A Framework for the Assessment and Training of Software Refactoring Competences

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

Keywords: Software Refactoring, Knowledge and Competence Management, Software Engineering Education and Training

Abstract: Long-living software systems are becoming increasingly complex and difficult to maintain. Software refactoring is considered important to achieve maintainability and extensibility of a software system over time. In practice, it is still often neglected, partly because of costs, the perceived risks of collateral damage and difficulties of individuals working on certain components of complex software. It is therefore important for software projects that software developers have the appropriate skills and competences to efficiently perform software refactoring. However, so far there is no systematization of competences in software refactoring to guide in the assessment or training of competences, e.g., for planning or evaluating training activities and paths. In this paper, we address this need by presenting a competence framework for software refactoring by applying Bloom's revised taxonomy for educational objectives. In particular, we specify competence levels by combining knowledge and cognitive-process dimensions. Via a case study with two existing training environments (i.e. a tutoring system and a serious game), we demonstrate by example that the framework can support (1) in analyzing the competence levels addressed by the training environments and (2) in reflecting training paths for software refactoring. Finally, we discuss the limitations and the further potential of the framework.

1 INTRODUCTION

According to studies, software maintenance accounts often to 80 % or more than 90 % of software project costs (Schach, 2007; Erlikh, 2000). When software systems are used in changing environments over many years, such systems face an increasing complexity, making it even more difficult and time consuming to maintain them (cf. *technical debt* (Kruchten et al., 2012) and *software aging* (Parnas, 1994)), which can result in a delay in delivering new software functionalities. Software refactoring aims at improving the maintainability and extensibility of software systems by restructuring the system's source code while preserving the observable system behavior (Opdyke, 1992; Fowler et al., 1999).

Although considered useful and important, refactoring is often omitted in practice, which is caused by several *barriers* that prevent software developers from refactoring (Tempero et al., 2017). In addition to missing resources (such as limited time) and other management issues, there are barriers that indicate a lack of competences such as the perceived risks and difficulties of performing refactoring. For example, more complex candidates for refactoring such as bad smells on the level of software design or architecture (Suryanarayana et al., 2014) are not directly identifiable by reviewing the source code alone but require to observe static and run-time code dependencies.

These challenges are addressed by several tools for automation and decision-support in detecting smells or in performing corresponding refactorings; see, e.g., (Fernandes et al., 2016). Due to the objective of improving the quality of the software code/design maintained by humans, these activities can hardly be fully automated, but require human expertise, e.g. in terms of software architects that are aware of the system's design rationale and competent in software refactoring. The success of strategies for repaying technical debt via refactoring depends essentially on the organizational capabilities in terms of the knowledge and competences contributed by the individual software developers and project managers.

However, besides textbooks (Fowler et al., 1999; Suryanarayana et al., 2014) that are mediating a basic understanding of refactoring techniques, there are only a few training environments to support software developers in acquiring and improving practical com-

Haendler, T. and Neumann, G.
A Framework for the Assessment and Training of Software Refactoring Competences.
DOI: 10.5220/0008350803070316
In *Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2019)*, pages 307-316
ISBN: 978-989-758-382-7

petences in software refactoring (see Section 2.1). Moreover, so far, there is no systematization of competences in software refactoring to provide guidance in assessment or training activities, which is important in the context of software projects or for education and training, e.g., for measuring the learning progress of students or software developers, planning training activities and paths, or applying, designing and evaluating training environments.

In this paper, we propose a framework that aims at supporting the assessment and training of competences in the field of software refactoring. For this purpose, the framework manifests the relevant concepts and specifies the competence levels. In particular, the framework includes a concept map in terms of a domain ontology structuring the relevant concepts (and relations between them) for competences in the field of software refactoring. Moreover, in order to specify competence levels in terms of knowledge and cognitive-process dimensions, we apply Bloom's revised taxonomy for educational objectives to software refactoring. A case study with two existing training environments demonstrates that the framework can support the analysis of prerequisite and target competences and in planning/evaluating training paths for software refactoring. In particular, this paper provides the following contributions.

- We propose a *competence framework* for software refactoring by specifying the knowledge and cognitive-process dimensions for competences in software refactoring according to Bloom's revised taxonomy for educational objectives.
- We investigate the applicability of the framework via a *case study* with two exemplary training environments for software refactoring (a tutoring system and a serious game).

The remainder of this paper is structured as follows. Section 2 describes the background and related work on software refactoring and competence frameworks. Section 3 deals with examples and concepts of training and assessment activities. In Section 4, we specify knowledge and cognitive-process dimensions for software refactoring by applying Bloom's taxonomy. Then, in Section 5, we use the framework to evaluate two exemplary training environments. In Section 6, we discuss limitations and further potential. Section 7 concludes the paper.

2 BACKGROUND AND RELATED WORK

This section gives a short overview of relevant background (and related work) on software refactoring (Section 2.1) and on frameworks for managing competences (Section 2.2).

