Ontology-based Analysis of Game Designs for Software Refactoring

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

Keywords: Software Refactoring, Gamification, Serious Games, Game Design, Game Analysis, Domain Ontology, Software Engineering Education and Training.

Abstract: Acquiring practical competences in computer programming and software engineering is challenging. Software refactoring, in particular, is considered an important and useful quality-assurance technique, but due to the perceived difficulties and risks of performing it, often neglected in practice. Still, it received little attention by software engineering education and training so far. Games are a popular means for fostering motivation as well as mediating and improving practical competences by providing an enjoyable and playful environment. However, for instructors it is challenging to develop and apply game designs that address certain learning objectives, which is important for integrating the game into existing or planned learning and training paths, e.g., in the framework of university courses or training units for (experienced) software developers. In this paper, we propose an ontology that aims to support the analysis and design of games in the field of software refactoring. We apply a structured process for creating a unifying domain ontology bridging core concepts from three related fields, i.e. game design (a), software refactoring (b), and competence management (c). The resulting ontology is then represented as a meta-model in terms of a UML class diagram and reflects concepts important for refactoring-game designs. We describe ontology-based options for game design and illustrate the use of the ontology by analyzing existing refactoring-gaming approaches. In addition, we reflect applying the ontology for reasoning about novel game designs and discuss further potential of the approach.

1 INTRODUCTION

Acquiring practical competences in computer programming and software engineering is challenging, since it requires in-dept knowledge and experience. Software refactoring, in particular, is considered an important and useful quality-assurance technique to reduce a system's maintenance costs by improving its internal quality (Kruchten et al., 2012), but which is difficult to acquire for software developers. Due to the perceived difficulties and risks of performing it, refactoring is often avoided in practice (Tempero et al., 2017). Even though textbooks with rules and best practices are available (Fowler et al., 1999; Suryanarayana et al., 2014), they can barely mediate the skills for actually reviewing larger code bases for refactoring opportunities or performing non-trivial refactorings. Still, it received little attention by software engineering education and training so far; see, e.g., (Haendler et al., 2019).

Games are in general a popular means for fostering motivation as well as mediating and improving practical competences by providing a playful environment (Hamari et al., 2014). In recent years, several approaches for serious gaming and gamification in programming and software engineering have been proposed; for an overview, see (Pedreira et al., 2015; Miljanovic and Bradbury, 2018; Alhammad and Moreno, 2018). In turn, in the field of software refactoring, so far only a few gaming approaches can be identified; see, e.g., (Raab, 2012; Elezi et al., 2016; Haendler and Neumann, 2019).

However, for instructors there are no guidelines or tools helping to develop and apply game designs that address certain learning objectives, which is important for integrating the game into existing or planned learning and training paths, e.g., university courses, or training units for software developers. In order to be able to systematically design and develop games for refactoring with the purposes of addressing certain practical competences or of fostering the motivation to perform refactorings, the conceptual field needs to be systematically structured.

In this paper, we propose an approach for the development of an ontology for designing and analyzing software-refactoring games. For this purpose, we reuse and combine relevant concepts from three related domain areas, i.e., game design (a), software

Haendler, T. and Neumann, G.
Ontology-based Analysis of Game Designs for Software Refactoring.
DOI: 10.5220/0007878300240035
In *Proceedings of the 11th International Conference on Computer Supported Education (CSEDU 2019)*, pages 24-35
ISBN: 978-989-758-367-4

refactoring (b), and competence management (c). In particular, it was useful, to break these domain areas further down to 5 different domains. The resulting domain ontology is documented in terms of a UML class diagram and aims to support in better understanding the concepts for designing games. For this purpose, besides the concepts and concept relations, also design options in terms of ontology instances are described in detail. This way, the ontology represents a step towards the structured design of games for software refactoring. The applicability of the ontology is illustrated via the ontology-based analysis of three existing game designs for software refactoring. In addition, we also reflect how the ontology can be used to develop novel game designs. This paper provides the following contributions:

- development of a domain ontology for representing relevant concepts for game designs in software refactoring.
- documentation of the refactoring-game ontology as a meta-model in terms of a UML class diagram including a description of game-design options (i.e., ontology instances).
- ontology-based analysis of (exemplary) existing refactoring-game designs.

Fig. 1 gives a high-level overview of the approach. For developing a domain ontology for software-refactoring games, relevant concepts of three related domain areas (i.e., software refactoring, competence management, and game design) are reused, combined and extended (where necessary).

The resulting ontology can be seen as an abstraction of concrete refactoring-game designs. Existing game designs (i.e., ontology instances) can be analyzed and classified by the ontology. Moreover, the ontology can support in reasoning about the development of novel game designs.

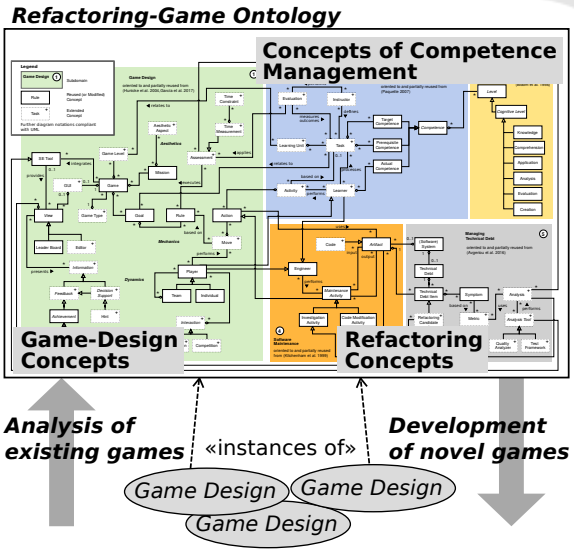

Figure 1: Conceptual overview of developing a domain ontology for analysis and design of refactoring games (for the resulting ontological model, see Fig. 2).

