source code relevant for the current game scenario). Besides the actual game configuration as handled by the instructor, other features regarding the architectural composition need to be configured by a corresponding software expert beforehand (see Tab. 2). In particular, the programming language, the system under analysis (SUA), the test framework and the quality analyzer need to be selected.

Table 2. Pre-game and game configuration.

Pre-Game Configuration	Game Configuration
Programming language	–
System under analysis (SUA)	Scope of the SUA (fragment)
Test framework	Specification of test script (runtime tests)
Software quality analyzer	Smell category (learning objects) & corresponding metrics
–	Refactoring game mode

System under Analysis (SUA) The SUA source code serves as basis for the refactoring moves in the game play. According to the configuration by the instructor, only images of the SUA or selected fragments are actually modified by the players. From an educational perspective, of course, the SUA should be based on a programming language that supports the learning setting. In addition, the SUA (or selected fragment) must provide a manageable complexity and must contain smell candidates according to learning setting. Regarding the technical implementation, it is an important requirement that the applied assessment components (i.e. test framework and quality analyzer, see below) are also available for the SUA programming language.

Game Control The game-control component coordinates the game interactions between the players and the test framework and quality analyzer. Thereby, its task is to forward the modifications in the SUA source code performed by the player(s) from the GUI component to the assessment tools (i.e. test framework and quality analyzer). And the other way round, to forward the feedback from the assessment tools (i.e. test result and debt score; or optionally other benchmarks) via the GUI component to the player(s).

Test Framework The test framework guarantees that no behavioral change (according to the specified functional requirements) was introduced during source-code modification. Via automated runtime regression tests (see, e.g., [31]) the SUA source code (fragment) is assessed after each refactoring move. From an implementation viewpoint, many test frameworks can be applied for verifying the runtime behavior, e.g., unit, integration, or scenario test frameworks, with the requirement that they are compatible with the SUA programming language.

Quality Analyzer The quality analyzer component investigates the quality of the source code (fragment) of the system under analysis. It requires the source code (fragment) as well as the metrics that should be applied for analysis. In turn, it provides the debt score. Popular quality-analyzer tools such as *SonarQube* [12], *JArchitect* [13], and *NDepend* [14] measure technical debt (TD) of a software system by quantifying the effort that would be necessary to fix the identified debt issues in terms of person-hours. The resulting debt score represents the sum of all debt estimations.

GUI Component The GUI component presents a graphical user interface (GUI) to the participating players allowing to modify the source code fragment via an integrated editor. Moreover, it reports the feedback on the current state of the SUA received from the assessment tools (i.e. test result from test framework, debt score from quality analyzer) via the game control, and optionally further decision support. For game navigation (e.g. for completing a move), the players must be provided with corresponding control elements.

6. Game Scenario

The following scenario illustrates the feasibility of serious games for a concrete learning setting.

Learning Setting Consider a learning group of students in the 3rd year SE major with good knowledge of *Java*. The students have completed a course in software architecture and are aware of software-design smells (aka architectural smells) by having read the book [2]. The instructor now aims at expanding the students' competences (towards *application, analysis* and *evaluation*) regarding the refactoring of architectural smells, in particular MODULARIZATION smells (i.e. smells violating the design/architecture principle of modularization, e.g., FEATUREENVY, DATACLASS or CYCLICDEPENDENCY; for details, please see, e.g., [2]).

Pre-Game Configuration For this purpose, the following pre-game configurations are made: *Java* as programming language, *JUnit* as test framework, *SonarQube* as quality analyzer, and as system under analysis (SUA) she applies *ArgoUML*[2].

Game Configuration After that, she configures the game by choosing the metrics type. The technical debt score of MODULARIZATION smells can be measured in terms of the metric of dependencies, especially inter-class dependencies in terms of *efferent* (i.e. out-going) and *afferent* (i.e. in-going) dependencies of methods and fields. As system scope she selects the entire system (alternatives would be, e.g., a few specific packages). Fortunately, for the selected SUA already *JUnit* tests are provided that are appropriate for verifying the intended system behavior. As game play mode she selects the *Parallel Multi-Player Mode* (see Section 4.2.2 and (b) in Fig. 3) to foster students' motivation and commitment through competition.