2.1 Software Refactoring and Its Challenges

Software refactoring represents a complex activity for improving a software system's maintainability or extensibility (Opdyke, 1992) and for repaying *technical debt* (Kruchten et al., 2012). Candidates for refactoring are issues (such as bad smells) in the source code, software design or architecture, or other artifacts (Fowler et al., 1999). These smells manifest via certain symptoms that can be identified and measured by applying certain metrics (e.g., dependencies between code elements). Thus, the refactoring process can be roughly divided into the following two activities; cf. (Haendler and Frysak, 2018) of *(A) identifying and assessing refactoring opportunities* and *(B) planning and performing refactoring steps*. In last years, several tools have been proposed to support developers, such as *smell detectors* (Tsantalis et al., 2008; Moha et al., 2010) or *software-quality analyzers* and *design-critique tools* (Campbell and Papapetrou, 2013; CoderGears, 2018). Despite these advances, studies indicate that tool-support is rarely used by software developers (Murphy-Hill et al., 2012). However, besides activities that can be potentially automated (e.g., candidate identification or executing refactoring steps), others require human judgment and expertise. For example, the tool-generated candidates need to be assessed to discard *false-positives* (Fontana et al., 2016), or the refactoring activities need to be prioritized regarding the project schedule (Ribeiro et al., 2016). However, so far, only modest attention has been paid to software refactoring by research in software engineering education and teaching. Besides popular textbooks such as (Fowler et al., 1999; Suryanarayana et al., 2014) and approaches for programming lessons (Smith et al., 2006; López et al., 2014), only a few interactive and computer-based approaches exist that provide intelligent and immediate feedback, such as tutoring systems (Sandalski et al., 2011; Rolim et al., 2017; Haendler et al., 2019) or (educational) games (Elezi et al., 2016; Haendler and Neumann, 2019b). So far, there exists no systematization of competences in software refactoring (see below) to support in training and assessment.

2.2 Competence Frameworks

For managing competences or skills in general, several generic frameworks are established. For exam-

ple, Dreyfus and Dreyfus structure skills in terms of five hierarchical learner levels (Dreyfus and Dreyfus, 1980), from *novice*, via *advanced beginner*, *competent*, and *proficient*, to *expert*. Bloom et al. proposed a taxonomy for educational objectives also structured into six hierarchical levels (Bloom et al., 1956; Krathwohl, 2002); also see Section 4.1. In addition, Paquette introduced a competence framework (Paquette, 2007) in which he distinguishes between *actual*, *prerequisite* and *target* competences. Moreover there are other generic frameworks for knowledge management such as (Alavi and Leidner, 2001) that provide different perspectives on and categorizations of knowledge. For specifying competences in software engineering and information systems, the revised version of Bloom's taxonomy (Krathwohl, 2002) is quite popular; see, e.g., (Britto and Usman, 2015; Bork, 2019). There are a few approaches for applying Bloom's framework to software refactoring (López et al., 2014; Haendler and Neumann, 2019a; Haendler and Neumann, 2019b), which are not comprehensively systematized. For example, they do not reflect the knowledge dimension for creating a two-dimensional matrix. Built on these approaches, we here propose a competence framework for training and assessment of software-refactoring competences.

3 TRAINING AND ASSESSMENT OF COMPETENCES IN SOFTWARE REFACTORING

Here we describe two short scenarios for the exemplary training and assessment of refactoring competences (Section 3.1) and present an overview of relevant basic concepts (Section 3.2).

3.1 Exemplary Scenarios for Training and Assessment

University Course. Consider a university course on software design and architecture located in a Master's study in *Information Systems* or *Software Engineering*. This course includes techniques for software refactoring of software-design issues. The course instructor introduces the refactoring techniques based on the textbook (Suryanarayana et al., 2014). For mediating practical skills for applying the techniques, she seeks for appropriate training environments (see, e.g., Section 5.1). In particular, for creating the training path, it is required that the complexity levels of the activities should build on each other and finally lead the students to the aimed learning objectives. For this purpose, the actual competences possessed by the students as well as the prerequisite and target competences of the training activities need to be identified.

Software Project. Also imagine a software project confronted with technical debt (see Section 2.1). Since the debt impedes the delivery of new features, it is planned to repay it by software refactoring. Among other challenges such as managing the allocation of resources such as time and people for the refactoring activities, the project manager has to investigate beforehand the organizational capabilities for performing the activities. That also includes considering the appropriateness of the skills provided by the software developers to efficiently apply the corresponding refactoring techniques as well as the selection and use of tool support. For this purpose, at first, the competences required for repaying the debt have to be assessed and compared with the actual competences possessed by the development team. From this delta the need for implementing training activities can be derived. Based on this, the prerequisite and target competences have to be identified to find a training environment that can optimally support in acquiring the aimed competences.

3.2 Conceptual Overview

In the following, we abstract from these scenarios by describing the underlying concepts related to the triangle of *training/assessment*, *competences* and *software refactoring* as described in the scenarios.