Structure. The remainder of this paper is structured as follows. In Section 2, we reflect on background (and related work) in software refactoring, games for programming (and refactoring in particular), and the use of ontologies (for game design). Section 3 details the process of creating the domain ontology of game designs for software refactoring. In Section 4, we present the resulting ontological model and describe design options for its instantiation. Section 5 illustrates, how the ontology can be applied for analyzing and classifying existing gaming approaches. In Section 6, we reflect on applying the ontology for reasoning about designing novel approaches and discuss further potential. Section 7 concludes the paper.

2 BACKGROUND AND RELATED WORK

2.1 Refactoring and its Challenges

Refactoring is an important quality-assurance technique that helps improving the code quality regarding its maintainability by restructuring the source code without changing the system's external behavior (Opdyke, 1992; Fowler et al., 1999). Despite considered important and helpful, refactoring is often neglected in practice (Tempero et al., 2017). The refactoring workflow can be roughly structured into the following tasks:

(1) identifying refactoring candidates (a.k.a. opportunities) in terms of smells or technical-debt items (Kruchten et al., 2012) and
(2) planning and performing the refactorings step sequences (Fowler et al., 1999).

In this process, software developers are supported by several analysis tools, such as quality analyzers, e.g., *SonarQube* (Campbell and Papapetrou, 2013), and refactoring tools, e.g., *JDeodorant* (Tsantalis et al., 2008) (for identifying refactoring candidates such as smells or for performing refactorings), or test frameworks, e.g., the *XUnit* framework, for automated runtime regression testing (in order to ensure that no errors have been introduced). Despite these advances regarding tool support, developers still perceive difficulties in the refactoring process and a lack of adequate tool support, which often prevents them from

performing refactoring in practice (Tempero et al., 2017). A way to address these challenges can be seen in fostering developers' motivation and practical competences in refactoring.

2.2 Game Designs for (Training) Programming

In recent years, several game designs for training skills in programming and software engineering have been proposed; for serious gaming, see, e.g., (Miljanovic and Bradbury, 2018), for gamification, (Pedreira et al., 2015; Alhammad and Moreno, 2018). However, so far, only a few games and game designs exist for software refactoring in particular. These approaches will be analyzed in Section 5. A goal of this paper is to identify design options based on ontological concepts for creating and analyzing game designs. For this purpose, we adopt and reuse existing frameworks, such as the MDA framework which structures game design through the perspectives of mechanics, dynamics and aesthetics (Hunicke et al., 2004). In addition to this, design options also depend on conditions and aspects in the application areas and purposes of the game, e.g., the fields of software refactoring as well as of learning and training.

2.3 Ontologies

Originating from the philosophical discipline of metaphysics, an ontology represents a structure of entities with the purpose of organizing knowledge (in a certain domain) and managing complexity. In contrast to automatically processed ontologies such as web knowledge graphs (Paulheim, 2017), visually represented ontologies for human consumption need to be manageable in size and are often less formally specified. For example, in information systems research and practice, ontological models often are understood as conceptual models with the objective to share a common understanding of structure of and relationships between concepts in a certain domain (Guizzardi et al., 2002). Concepts are often represented as classes related to other classes, e.g., in terms of a UML class diagram (Object Management Group, 2015).

3 ONTOLOGY DEVELOPMENT

In developing the ontology, we are guided by the structured process proposed by (Noy et al., 2001) consisting of seven distinguished steps from *defining the scope* of the ontology, via *considering the reuse of existing ontologies* to *defining classes and class relations*. In Section 3.1, we describe the scope of the domain ontology and reflect the ontologies identified and considered for reuse. In Section 3.2, we explain how the classes and relations have been defined and what kinds of aspects had to be extended. The resulting ontology is presented in Section 4.1 and the options for creating ontological instances are elaborated in Section 4.2. In Section 5, the ontology is applied to analyze existing gaming approaches.

3.1 Ontology Scope and Reuse of Existing Ontologies

The scope of the ontology is defined by the question: ***How to design games for imparting practical competences for software refactoring?*** (see Section 1).

For this purpose, concepts from the following three fields are relevant:

- **Game Design:** (as the interaction environment),
- **Software Refactoring:** (as the technical domain), as well as
- **Competence Management:** (since the games aim at mediating certain competences to users).

In these three fields, already ontologies and/or taxonomies are established that can be reused as explained in the following.

Game Design. The *MDA framework* for structuring the design concepts from the three perspectives *mechanics*, *dynamics* and *aesthetics* is quite popular and represents a quasi-standard for game design (Hunicke et al., 2004). In particular, *mechanics* comprise the basic game components such as actions performed by the player and the rules. The *dynamics* represent the run-time behavior of the game including feedback mechanisms and user interaction. *Aesthetics*, finally reflect (the more abstract) aspects of emotional and motivational responses evoked in the player (Hunicke et al., 2004). However, the framework reflects the important perspectives for game design, but does not represent the concepts in detail in terms of a domain ontology. (García et al., 2017) builds upon these perspectives and refines them by presenting a framework for the introduction of *gamification in software engineering*, which also includes a structured domain ontology. With the focus on gamification in software engineering, i.e., how software-engineering tools (applied in practice) can be extended by gamification elements, multiple concepts are also related to games in software refactoring (see Section 3.2). We combine both frameworks to cover a large part of concepts of game design.

Software Refactoring. We have identified two popular ontologies and concept formations in the field of software refactoring. First, refactoring can be classified as a *preventive maintenance activity*. The *ontology of software maintenance* proposed by (Kitchenham et al., 1999) structures important software-maintenance activities and in particular covers different tasks. Second, from the perspective of *technical debt management*, the candidates for refactoring (e.g., code smells) are kinds of technical debt items. The ontology proposed by (Avgeriou et al., 2016) represents the concepts relevant for managing technical debt. A combination of both covers some important concepts in the field of refactoring.