Game Play Two learners (P_1 and P_2) are participating, each works with an own image of the SUA code base with an identical state at the game start (regarding source code and debt score, which is at 5 hours and 30 minutes). While reviewing the source code, P_1 identifies at first a candidate method for a FEATUREENVY smell that calls two attributes of another class that are not used by the host class itself. Based on the rules provided in [2], he then performs for each attribute a MOVEFIELD (see [1] refactoring to class of the envy method in order to reduce inter-class dependencies. Afterwards he triggers the test run of which 3 unit tests fail. He then investigates the failing tests and recognizes that also two other methods have called the moved attributes. So he redirects the target class of these methods and runs the tests again,

[2]Available at http://argouml.tigris.org/.

which now pass altogether. In contrary, in the same time during code review P_2 identifies a candidate for a CYCLICDEPENDENCY smell via the symptoms of two classes which provide multiple mutual inter-class dependencies in terms of references and method-call dependencies from one to the other class and vice versa. For instance, the book [2] suggests four options for refactoring such cyclic dependencies:

1. introduce an interface
2. remove dependency by a MOVEMETHOD or MOVEFIELD refactoring
3. move the code that is responsible for cyclic dependency to another class
4. merge the involved classes into one class

P_2 selects option 4 and merges the two classes into one class. He transfers all fields and methods (in terms of multiple MOVEFIELD and MOVEMETHOD refactorings) into the other class and redirects where necessary the targets of the moved methods. After that, he runs the tests of which 7 fail. He analyses the failing unit tests and finds out that some other methods have used the disintegrated class. So he redirects all these methods to the merged class and triggers the tests again, which now pass.

Preliminary Score (after first round) These performed refactorings have an effect on the technical debt score. Removing the inter-class dependencies of this particular FEATUREENVY reduces the score by 8 minutes. The removal of the CYCLICDEPENDENCY in contrast lowers the score by 20 minutes. So after the first round, player P_1 has a score of 5 hours and 22 minutes, player P_2 one with 5 hours and 10 minutes.

This simple scenario illustrates one round of a possible game variant (e.g., by selecting inter-class dependency metrics and *Parallel Multi-Player Mode*). For other exemplary variants, see Figs. 1 and 3.

7. Related Work

To the best of our knowledge, there are only very few approaches addressing serious games, gamification or game-based learning in the field of software refactoring so far. For instance, [32] proposes a serious refactoring game in the sense that it is based on the code base of a real-world system. The presented tool includes some gamification elements (i.e. activity feeds, points and leader-boards as well as progress bars). In particular, it traces and rewards users for applying *Eclipse* build-in refactoring commands. For each of these refactorings, the user receives points – regardless of whether the refactoring was actually justified or not (i.e. also in case no smell exists). In contrast, our approach rewards *meaningful* refactorings. For this reason, our framework reflects the technical debt score (measured according to predefined metrics and thresholds) for scoring the value of refactoring. Moreover, our framework assesses that no error has been introduced according to an instructor-based or pre-existing (adapted) test specification.

Further research related to our approach can be roughly divided into the following two groups: (1) *teaching software refactoring* and (2) *serious games (and gamification) in software engineering education.*

Teaching Software Refactoring Closely related are learning environments and tutoring systems for refactoring (see, e.g., [33, 34]). In particular, Sandalski et al. propose an intelligent analyzer assistant very similar to a refactoring recommendation system (see, e.g., [9]) which identifies and highlights simple code-smell candidates [33]. After the student's code modification, it reacts by proposing (better) refactoring options. In addition, López et al. report on a study for teaching refactoring [34]. They propose exemplary refactoring task categories and classify them according to Bloom's taxonomy and according to learning types. They also describe learning settings which aim at simulating real-world conditions by providing, e.g., an IDE and revision control systems, but are based on small synthetic code examples (e.g., 200 LOC).

In a broader context, also teaching of refactoring without integrated automated feedback is related to our approach. For instance, Smith et al. propose an incremental approach for teaching different refactoring types on college level in terms of learning lessons [35, 36]. The tasks are designed in an instructive way with exemplary solutions that have to be transferred to the current synthetic smell. Within this, they provide an exemplary learning path for refactorings. 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) [37]. Then they compared the quality of the resulting code.

Serious Games in SE Education Some related approaches for serious games and gamification exist addressing other activities in the software engineering (education) domain (see, e.g., [21, 22, 38]). For an overview, see [23] and [24]. Besides general programming aspects, only a few of these approaches address topics more related to refactoring, such as games for code reviews [39] or games for exploring code smells via visualized dependency graphs [40]. In a broader sense, several gamification approaches exist, for instance, in intelligent tutoring systems (ITS) [41] or in on-line learning environments [42].