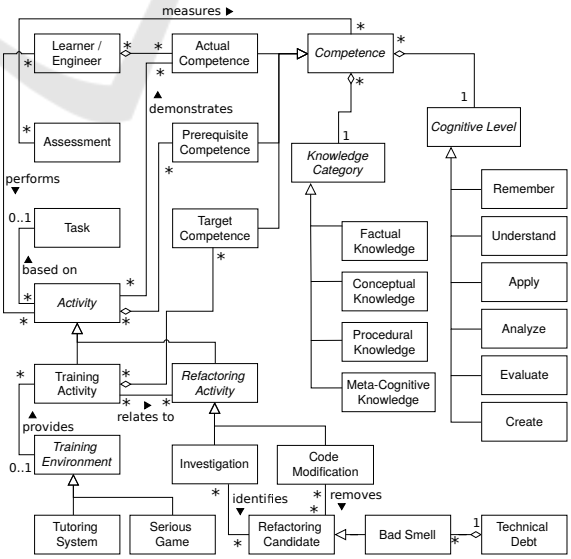

Figure 1: Concept map (domain ontology) with concepts relevant for assessing and training competences in software refactoring; the map extends an excerpt from an existing ontology (Haendler and Neumann, 2019a).

The concept model depicted in Fig. 1 gives a short overview of these basic concepts in terms of a UML class diagram (Object Management Group, 2017). It builds on the ontology developed in (Haendler and Neumann, 2019a) by extending a relevant excerpt. Basically, every `Competence` (see Fig. 1) can be classified by the dimensions of `Knowledge Category` (i.e. four categories) and `Cognitive Level` (i.e. six hierarchical levels) (Krathwohl, 2002); for details, see Section 4. A `Competence` can express in terms of an `Actual Competence` possessed, acquired or improved by a `Software Engineer`. Moreover, a `Prerequisite Competence` is required for performing a certain `Activity` (optionally based on a given `Task`). A `Target Competence` can describe a *learning objective*; also see (Paquette, 2007). In turn, an `Activity` can demonstrate the mastery of an `Actual Competence` and can manifest as `Training Activity` optionally supported by a `Training Environment` (e.g., a `Tutoring System` or a `Serious Game`; for details, see Section 5.1) or as `Refactoring Activity`, e.g., as `Investigation` (e.g., for identifying or assessing `Refactoring Candidates`, such as `Bad Smells`) or as `Code Modification` (refactoring step) for removing the candidates; also see (Kitchenham et al., 1999; Avgeriou et al., 2016). The sum of `Bad Smells` (*debt items*) constitutes a software system's `Technical Debt` (Kruchten et al., 2012).

4 REFACTORING COMPETENCES

In this section, we specify competence levels for software refactoring by applying Bloom's revised taxonomy for educational objectives (Krathwohl, 2002).

4.1 Bloom's Taxonomy

Bloom's taxonomy for cognitive educational objectives (Bloom et al., 1956) represents a very popular framework for classifying and specifying objectives for learning and training in terms of task statements (also see Section 2.2). In the original taxonomy, these competence definitions are structured into six categories in terms of a hierarchy with sequencing levels (from simple/concrete to complex/abstract). In order to overcome limitations regarding the differentiability of knowledge applied for each level, a revised and extended version of the taxonomy has been proposed (Krathwohl, 2002) that includes a second dimension for specifying the types of knowledge, orthogonal to the cognitive levels. By correlating the cognitive-process dimension to the knowledge dimension in a a two-dimensional matrix, the competences can be specified in a more precise way via. This revised version is especially popular in the fields of information systems and software engineering (cf. (Britto and Usman, 2015)) and will be used for our framework. In the following, the structure of the four categories of the knowledge dimension and the six levels of the cognitive-process dimension of the revised taxonomy (Krathwohl, 2002) is shortly described.

Four Knowledge Categories.

- *Factual Knowledge* represents the basic elements a student must know to be acquainted with a discipline or solve problems in it.
- *Conceptual Knowledge* represents the interrelationships among the basic elements within a larger structure that enable them to function together.
- *Procedural Knowledge* represents the how-to-do something in terms of methods of inquiry and criteria for using skills, algorithms, techniques, and methods.
- *Meta-Cognitive Knowledge* represents the knowledge of cognition in general as well as awareness and knowledge of one's own cognition.

Six Levels of Cognitive Processes.

- *Remember* represents the ability of retrieving relevant knowledge from long-term memory.
- *Understand* represents the ability of determining the meaning of instructional messages, including oral, written, and graphic communication.
- *Apply* represents the ability of carrying out or using a procedure in a given situation.
- *Analyze* represents the ability of breaking material into its constituent parts and detecting how the parts relate to one another and to an overall structure or purpose.
- *Evaluate* represents the ability of making judgments based on criteria and standards.
- *Create* represents the ability of putting together to form a novel, coherent whole or make an original product.

4.2 Knowledge and Cognitive Processes in Software Refactoring

In the following, we investigate how the knowledge and cognitive-process dimensions of Bloom's revised taxonomy (see Section 4.1) can be applied to software refactoring. For this purpose, we take into account key activities of the refactoring process. Oriented

Table 1: Categories of the knowledge dimension of Bloom's revised taxonomy (Krathwohl, 2002) applied to key activities in software refactoring (A and B).