Competence Management. For competence management (and learning objectives), popular ontologies and/or taxonomies can be found. First, Paquette provides an ontology for *competence management* (Paquette, 2007). In this work, competences are specialized into *actual*, *prerequisite* and *target competences* (a.k.a. learning objectives). Complementing this ontology, Bloom's taxonomy (Bloom et al., 1956) provides details on *complexity levels* of (cognitive) learning objectives in terms of a hierarchical order of six levels. In (Krathwohl, 2002), a revised version has been proposed, which is applied for our ontology. The taxonomy is especially popular for specifying (levels of) competences in engineering education and training (Britto and Usman, 2015; Masapanta-Carrión and Velázquez-Iturbide, 2018).

3.2 Defining Classes and Relations

The five ontologies from different domains identified in Section 3.1 cover multiple concepts in the field of (educational) software-refactoring games. In the next step, we extracted the concepts relevant for this purpose and checked them for synonyms and homonyms, for further overlaps and possible connection points between the concepts via relationships (such as user roles, kinds of actions or artifacts). While defining classes, class hierarchies and relationships between classes, it became obvious that a few extensions would be necessary. In particular, refactoring-related concepts (such as smells, analysis and analysis tools), basic educational concepts (such as learner, instructor, task, activity, learning unit, and evaluation) as well as further concepts oriented to game design (such as move, assessment, user interaction and feedback) have been extended in order to cover the conceptional scope for refactoring-game design. Details on these extensions are presented in Section 4.

4 REFACTORING-GAME ONTOLOGY

For documenting the ontological concepts for refactoring-game designs, we (1) structure the concepts in terms of a meta-model (see Section 4.1) and (2) reflect on options for instantiating the game ontology (see Section 4.2).

4.1 Ontology Representation

The domain ontology resulting from the applied development process (see Section 3) is documented as a meta-model in terms of a class diagram of the Unified Modeling Language (UML2) (Object Management Group, 2015). Fig. 2 depicts the resulting ontology.[1] The class diagram allows for organizing the identified concepts as classes and their relationships in terms of generalizations and kinds of associations, see (Object Management Group, 2015). The 67 concepts covered by the domain ontology are (visually) structured into the five sub-domains (see ① to ⑤ in Fig. 2) identified for containing concepts relevant for game designs in software refactoring (see Table 1).

Table 1: Ontology sub-domains in Fig. 2.

No	Sub-Domain	Reference
①	Game Design	(Hunicke et al., 2004; García et al., 2017)
②	Competence Management	(Paquette, 2007)
③	Levels of Learning Objectives	(Bloom et al., 1956)
④	Software Maintenance	(Kitchenham et al., 1999)
⑤	Managing Technical Debt	(Avgeriou et al., 2016)

The sub-domains are connected via several links between concepts, e.g., in terms of relationships. Multiple concepts (classes) and relationships have been reused (or slightly modified regarding name or relationships for the purpose of the new contextualization). A few concepts (marked with dashed border and plus sign) have been added in order to (1) allow concept combination and/or (2) provide concepts that are particularly important for game design in software refactoring, but are not covered by the selected reused (sub-) domain ontologies. The notations applied in the diagram are compliant with UML, e.g., class names in italic (such as *Competence* in ② in Fig. 2) indicate abstract concepts that do not allow instantiation directly, but via sub-classes.

In the following, hubs in the diagram (see Fig. 2) with connections between key concepts from multiple

[1] The ontological model is also available for download as SVG/XML file from http://refactoringgames.com/ontology

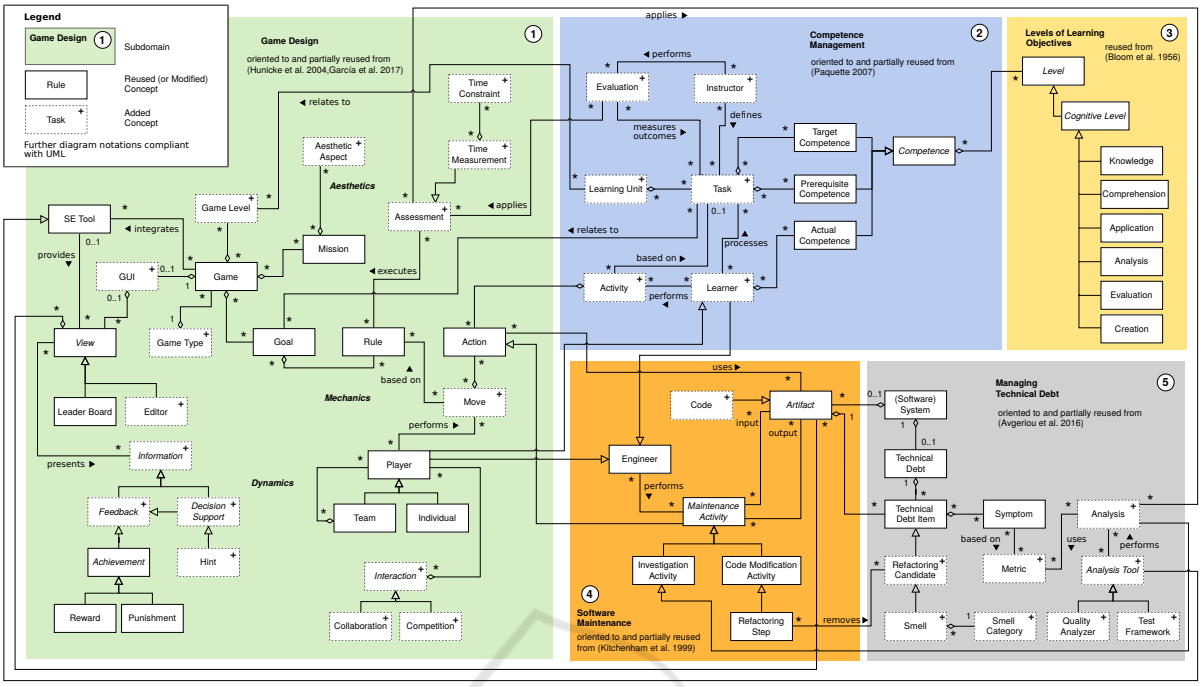

Figure 2: Domain ontology for game design in software refactoring combining concepts from related sub-domains.