8. Discussion

In this paper, we reflected on the possibilities of combining pre-existing and popular technical components for quality analysis and runtime testing for creating serious refactoring games. The proposed game design and component architecture allow for deriving multiple variants of game scenarios for different learning settings. By recycling pre-existing technical components, the implementation costs are low. An empirical evaluation of the effects of the outlined game design requires a meaningful use in a learning environment and will be approached in the next step.

We believe that these refactoring games are not limited to learning purposes, but could also be applied for improving the quality of industry software. For instance, the code base of a system used in practice (or parts of it that are not sensitive in terms of security or business secrecy) can be analyzed and improved by a multitude of players. Each successful move could be then financially rewarded, similar to micro-task crowd-sourcing [43]. The measured times for reducing code smells could also be used to improve the statistical estimates (debt values) provided by quality analyzers.

9. Conclusion

In this paper, we investigated serious games in software refactoring. In particular, we presented a game design with component architecture and technical requirements for game implementation. Within this, exemplary variants of serious games for refactoring based on this infrastructure (e.g., variable in terms of quality metrics and game modes) have been shown. A short exemplary game scenario demonstrates the feasibility in a concrete learning setting.

The paper addresses the challenge of how to foster higher levels of competences (according to Bloom's taxonomy) in software refactoring by combining pre-existing components for creating serious games. As shown, the focus of the games is on analysis and application tasks (e.g., for analyzing the code base and performing the refactorings). The games also demand evaluation competences for comparing refactoring candidates and options. Further potential for promoting evaluation competences lies in a combination of metric types, which then requires to select a paradigm/strategy for prioritizing candidates. The games also provide instant feedback and (self-)assessment (e.g., via test result and debt score). The proposed game design and architecture are generic in the sense that they are independent of a particular programming language, test framework and quality metrics. So, they can be applied and extended for different learning settings.

For future work, we aim at conducting experiments with our students in Information Systems for investigating to what extent playing the games has an effect on the students' motivation and competence acquisition.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[2] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: Managing technical debt*. Morgan Kaufmann, 2014.

[3] J. Long, "Software reuse antipatterns–revisited.," *Software Quality Professional*, vol. 19, no. 4, 2017.

[4] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, pp. 100–121, 2016.

[5] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, p. 18, ACM, 2016.

[6] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment.," *J. Object Technology*, vol. 11, no. 2, pp. 5–1, 2012.

[7] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Antipattern and code smell false positives: Preliminary conceptualization and classification," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1, pp. 609–613, IEEE, 2016.

[8] R. Britto and M. Usman, "Bloom's taxonomy in software engineering education: A systematic mapping study," in *Frontiers in Education Conference (FIE), 2015 IEEE*, pp. 1–8, IEEE, 2015.

[9] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of type-checking bad smells," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pp. 329–331, IEEE, 2008.

[10] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[11] C. F. Kemerer and M. C. Paulk, "The impact of design and code reviews on software quality: An empirical study based on PSP data," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 534–550, 2009.

[12] G. Campbell and P. P. Papapetrou, *SonarQube in action*. Manning Publications Co., 2013.

[13] CoderGears, "JArchitect," 2018. [Sept. 22, 2018].

[14] ZEN PROGRAM, "NDepend," 2018. [Sept. 22, 2018].

[15] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE software*, vol. 29, no. 6, pp. 18–21, 2012.

[16] E. Tempero, T. Gorschek, and L. Angelis, "Barriers to refactoring," *Communications of the ACM*, vol. 60, no. 10, pp. 54–61, 2017.

[17] L. F. Ribeiro, M. A. de Freitas Farias, M. G. Mendonça, and R. O. Spínola, "Decision criteria for the payment of technical debt in software projects: A systematic mapping study.," in *ICEIS (1)*, pp. 572–579, 2016.

[18] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003.

[19] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson, "A survey of literature on the teaching of introductory programming," in *ACM SIGCSE Bulletin*, vol. 39, pp. 204–223, ACM, 2007.

[20] J. Sorva, V. Karavirta, and L. Malmi, "A review of generic program visualization systems for introductory programming education," *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 4, p. 15, 2013.

[21] K. Berkling and C. Thomas, "Gamification of a software engineering course and a detailed analysis of the factors that lead to it's failure," in *Interactive Collaborative Learning (ICL), 2013 International Conference on*, pp. 525–530, IEEE, 2013.

[22] T. M. Connolly, M. Stansfield, and T. Hainey, "An application of games-based learning within software engineering," *British Journal of Educational Technology*, vol. 38, no. 3, pp. 416–428, 2007.