Knowledge Category	Knowledge related to identifying and assessing refactoring candidates (A)	Knowledge related to planning and performing refactoring steps (B)
(I) Factual	*Basic elements* and *terminology*, such as: • bad smells, smell symptoms, code and design metrics, the metaphor of technical debt (TD) etc.	*Basic elements* and *terminology*, such as: • code modification, preventive maintenance, regression testing, *red–green–refactor*, repaying technical debt etc.
(II) Conceptual	*Conceptual structure of* and *relationships between the basic elements and terms of (I)*, such as: • the *combination of symptoms* to identify candidates (smells; on a *conceptual* level). • the *classification of refactoring candidates* (bad smells) in terms of types and categories (e.g., software design smells (Suryanarayana et al., 2014) such as according to design principles of ABSTRACTION or MODULARIZATION).	*Conceptual structure of* and *relationships between the basic elements and terms of (I)*, such as: • the *options* and *combinations* of code-modification techniques to realize a refactoring technique (*conceptually*). • the *classification* of refactoring techniques in terms of types and categories (Fowler et al., 1999), such as *moving code elements* (e.g., MOVEMETHOD, EXTRACTCLASS) or *simplifying interfaces* (e.g., ADDPARAMETER, RENAMEMETHOD).
(III) Procedural	*Procedural aspects*, such as: • the *techniques, algorithms and metrics* of applying factual and conceptual aspects (I and II) practically for identifying refactoring candidates (bad smells). • the handling of tool support (e.g., for automated smell detection (Tsantalis et al., 2008), design critique (CoderGears, 2018) and/or technical-debt analysis (Campbell and Papapetrou, 2013)).	*Procedural aspects*, such as: • the *techniques and algorithms* of applying factual and conceptual aspects (I and II) practically for performing options and sequences of refactoring steps (i.e. code modifications). • the handling of tool support (e.g., for automatically performing refactoring steps, for automated regression testing).
(IV) Metacognitive	*Strategies, contextual aspects and self-knowledge*, such as: • the *strategies* for identifying and assessing refactoring candidates; e.g., for discarding *false positives* (Fontana et al., 2016). • *contextual aspects* of identifying and assessing refactoring candidates such as the availability of resources (e.g., time, costs), the project schedule and other activities. • the awareness of one's own capabilities and limitations in identifying refactoring candidates (*self-knowledge*), e.g., regarding the smells covered. • the awareness of the capabilities and limitations of software tools (e.g., smell coverage, availability for progr. languages, *false positives*; see above).	*Strategies, contextual aspects and self-knowledge*, such as: • the *strategies* for performing refactoring, e.g., based on paradigms and criteria for deciding how and when to refactor, see e.g. (Ribeiro et al., 2016). • *contextual aspects* of performing refactorings such as the availability of resources (e.g., time, costs), the project schedule and activities. • the awareness of one's own capabilities and limitations in performing refactorings (*self-knowledge*), e.g., regarding the availability of refactoring options. • the awareness of the capabilities and limitations of software tools (e.g., techniques covered, availability for progr. languages).

to existing process models (Leppänen et al., 2015; Haendler and Frysak, 2018)), we subdivide the refactoring process into the following two basic activities (also see Sections 2.1 and 3.2):

(**A**) *identifying and assessing refactoring candidates/opportunities*, and

(**B**) *planning and performing refactoring steps*.

Knowledge Categories in Refactoring. The knowledge categories describe an increasing complexity from *factual* knowledge (e.g., about basic elements/terms in software refactoring), via *conceptual* knowledge (including conceptual combinations between these elements), *procedural* knowledge (on *how to* perform the candidate-identification and refactoring techniques) finally to *meta-cognitive* knowledge (consisting of reflexive aspects about strategies, contextual aspects and knowledge about one's own capabilities and limitations with regard to refactoring activities). Table 1 reports the knowledge categories according to Section 4.1.

Cognitive-process Levels in Refactoring. Each knowledge category (see above) can be cognitively processed on six different levels (see Table 2) by the software developer or learner respectively; from *remembering* knowledge that has been previously consumed, via *applying* it in a concrete refactoring situation and more advanced cognitive levels such as the *analysis* and *evaluation*, to finally *creating* novel products. The combination of these two dimensions forms a two-dimensional matrix that allows for precisely specifying competence levels. For instance, a learner demonstrating that she can analyze and differentiate existing options for code modifications in order to realize a certain refactoring technique can

Table 2: Levels of cognitive processes of Bloom's revised taxonomy (Bloom et al., 1956; Krathwohl, 2002) applied to key activities in software refactoring (A and B); this table extends (Haendler and Neumann, 2019b).