sub-domains are described in detail as well as the extensions by new concepts are argued. For further details on reused concepts not explained in detail, please consult the research papers that have introduced the corresponding sub-domain ontologies (Table 1).

User Roles and Competences. A user of refactoring games acts in three roles, i.e., first, in the role of a `Player` that performs game moves and aims at accomplishing the goal of the game (see (1) in Fig. 2), then in the role of a `Learner` that performs training `Activities` (that optionally base on a `Task`) and aims at acquiring and improving their `Actual Competences` (in software refactoring; see (2)), and finally in the role of an `Engineer` that performs refactoring activities or, in general, maintenance-related activities (Kitchenham et al., 1999) (see (4)). This way, the player can be seen as a kind of learner and engineer (i.e. super-classes of player). Besides these user roles, moreover an `Instructor` defines tasks that are oriented to certain `Prerequisite Competences` (necessary to perform the activity) and aim at mediating certain `Target Competences` (i.e., learning objectives) to the learner. Each kind of `Competence` can be characterized by the complexity `Levels`, i.a., the `Cognitive Level`; see (Bloom et al., 1956) and (3) in Fig. 2.

User Activities and Interaction. While playing a game, a user performs `Actions` that are on the one hand part of a game `Move` (from a gaming perspective, see (1) in Fig. 2) and on the other hand part of training `Activity` (optionally) based on a `Task` set by an `Instructor`(from learning perspective, see (2)). Moreover, in the context of software refactoring, an action can be a `Maintenance Activity` that can be expressed as an `Investigation Activity` or a `Code Modification Activity` (e.g., in terms of a concrete `Refactoring Step`, see (4); in (Kitchenham et al., 1999) defined as *enhancement* and more specific as *changed implementation*). A `Player` can be an `Individual` or a `Team` (see (1)). Each player can be part of `Interactions` between multiple players, which can express as `Collaboration` or `Competition`. This way, for instance, also competitions between teams can be captured.

Artifacts and Information. `Artifacts` (such as the software system's source `Code`, test specification or documentation) are used as *input* or *output* of `Actions` (e.g., `Maintenance Activity`) performed by users (see (4) in Fig. 2). Training- and game-play-related artifacts are represented by `Views` (from technical perspective a.k.a. widgets (García et al., 2017)) such as `Editors` or `Leaderboards` (see (1)) . The views also present other kind of `Information`, e.g., in terms of `Decision Support` for supporting users in refactoring activities (such as `Hints`, e.g., the lo-

cation of a refactoring candidate) or in terms of other Feedback to user activities such as the accomplished Achievement (e.g., expressed as Rewards via badges or points).

Smells and Analysis. Each Refactoring Candidate (a.k.a. refactoring opportunity) can be seen as a Technical Debt Item that is part of a System's Technical Debt; see (5) in Fig. 2 and for further details, e.g., (Kruchten et al., 2012; Avgeriou et al., 2016)). Among others, Smells are specific refactoring candidates that can be allocated to a certain Smell Category (such as code or architecture smells). A refactoring candidate (e.g., a smell) can be removed via one ore multiple Refactoring Steps (see (4)). Each debt item manifests via certain Symptoms (Fowler et al., 1999) that are based on (and can be measured via) corresponding Metrics (5). These metrics can be used during Analysis (e.g., for smell detection), which represents a kind of Investigation Activity (see (4)). In general, different kinds of analysis can be performed by Analysis Tools, which are specific Software Engineering (SE) Tools (see (1) and (5)) such as Quality Analyzers (identifying debt items such as smells or measuring the systems technical debt, see, e.g., (Fontana et al., 2012)) or Test Frameworks (ensuring that no error has been introduced). The concepts in the scope of smells and analysis tools have been extended, since they are essential concepts for the refactoring workflow, see, e.g., (Haendler and Frysak, 2018). Further details on concrete tools are provided in Section 4.2.

Evaluation and Assessment. The terms *evaluation* and *assessment* are often used synonym. Also in the field of games and training/education, especially with regard to programming exercises, the meanings are manifold, since it can be seen from different perspectives with different purposes. For the purposes of this ontology, we distinguish between the (automated) Assessment in the gaming context and the Evaluation performed by the Instructor for measuring the learner's activity performance. In particular, an Assessment (see (1) in Fig.2) measures the Player's Moves by executing the defined Rules of game play. In turn, an Evaluation is applied to evaluate the outcomes of the learner's activities against the defined task, e.g., in order to determine the learner's actual competences (see above and (2)). It may be performed by the instructor alone or also supported by a tool-based Analysis (see above). A specific concept of Assessment is Time Measurement, since many games in general require performing actions or moves in a given time frame (see associated class Time Constraint) or the time is a crucial factor in competitions between multiple players (see above).

Mechanics, Dynamics and Aesthetics. The identified concepts of game design can also be loosely divided into the perspectives of the MDA framework (Hunicke et al., 2004). For instance, Action, Move, Rule, Goal as basic game elements can be allocated to *Mechanics* (see (1) in Fig.2). *Dynamics* comprise all concepts directly impacting the runtime behavior of the game, i.e., for instance the kinds of Feedback and Decision Support as well as the player Interaction. *Aesthetics* are expressed in the game Mission associated to one or multiple Aesthetic Aspects.