[23] O. Pedreira, F. García, N. Brisaboa, and M. Piattini, "Gamification in software engineering–a systematic mapping," *Information and Software Technology*, vol. 57, pp. 157–168, 2015.

[24] M. M. Alhammad and A. M. Moreno, "Gamification in software engineering education: A systematic mapping," *Journal of Systems and Software*, vol. 141, pp. 131–150, 2018.

[25] T. M. Connolly, E. A. Boyle, E. MacArthur, T. Hainey, and J. M. Boyle, "A systematic literature review of empirical evidence on computer games and serious games," *Computers & Education*, vol. 59, no. 2, pp. 661–686, 2012.

[26] S. Masapanta-Carrión and J. Á. Velázquez-Iturbide, "A systematic review of the use of bloom's taxonomy in computer science education," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pp. 441–446, ACM, 2018.

[27] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp. 56–65, IEEE, 2012.

[28] H. Femmer, D. M. Fernández, S. Wagner, and S. Eder, "Rapid quality assurance with requirements smells," *Journal of Systems and Software*, vol. 123, pp. 190–213, 2017.

[29] R. Hunicke, M. LeBlanc, and R. Zubek, "MDA: A formal approach to game design and game research," in *Proceedings of the AAAI Workshop on Challenges in Game AI*, vol. 4, pp. 1–5, AAAI Press San Jose, CA, 2004.

[30] Object Management Group, "Unified Modeling Language (UML), Superstructure, Version 2.5.0," June 2015. [Sept. 22, 2018].

[31] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *Journal on Educational Resources in Computing (JERIC)*, vol. 5, no. 3, p. 4, 2005.

[32] L. Elezi, S. Sali, S. Demeyer, A. Murgia, and J. Pérez, "A game of refactoring: Studying the impact of gamification in software refactoring," in *Proceedings of the Scientific Workshop Proceedings of XP2016*, p. 23, ACM, 2016.

[33] M. Sandalski, A. Stoyanova-Doycheva, I. Popchev, and S. Stoyanov, "Development of a refactoring learning environment," *Cybernetics and Information Technologies (CIT)*, vol. 11, no. 2, 2011.

[34] C. López, J. M. Alonso, R. Marticorena, and J. M. Maudes, "Design of e-activities for the learning of code refactoring tasks," in *Computers in Education (SIIE), 2014 International Symposium on*, pp. 35–40, IEEE, 2014.

[35] S. Smith, S. Stoecklin, and C. Serino, "An innovative approach to teaching refactoring," in *ACM SIGCSE Bulletin*, vol. 38, pp. 349–353, ACM, 2006.

[36] S. Stoecklin, S. Smith, and C. Serino, "Teaching students to build well formed object-oriented methods through refactoring," *ACM SIGCSE Bulletin*, vol. 39, no. 1, pp. 145–149, 2007.

[37] S. Abid, H. Abdul Basit, and N. Arshad, "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.

[38] M. R. de Almeida Souza, K. F. Constantino, L. F. Veado, and E. M. L. Figueiredo, "Gamification in software engineering education: An empirical study," in *Software Engineering Education and Training (CSEE&T), 2017 IEEE 30th Conference on*, pp. 276–284, IEEE, 2017.

[39] S. Khandelwal, S. K. Sripada, and Y. R. Reddy, "Impact of gamification on code review process: An experimental study," in *Proceedings of the 10th Innovations in Software Engineering Conference*, pp. 122–126, ACM, 2017.

[40] F. Raab, "Codesmellexplorer: Tangible exploration of code smells and refactorings," in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, pp. 261–262, IEEE, 2012.

[41] C. González, A. Mora, and P. Toledo, "Gamification in intelligent tutoring systems," in *Proceedings of the Second International Conference on Technological Ecosystems for Enhancing Multiculturality*, pp. 221–225, ACM, 2014.

[42] S. J. de Santana, H. A. Souza, V. A. Florentin, R. Paiva, I. I. Bittencourt, and S. Isotani, "A quantitative analysis of the most relevant gamification elements in an online learning environment," in *Proceedings of the 25th international conference companion on world wide web*, pp. 911–916, International World Wide Web Conferences Steering Committee, 2016.

[43] U. Gadiraju, R. Kawase, and S. Dietze, "A taxonomy of microtasks on the web," in *Proceedings of the 25th ACM conference on Hypertext and social media*, pp. 218–223, ACM, 2014.