Process Level	Cognitive Processes related to *identifying and assessing refactoring candidates* (A)	Cognitive Processes related to *planning and performing refactoring steps* (B)
(1) Remember	*Remembering* the knowledge for identifying and assessing refactoring candidates. In a refactoring context, remembering can be important in terms of *recognizing* a bad smell based on given terms of symptoms or *recalling* the symptoms for certain types of refactoring candidates.	*Remembering* the knowledge on planning and performing refactoring (steps) is important in terms of *recognizing* a set of given modifications as refactoring or as *recalling* the rules for applying a refactoring technique.
(2) Understand	*Understanding* the knowledge for identifying and assessing refactoring candidates can manifest in *explaining* the rules/symptoms for identifying certain smells or in *describing* examples of smell types.	*Understanding* the knowledge for planning and performing refactoring steps can be demonstrated by *interpreting* or *explaining* techniques or algorithms for refactoring.
(3) Apply	*Applying* the knowledge for identifying and assessing refactoring candidates can express as concretely *executing* the rules for candidate identification in a (smaller) code base (*procedural knowledge*) or by *applying* strategies to discard false-positives (meta-cognitive knowledge).	*Applying* the knowledge for planning and performing refactoring steps can be, for example, demonstrated by *executing* the code modifications for a certain refactoring technique (*procedural knowledge*) or by *applying* strategies for prioritizing refactorings.
(4) Analyze	*Analysis* in the context of candidate identification/assessment can be expressed by *investigating* the system's source code and design as well as the candidate's structural and behavioral dependencies.	For performing the refactoring steps, *analysis* can also be demonstrated by checking the structural and behavioral dependencies of the code fragment to be modified.
(5) Evaluate	*Evaluating* the knowledge on identifying refactoring candidates can be expressed by *comparing* and *prioritizing* refactoring candidates (according to *meta-cognitive* aspects).	In performing refactoring, the *evaluation* level is, for example, demonstrated by *comparing* and *selecting* options/paths addressing certain candidates.
(6) Create	The ability of *creating* a novel, coherent whole can be demonstrated by 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.	

be assigned the competence to *analyze* (cognitive-process level 4) *conceptual* knowledge (knowledge category II) for performing refactoring (activity B).

5 CASE STUDY

In order to demonstrate the applicability of the framework, we use it to evaluate two exemplary training environments for software refactoring.

5.1 Training Environments

For the case study, we focus on two training environments, i.e. a tutoring system (Haendler et al., 2019) (as an example for instructional learning) and a serious game (Haendler and Neumann, 2019b) (as an example for an educational game). Both environments are implemented as browser-based development environments where the software artifacts (source code) can be modified by a client without the need of installing any additional software locally. The goal of both environments is to address active learning for accomplishing and improving practical competences (elaborated in detail in Section 5.2).

Tutoring System. In (Haendler et al., 2019), we introduced an interactive tutoring system for software refactoring (called REFACTUTOR). It provides immediate feedback to the users regarding the functional correctness and the software-design quality of the (modified) source code. For this purpose, the results of run-time regression tests as well as reverse-engineered diagrams (reflecting the *as-is design*) of the *Unified Modeling Language (UML)* (Object Management Group, 2017) are presented to the user and can be compared with the intended (*to-be*) design (also in UML) pre-specified by the instructor (see Fig. 2). In (Haendler et al., 2019), two exemplary exercise scenarios are described, i.e. *test-driven development* and *design refactoring*. For the following analysis, we focus on the design-refactoring exercise, for which a source-code fragment with design smells is given to the user; in addition, run-time tests ensure the correct behavior. The user's task is then to remove the design issues by restructuring the source code oriented to the

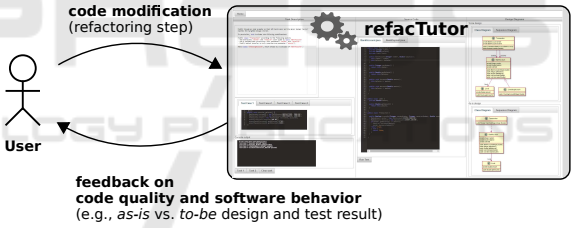

Figure 2: Exercise interaction supported by the REFACTUTOR tutoring system; for details, see (Haendler et al., 2019).

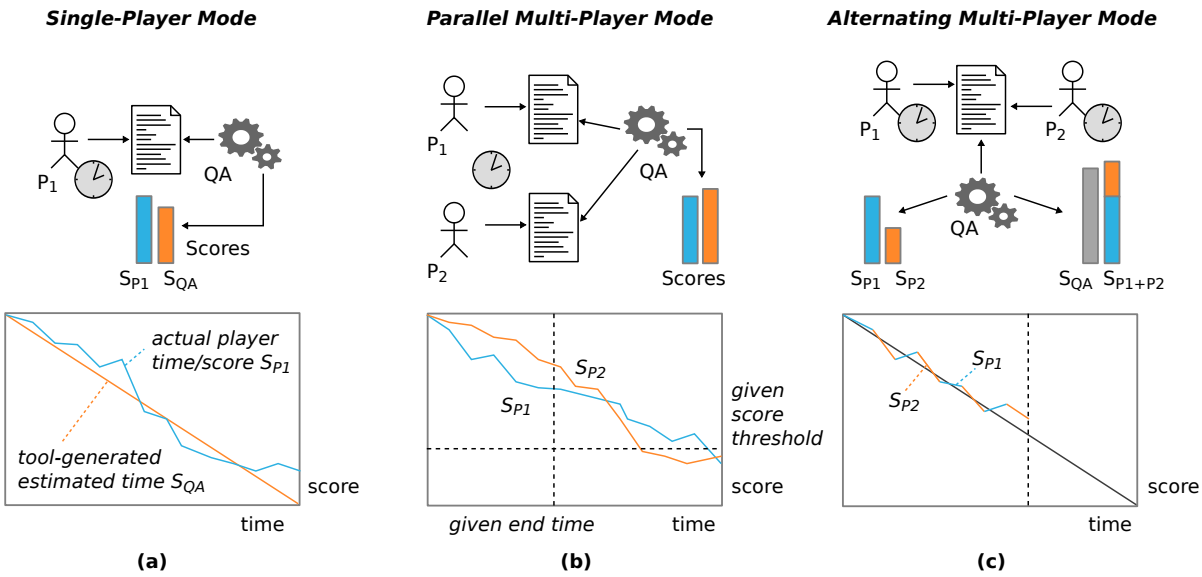

Figure 3: Exemplary refactoring game modes with game constellations (top) and corresponding leader-boards with score progression (bottom); for details, see (Haendler and Neumann, 2019b).

to-be design, while preserving the behavior (i.e. tests finally have to pass). For further details and exemplary exercises, please see (Haendler et al., 2019).