4.2 Ontology Instances and Game-Design Options

Concrete game designs can be seen as instances of the ontology (also see Fig. 1), while they differ in terms of the concept values. By structuring options of these values, e.g., in terms of possible enumerations, a knowledge base can be built (Noy et al., 2001) that can serve as foundation for analyzing and for designing gaming approaches (see Sections 5 and 6). For this purpose, we will examine the following aspects, which are especially important for refactoring-related games: competence levels for refactoring tasks, smell types, analysis and assessment (including analysis tools), options for decision support, aesthetic aspects (w.r.t. the MDA framework (Hunicke et al., 2004)), and game types. Instances of other (more generic) concepts such as kinds of Views (e.g., bar charts, activity feeds etc.) or kinds of Rewards (e.g., badges, points etc.) are not in scope of this section. Explanations on these can be found in research literature on gamification and/or game-based learning in general, see, e.g., (García et al., 2017).

Refactoring Tasks and Competence Levels. From the perspective of software developers, refactoring can be basically distinguished into the two following tasks (or steps; also see Section 2.1).
(**A**) *identify and assess refactoring candidates*, and
(**B**) *plan and perform refactoring steps*.
Table 2 provides descriptions on the competence levels (see (3) in Fig. 2) of these tasks according to Bloom's revised taxonomy of cognitive learning objectives (Krathwohl, 2002). Practical competences as targeted by serious games or gamification approaches

Table 2: Competences in software refactoring (Haendler and Neumann, 2019) structured by levels of Bloom's taxonomy of cognitive learning objectives.

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.	

can be located on levels 3 to 6 (see Table 2). Levels 1 and 2 are more knowledge-oriented. For further details, also see (Haendler and Neumann, 2019).

Smell Types and Learning Objects. According to the different quality aspects (w.r.t artifact types and/or abstraction levels), several `Smell Categories` and types of `Technical Debt Items` (see ⑤ in Fig. 2) can be distinguished (Alves et al., 2016), such as the following:

- code smells (Fowler et al., 1999)
- software design or architecture smells (Suryanarayana et al., 2014)
- test smells (Bavota et al., 2012)
- requirements smells (Femmer et al., 2017)
- model smells (Misbhauddin and Alshayeb, 2015)

Each `Smell Type` (or `Smell Category`) can be seen as a *learning object*, i.e. as a content item or chunk for a learning unit. For instance, such as a unit for MODULARIZATION smells as sub-group of software design smells, see (Suryanarayana et al., 2014).

Analysis and Assessment. According to the types of smells (see above) and their symptoms, different quality `Metrics` can be applied (Kan, 2002) to identify them, also in different artifact types. For this purpose, two types of `Quality Analyzers` (see ⑤ in Fig. 2) are exemplary listed that pursue the (mostly static) analysis of the internal quality of system artifacts:

- *Smell detection and refactoring recommendation tools* that support identifying smell and refactoring candidates via symptoms based on certain metrics; e.g., *JDeodorant* (Tsantalis et al., 2008) or *DECOR* (Moha et al., 2010).
- *Technical-debt analyzers* applying different metrics for measuring and quantifying a system's debt in terms of person hours to fix (repay) the debt; e.g., *SonarQube* (Campbell and Papapetrou, 2013) *JArchitect* (CoderGears, 2018) or *NDepend* (ZEN PROGRAM, 2018).

Besides the analysis of the internal code quality, test frameworks (such as *XUnit* or scenario tests) are commonly in use in terms of run-time regression tests that ensure that no errors have been introduced while modifying the source code. As described above (Section 4.1), tool-supported analysis can be applied for assessing and evaluating the user's actions and moves. The choice of tool and what information provided to the user (in terms of decision support and feedback) impacts the addressed refactoring competences (see below).

Decision Support and Learning Objectives. Depending on the analysis information provided by analysis tools (see above) presented to the user in terms of decision support and feedback, different competences are addressed (see Table 3).

Via the integration of regression-testing frameworks, the functional correctness (w.r.t. the specified tests) can be assessed. This way, the low-level task (1) in Table 3 of performing *code modifications* that do not change the external behavior of the system (i.e. refactoring) can be automatically assessed, which lays the basis for further tasks and assessments. In order to address the competence of (2) *identifying refactoring candidates* (e.g., code smells), the technical debt score can be provided to the user, which indicates the existence but not the location of the debt item in the system's artifact. For (3) *planning and performing refactoring steps*, finally a combination of regression

Table 3: Decision support provided by analysis tools (horizontal) with addressed refactoring tasks and competence levels (vertical).

Task	regression testing	TD measured	smells identified	ref. options
(1) behavior preserving code modification	◆	–	–	–
(2) identification of ref. candidates	–	◆	–	–
(3) planning and performing ref. steps	◆	◆	◆	–
(4) strategy selection	◆	◆	◆	◆

testing, measured technical debt and also the location of concrete smells (as provided by smell detectors, see above) can be presented to the user. For addressing the competence of (4) *strategy selection*, in addition to the identified smells also the refactoring options (provided by refactoring recommendation tools) can be presented to the user. In this case, the challenge is in prioritizing the given smells and evaluating the already available options for refactoring. For corresponding competence levels, also see Table 2.

Aesthetics. While the `Goal` of a refactoring game can be generally seen in increasing the system quality by removing smells or reducing technical debt score (or parts of that, e.g., identifying smells), the `Mission` as the narrative purpose of the game (as motivation for the user) can differ from this. Via emotional responses from a game (evoked by the applied underlying game dynamics), a user is motivated to accomplish the set game `Mission` (Hunicke et al., 2004). The MDA framework suggests 8 kinds of `Aesthetic Aspects` for game designs (see Table 4).