Serious Game. In (Haendler and Neumann, 2019b), we investigated serious games and proposed a game for software refactoring. After each game move (that includes a modification of the source code), immediate feedback on the functional correctness (via integrated run-time tests) and on the technical-debt score is reported to the user. For this purpose, quality analyzers (QA) such as (Campbell and Papapetrou, 2013) or (CoderGears, 2018) for measuring the technical-debt score (calculated in person hours required to repay the debt via refactoring) are integrated. From this basic move cycle multiple game modes can be derived, of which three exemplary are depicted in Fig. 3. For our analysis, we focus on the *single-player game mode*; see (a) in Fig. 3. As already elaborated in (Haendler and Neumann, 2019b; Haendler and Neumann, 2019a), the addressed competences depend a lot on the provided decision support. For this case study, only the test result and the debt score are provided to the user. For further details on the game, please see (Haendler and Neumann, 2019b).

5.2 Allocation of Competences

In the following, the competence levels in terms of the addressed knowledge and cognitive-process dimensions specified in Section 4.2 are allocated to selected training activities supported by the two environments (see above, Section 5.1). We distinguish between *prerequisite* (necessary to perform the activities) and *target* competences (a.k.a. learning objectives; see (Paquette, 2007) and Fig. 1). In particular, the addressed competences are allocated within a two-dimensional matrix (see Fig. 4) correlating the knowledge categories (I–IV) and cognitive-process levels (1–6) for the two basic refactoring activities (A and B; see Section 4.2).

Competences Addressed by the Tutoring System. Performing the *design-refactoring* exercise provided by REFACTUTOR (Haendler et al., 2019) requires *conceptual* knowledge on different cognitive levels and aims at mediating *procedural* and *practical* competences for performing refactoring (see ① and cells colored yellow in Fig. 4).

In particular, the exercise requires the following ***prerequisite competences*** (see PC_{Tutor} in Fig. 4): the learner should already be able to *remember*, *understand* and *apply* (levels 2, 3, and 4) techniques for refactoring (activity B) on a conceptual level (category II), also including at least (conceptual) hypotheses on options and combinations of code modifications to realize a refactoring technique. In turn, the training exercise aims at mediating the following ***target competences*** (see TC_{Tutor} in Fig. 4): By practically refactoring the source code for improving the underlying software design, the learners demonstrate the ability to practically select refactoring options and apply algorithms for performing refactoring steps for

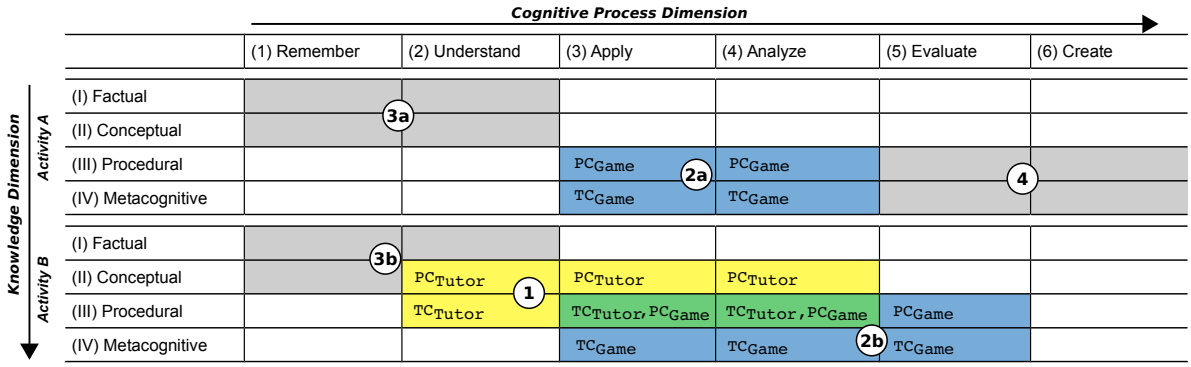

Figure 4: Prerequisite (PC) and target competences (TC) of selected training exercises in a tutoring system (Tutor) (Haendler et al., 2019) and a serious game (Game) (Haendler and Neumann, 2019b) allocated to knowledge categories (I–IV; vertical) and cognitive-process levels (1–6; horizontal) of Bloom's revised taxonomy for educational objectives (Krathwohl, 2002) applied to identification and assessment of candidates (activity A; top rows) as well as to planning and performing refactoring techniques (activity B; bottom rows).

removing a given bad smell. This represents *understanding*, *applying* and *analyzing* of *procedural* knowledge for performing refactoring steps (i.e. category III, levels 2, 3 and 4, and activity B).

Competences Addressed by the Serious Game. Performing the *single-player game mode* in (Haendler and Neumann, 2019b) based on the test result and the technical-debt score as decision support only, already requires *procedural* knowledge both for identifying refactoring candidates (see 2a in Fig. 4) and for performing the refactoring steps (see 2b).

In particular, as ***prerequisite competences*** the *practical* and *analytic* procession (i.e. levels 3 and 4) of *procedural* knowledge (III) for identifying and refactoring smells are needed (activities A and B; see cells colored blue in Fig. 4). In addition, the evaluation of procedures to plan/perform refactorings is required, i.e. not only the knowledge on refactoring options, but also on algorithms to realize them (activity B, level 5, category III). In turn, the game mode focuses on mediating ***target competences*** in terms of *meta-cognitive* knowledge (category IV) for identifying candidates and planning/performing refactorings (activities A and B). This includes the *application* and *analysis* of strategies for selecting candidates for refactoring (levels 3 and 4). Moreover, by competing against the debt score, it also fosters the comparison and critical reflection (i.e. *evaluate*, level 5) of options and strategies for refactoring.

5.3 Exemplary Training Path

The case study has shown that the selected activities of the two training environments cover certain distinguishable competence levels. In particular, the design-refactoring exercise in the tutoring system mainly addresses refactoring activities and already requires *conceptual* knowledge for performing refactoring (see 1 in Fig. 4). The serious game, in turn, demands *procedural* knowledge on the *practical* and *analytic* level, by which it represents a very advanced training environment (see 2a and 2b in Fig. 4). Given that developers already possess *conceptual* knowledge in performing refactoring techniques (in terms of *understanding* the concepts), a (short) training path can be described by combining training activities by the tutoring system and the serious game.

However, nevertheless some competence levels are not addressed by the activities, for which other training activities could be applied. For example, for training (and assessing) the *remembering* and *understanding* of knowledge in the lower categories (of *factual* and *conceptual* knowledge; i.e. levels 1 and 2 and categories I and II), (gamified) ***quizzes*** (e.g., with multiple-choice questions) could be applied (see 3a and 3b in Fig. 4). Moreover, after playing the serious game and in order to address the highest cognitive levels, simulations in terms of *project-based learning* (***capstone project*** (Bastarrica et al., 2017)) could, for example, be used to plan and develop a tool for the automated smell detection. This activity focuses on the *creation* of a novel product, but also includes the *analysis* and *evaluation* of detection criteria and metrics (i.e. categories III and IV and levels 4, 5 and 6 of activity A; see 4 in Fig. 4).

6 DISCUSSION

The aim of the proposed framework was to structure and systematize competences for software refactoring in order to support assessment and training.

To demonstrate the applicability of the framework, we evaluated exemplary training activities of two selected environments by allocating the corresponding prerequisite and target competences.

6.1 Limitations

For our framework, we have focused on knowledge and cognitive-process dimensions only (Krathwohl, 2002). In addition, other perspectives on competences can be applied. For example, Bloom's taxonomy also defines *psychomotor* and *affective* dimensions. Moreover, technical skills in other areas of software engineering (such as regarding the usage of UML diagrams; see Section 5) or soft skills (Sedelmaier and Landes, 2014) (e.g., problem-solving, communication) are important for performing refactoring in software projects. In addition, the developer's attitude (e.g., motivation) can have a major impact on the outcomes, which also can be addressed by (educational) games (Hamari et al., 2014).

6.2 Further Potential

As described in Section 5.3, the competence framework also has the potential to help with the planning and evaluation of training paths. Based on the knowledge about the actual competences possessed by the learners and the learning objectives (target competences), a training path of multiple activities can be designed that aims at guiding a learner from basic (i.e. simple/concrete) to proficient (i.e. complex/abstract) competences. This way, university courses for novice developers (starting with low-level competences and possibly with a stronger focus on conceptual aspects) as well as training-on-the-job for experienced software developers (starting with application-level competences, possibly with a stronger focus on procedural and meta-cognitive aspects) can be designed. In addition to analyzing training environments, the framework could be used for assessing the actual competences of software engineers. By comparing these with competences required for performing refactoring activities, the need and requirements for training environments could be identified. From a software-project context it is also interesting to further investigate the return-on-investment of training activities (by comparing costs and benefits).

7 CONCLUSION

In this paper, we have introduced a framework for the assessment and training of competences in the field of software refactoring. In particular, the framework provides a specification of competence levels according to Bloom's revised taxonomy for educational objectives along the dimensions of knowledge categories and cognitive-process levels. Using the framework, prerequisite and target competence levels of training environments can be specified, which has been illustrated via a case study with two selected training environments (i.e. a tutoring system and a serious game). In this course, it has also been shown that the allocation of competence levels can support in reasoning about the design of training paths for software refactoring. In addition, we believe that the framework also has the potential to be used for other purposes such as for assessing the actual competences possessed by software developers.

For future work, we plan to apply the framework to evaluate via a user study (including a survey) whether and to what extent our training environments (Haendler et al., 2019; Haendler and Neumann, 2019b) can actually contribute to accomplishing and improving competences in software refactoring.