Table 4: Exemplary aesthetic aspects for game designs (Hunicke et al., 2004).

Fun aspect	Description
(1) Sensation	Game as sense-pleasure
(2) Fantasy	Game as make-believe
(3) Narrative	Game as drama
(4) Challenge	Game as obstacle course
(5) Fellowship	Game as social framework
(6) Discovery	Game as uncharted territory
(7) Expression	Game as self-discovery
(8) Submission	Game as pastime

For refactoring games, the following exemplary conditions can evoke aesthetic aspects. As a *Challenge* aspect can be seen the identification of smells and/or performing of refactorings, especially when performed under time pressure (`Time Constraint`; see ① in Fig. 2). In case the player acts in a certain role in a defined scenario (such as the role of a software developer confronted with quality issues), the *Narrative* aspect can be seen addressed. In general, playing a refactoring game in team or competing against each other for (collectively) improving the code quality may be regarded as *Fellowship* aspect. In addition, also the *Discovery* can be addressed, e.g., by letting the player explore system artifacts for identifying refactoring candidates.

Game Types. With focus on games for the training of technical and engineering-related competences, we can distinguish three `Game Types`, i.e., game-based learning, gamification, and serious game. In particular, the term *game-based learning* reflects the use of games for learning purposes (Prensky, 2003), which also includes simple game designs with the goal to mediate knowledge and understanding (i.e., lower levels of competences in Bloom's taxonomy). *Gamification* can be defined as using game-design elements in non-gaming contexts (Huotari and Hamari, 2012). For software engineering (and refactoring in particular), gamification can be regarded as extending development and engineering activities (which are based on certain SE tools) by gaming elements, also see (García et al., 2017). Moreover, *serious gaming* means playing games with a certain purpose, e.g., educational or training games (Landers, 2014). In addition (and in contrast to game-based learning), serious games are regarded to provide real-world conditions (e.g., via video-based simulations (Connolly et al., 2012)) and to target higher-level competences. Furthermore, serious games and gamification are often grouped together, but they differ regarding the scope/depth of game structure. Basically, serious games aim to provide typical game mechanics (e.g., consisting of game moves and goal/mission), while gamification approaches focus on the (sometimes modest) application of gaming elements such as certain (social) feedback mechanisms (e.g., points, badges, leaderboards).

5 ANALYZING GAME APPROACHES

Here we illustrate by example how the ontology can be used for analyzing existing game designs. In addition, in Section 6, we also reflect on applying the approach for reasoning about novel game designs.

In research literature, only a few approaches for refactoring games can be identified. Based on current

Table 5: Comparison of exemplary refactoring-game approaches according to ontology-based design options.

Design Aspect	**(A)** (Raab, 2012)	**(B)** (Elezi et al., 2016)	**(C)** (Khandelwal et al., 2017)
Tasks and Actions	Identify refactoring opportunities and perform corresponding refactorings in small code examples in the framework of a learning path (with single- and multiple-choice questions)	Identify code smells and select appropriate refactorings in real-world code base and trigger the execution of *Eclipse* refactoring commands	Review source code produced by peers for identifying code smells and bugs
Mission	Master the set learning path by accomplishing the learning units	Collectively improve quality of real-world code; compete against fellow developers while collecting points for performing code refactoring	*Experience the use of gamification elements* while reviewing the code produced by peers for identifying code smells and bugs
Aesthetic Aspect	Challenge, discovery, sensation	Challenge, fellowship, narrative	Challenge, (discovery)
Target Competences	Identify smells and refactoring options as well as perform refactorings (small examples)	Assess and prioritize refactoring candidates in (real-world) code	Identify code smells (and bugs) while reviewing the code (larger examples)
Prerequisite Competences	Understanding the (basic) rules/symptoms for (1) identifying selected smell types and (2) performing refactoring steps	Using *Eclipse* refactoring tool	Understanding the rules/symptoms for identifying selected smell types (and bugs)
Smell types	Basic stylistic code and design smells, based on (Fowler et al., 1999)	Code smells according to *Eclipse's* code smell detector	Stylistic code smells (*clean code*)
Modified Artifacts	Small code examples	Real-world code base	Code base produced by peers (in the framework of a programming exercise)
Views / Feedback	Editor/code, tangible cards/slides (with general refactoring information), interactive screen for visualizing refactoring options (in terms of a graph, see below)	Editor/code, leaderboards, activity feed, progress bar	Editor/code, leaderboards, activity stream
Decision support	Tangible cards with general information on selected smell types as well as dependency graph (between smell types and related refactoring techniques)	List of identified and source-located refactoring candidates as well as refactoring recommendation and automation	General information on symptoms of a selected set of most common code smells
Assessment / Evaluation	Automated assessment of multiple-choice questions enhanced by human evaluation of code-modification tasks	Tracking of command execution and automated assessment in terms of a score via pre-defined difficulty levels	Peer reviews of code (i.e. actual task, no assessment in narrow sense)
Rewards	Several kinds of animations and sounds	Score points and levels	Likes, badges
Tools or games integrated	–	*Eclipse* refactoring tool	Code-review tool
User interaction	Informal, in terms of discussions between users	Competitive and indirect via shared leaderboards	Competitive and indirect via shared leaderboards
Game levels / learning units	Path consisting of sequence of learning units	Documented progress (e.g., levels, bar charts)	–
Type of game	Game-based learning	Gamification	Gamification

systematic literature reviews (SLRs) on serious games in programming (Miljanovic and Bradbury, 2018), gamification in software engineering (education) (Pedreira et al., 2015; Alhammad and Moreno, 2018) and further review, only the following four approaches for applying games in the field of software refactoring have been identified (i.e., with regard to gamification, game-based learning or serious gaming).

(**A**) (Raab, 2012)
(**B**) (Elezi et al., 2016)
(**C**) (Khandelwal et al., 2017)
(**D**) (Haendler and Neumann, 2019)

Since approach (**D**) represents a design for serious games proposed by the authors of this paper, we focus on the other three design approaches for analysis. In Table 5, these identified approaches are evaluated according to ontology-based design options and further concept values of the ontology as described in Section 4.2 . In particular, the values for the following design aspects have been attributed to approaches (A–C): addressed refactoring tasks and actions, the game mission, the addressed aesthetic aspects, the addressed (prerequisite and target) competences, the kinds of refactoring candidates (smell types), the artifacts modified by the users, the provided views and feedback as well as decision-support mechanisms, the techniques of assessment and evaluation, the kinds of rewards, whether pre-existing tools and games have been included, the interactions between users, the kind of game progress and the type of game (e.g., game-based learning or gamification).

As described in Table 5, all three analyzed game designs address i.a. the task (and target competence) of identifying candidates for refactoring smells (i.e., on code level). However, in addition to this, they differ in terms of further tasks and (levels of) competences. These differences are correlated with the choice of game-design options, such as the game mission, the size of artifacts to be analyzed and modified, as well as the kinds of decision and tool support.

In particular, approach (**A**) (Raab, 2012) proposes a game-based learning approach for learning to identify smells and refactoring options as well as to perform refactorings in small code examples (also see Table 5). The exercises (e.g., multiple-choice questions) are framed by a learning path with several learning units. As decision support, tangible cards with general information on selected smell types are provided as well as an interactive screen with a visualization of smell types and related refactoring techniques. the result of multiple-choice questions are assessed automatically, the code modifications by the instructor. For this game, the theoretical knowledge on rules/symptoms for identifying smells and performing corresponding refactorings is required. The target audience can be regarded as primarily novices, e.g., in terms of (university) students aiming to gain first practical refactoring competences.

Approach (**B**) (Elezi et al., 2016) in turn is based on real-world code bases and integrates a refactoring tool (in *Eclipse*) for identifying code smells and selecting appropriate refactorings. The approach can be classified as gamification, since it adds gaming elements (such as leaderboards, activity feed) to a real-world setting with the purpose of fostering motivation. For each successful refactoring, a user scores points. The mission is to collectively improve code quality by competing against fellow developers. The integrated tool provides high-level decision support by suggesting refactoring candidates and then automatically performing them. Thus, this game's technical challenge is to prioritize the candidates. The user's performance is assessed by tracking the command execution within the *Eclipse* refactoring tool and weighting them in scores according to their difficulty level. An important prerequisite competence lays in using the applied refactoring tool. Software developers can generally be regarded as the target group of this approach.

Finally, approach (**C**) (Khandelwal et al., 2017) proposes the gamification in performing code reviews for answering the question whether a playful environment can foster users' motivation. For this purpose, code-review tools are extended by several gaming elements (such as likes and badges). The users then review code developed by peers for identifying smells and bugs. For decision support only general information on symptoms of a selected set of most common code smells are presented to the user. Target audience of this approach are primarily also software developers.

The analysis shows that existing approaches for games in software refactoring (identified in recent SLRs, see above) cover only a few of the possible aspects (i.e., design options) presented in the proposed domain-ontology. In particular, games (B) (Elezi et al., 2016) and (C) (Khandelwal et al., 2017) represent approaches for gamifying software engineering tools (i.e., a refactoring tool and a code-review tool) by adding single gaming elements. Their focus is less on mediating competences than on fostering users' motivation to perform the activities. In contrast, approach (A) (Raab, 2012) proposes game-based learning for small code examples and simple smell types and thus only addresses lower-level competences according to Bloom's taxonomy (see Table 2). All three approaches focus on smells on the code level (i.a., stylistic code smells), whereas more complicated smells can be found, for instance, on the level of software design and architecture (Suryanarayana et al., 2014) or in other kinds of artifacts.

6 DISCUSSION

In this paper, we have developed a domain ontology for representing the concepts relevant for games in the field of software refactoring. We have illustrated how the resulting ontology and the corresponding design aspects and options can be used for analyzing existing games. The analysis has shown by example that the covered concepts are appropriate to analyze and classify game designs for software refactoring. In addition, the analysis indicates that there is further potential for game approaches, especially in the sense of serious games, that aim to acquire practical competences in refactoring (Haendler and Neumann, 2019).

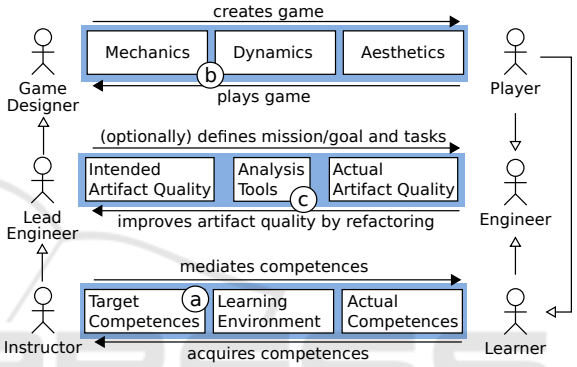

Figure 3: Layers relevant for development of game design.