REFERENCES

Alavi, M. and Leidner, D. E. (2001). Knowledge management and knowledge management systems: Conceptual foundations and research issues. *MIS quarterly*, pages 107–136.

Avgeriou, P., Kruchten, P., Ozkaya, I., and Seaman, C. (2016). Managing technical debt in software engineering (dagstuhl seminar 16162). In *Dagstuhl Reports*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Bastarrica, M. C., Perovich, D., and Samary, M. M. (2017). 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)*, pages 137–145. IEEE.

Bloom, B. S. et al. (1956). Taxonomy of educational objectives. vol. 1: Cognitive domain. *New York: McKay*, pages 20–24.

Bork, D. (2019). A framework for teaching conceptual modeling and metamodeling based on Bloom's revised taxonomy of educational objectives. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*, pages 7701–7710.

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

Campbell, G. and Papapetrou, P. P. (2013). *SonarQube in action*. Manning Publications Co.

CoderGears (2018). JArchitect. http://www.jarchitect.com/ [July 31, 2019].

Dreyfus, S. E. and Dreyfus, H. L. (1980). A five-stage model of the mental activities involved in directed skill acquisition. Technical report, California Univ Berkeley Operations Research Center.

Elezi, L., Sali, S., Demeyer, S., Murgia, A., and Pérez, J. (2016). A game of refactoring: Studying the impact of gamification in software refactoring. In *WS Proc. of XP2016*, pages 23:1–23:6. ACM.

Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *It Professional*, 2:17–23.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. (2016). A review-based comparative study of bad smell detection tools. In *Proc. of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 18:1–18:12. ACM.

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

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

Haendler, T. and Frysak, J. (2018). Deconstructing the refactoring process from a problem-solving and decision-making perspective. In *Proc. of the 13th International Conference on Software Technologies (ICSOFT)*, pages 363–372. SciTePress.

Haendler, T. and Neumann, G. (2019a). Ontology-based analysis of game designs for software refactoring. In *Proc. of the 11th International Conference on Computer Supported Education (CSEDU)*, volume 1, pages 24–35. SciTePress.

Haendler, T. and Neumann, G. (2019b). Serious refactoring games. In *Proc. of the 52nd Hawaii International Conference on System Sciences (HICSS)*, pages 7691–7700.

Haendler, T., Neumann, G., and Smirnov, F. (2019). An interactive tutoring system for training software refactoring. In *Proc. of the 11th International Conference on Computer Supported Education (CSEDU)*, volume 2, pages 177–188. SciTePress.

Hamari, J., Koivisto, J., and Sarsa, H. (2014). Does gamification work?–a literature review of empirical studies on gamification. In *Proc. of 47th Hawaii International Conference on System Sciences (HICSS)*, pages 3025–3034. IEEE.

Kitchenham, B. A., Travassos, G. H., Von Mayrhauser, A., Niessink, F., Schneidewind, N. F., Singer, J., Takada, S., Vehvilainen, R., and Yang, H. (1999). Towards an ontology of software maintenance. *J. Software Maintenance: Research and Practice*, 11(6):365–389.

Krathwohl, D. R. (2002). A revision of Bloom's taxonomy: An overview. *Theory into practice*, 41(4):212–218.

Kruchten, P., Nord, R. L., and Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *IEEE software*, 29(6):18–21.

Leppänen, M., Lahtinen, S., Kuusinen, K., Mäkinen, S., Männistö, T., Itkonen, J., Yli-Huumo, J., and Lehtonen, T. (2015). Decision-making framework for refactoring. In *Proc. of 7th International Workshop on Managing Technical Debt (MTD)*, pages 61–68. IEEE.

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

Moha, N., Gueheneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Murphy-Hill, E., Parnin, C., and Black, A. P. (2012). How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18.

Object Management Group (2017). Unified Modeling Language (UML), Superstructure, Version 2.5.1. https://www.omg.org/spec/UML/2.5.1 [July 31, 2019].

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign Champaign, IL, USA.

Paquette, G. (2007). An ontology and a software framework for competency modeling and management. *Educational Technology & Society*, 10(3):1–21.

Parnas, D. L. (1994). Software aging. In *Proc. of 16th International Conference on Software Engineering*, pages 279–287. IEEE.

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

Rolim, R., Soares, G., D'Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., Suzuki, R., and Hartmann, B. (2017). Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, pages 404–415. IEEE Press.

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

Schach, S. R. (2007). *Object-oriented and classical software engineering*, volume 6. McGraw-Hill New York.

Sedelmaier, Y. and Landes, D. (2014). Software engineering body of skills (SWEBOS). In *2014 IEEE Global Engineering Education Conference (EDUCON)*, pages 395–401. IEEE.

Smith, S., Stoecklin, S., and Serino, C. (2006). An innovative approach to teaching refactoring. In *ACM SIGCSE Bulletin*, volume 38, pages 349–353. ACM.

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

Tempero, E., Gorschek, T., and Angelis, L. (2017). Barriers to refactoring. *Communications of the ACM*, 60(10):54–61.

Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. (2008). JDeodorant: Identification and removal of type-checking bad smells. In *Proc. of 12th European Conference on Software Maintenance and Reengineering (CSMR 2008).*, pages 329–331. IEEE.