Ontology-Supported Game Design. In addition to the game analysis, we believe that the ontology can support in designing novel game designs. As starting point for designing games, we can consider the following three scenarios (also see Fig. 3):

- (a) **From target competences to game.** *How to design a game that addresses certain competences?*
- (b) **From existing game to target competences.** *How to use an existing gaming environment to address certain competences?*
- (c) **From existing SE tools to game that addresses target competences.** *How to gamify a development environment in order to address certain competences?*

The developed ontology provides support for reasoning about game design from all three viewpoints. Fig. 3 depicts an overview of the three layers relevant for game designs in software refactoring covered by the ontology. In case (a) (see Fig. 3), the instructor is primarily also in the role of the game designer (i.e., the game infrastructure is devoted to learning purposes). Starting *from prerequisite and target competences*, she can consult the ontology for possible design options in game-design mechanics, e.g., what kinds of actions/moves to be performed and which tools to in-

tegrate, and game dynamics, e.g., what kinds of information presented to the user, and also regarding user interaction. In case (b), the instructor aims to *modify and adapt an existing gaming environment* in order to address certain (target) competences. Depending to existing game mechanics (and integrated tools), she can explore the options for decision support and feedback (provided by the tools) and consider user-interaction and other (motivational) feedback mechanisms. Case (c) describes the *typical gamification approach*, i.e., based on existing SE tools and activities, it is aimed to add gaming elements. Starting from the technical sub-domains in (4) and (5) in Fig. 2, she can (based on applied SE tools) reflect on design options for game mechanics, dynamics and aesthetics (see (1)) in order to address certain (target) competences (see (2) and (3)).

Further Potential. The development of the domain ontology was driven by the motivation to better understand the concepts relevant for designing (training) games in the field of software refactoring. Thus, its scope is focus on corresponding concepts. For a broader application purpose, e.g., for designing and analyzing games in computer programming and software engineering, certain domain-specific concepts need to be extended. However, we believe that the resulting ontology can be used and adapted for other technical domains. For instance, in case of designing games for training requirements-engineering techniques, the concepts in sub-domains (1) to (3) in Fig. 2 can be reused. For this purpose, concepts in (4) and (5) need to be replaced by corresponding concepts (e.g., activities, artifacts and tools) for requirements engineering.

7 CONCLUSION

In this paper, we presented an approach for developing a unifying ontology for representing concepts and design options relevant for (educational) games in software refactoring. For creating the ontology, concepts from five related existing domain ontologies and taxonomies from the fields of game design, software refactoring and competence management have been reused, combined and extended.

As a result of this process, a domain ontology in terms of a meta-model documented as UML class diagram is presented. The ontology provides an overview of concepts (and dependencies between concepts) relevant for games in software refactoring. In addition to the model, possible design options (i.e., ontology instances) are described in detail. Moreover, it has been demonstrated by example that the ontology can be used to analyze existing gaming approaches and provides potential for supporting game designers and instructors in the process of developing novel refactoring-game designs.

Based on the analysis of existing approaches, we see potential for games addressing further aspects in order to promote practical competences and motivation in software refactoring. For future work, we plan to extend our serious-gaming approach (Haendler and Neumann, 2019) to a generic gaming framework for refactoring that provides variable design options to the instructor, e.g., in order to address different competence levels and allow the flexible integration of analysis tools.

REFERENCES

Alhammad, M. M. and Moreno, A. M. (2018). Gamification in software engineering education: A systematic mapping. *J. Systems and Software*, 141:131–150.

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

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.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., and Binkley, D. (2012). An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Proc. of ICSM 2012*, pages 56–65. IEEE.

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

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/ [March 25, 2019].

Connolly, T. M., Boyle, E. A., MacArthur, E., Hainey, T., and Boyle, J. M. (2012). A systematic literature review of empirical evidence on computer games and serious games. *Computers & Education*, 59(2):661–686.

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 *Proc. of the Scientific Workshops of XP2016*, page 23. ACM.

Femmer, H., Fernández, D. M., Wagner, S., and Eder, S. (2017). Rapid quality assurance with requirements smells. *J. Systems and Software*, 123:190–213.

Fontana, F. A., Braione, P., and Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *J. Object Technology*, 11(2):5–1.

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

García, F., Pedreira, O., Piattini, M., Cerdeira-Pena, A., and Penabad, M. (2017). A framework for gamification in software engineering. *J. Systems and Software*, 132:21–40.

Guizzardi, G., Herre, H., and Wagner, G. (2002). On the general ontological foundations of conceptual modeling. In *International Conference on Conceptual Modeling*, pages 65–78. Springer.

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. (2019). 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)*.

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.

Hunicke, R., LeBlanc, M., and Zubek, R. (2004). MDA: A formal approach to game design and game research. In *Proc. of the AAAI Workshop on Challenges in Game AI*, volume 4, pages 1–5. AAAI Press San Jose, CA.

Huotari, K. and Hamari, J. (2012). Defining gamification: a service marketing perspective. In *Proc. of the 16th international academic MindTrek conference*, pages 17–22. ACM.

Kan, S. H. (2002). *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc.

Khandelwal, S., Sripada, S. K., and Reddy, Y. R. (2017). Impact of gamification on code review process: An experimental study. In *Proc. of the 10th Innovations in Software Engineering Conference*, pages 122–126. ACM.

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.

Landers, R. N. (2014). Developing a theory of gamified learning: Linking serious games and gamification of learning. *Simulation & Gaming*, 45(6):752–768.

Masapanta-Carrión, S. and Velázquez-Iturbide, J. Á. (2018). A systematic review of the use of Bloom's taxonomy in computer science education. In *Proc. of the 49th ACM Technical Symposium on Computer Science Education*, pages 441–446. ACM.

Miljanovic, M. A. and Bradbury, J. S. (2018). A review of serious games for programming. In *Joint International Conference on Serious Games*, pages 204–216. Springer.

Misbhauddin, M. and Alshayeb, M. (2015). UML model refactoring: a systematic literature review. *Empirical Software Engineering*, 20(1):206–251.

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.

Noy, N. F., McGuinness, D. L., et al. (2001). Ontology development 101: A guide to creating your first ontology.

Object Management Group (2015). Unified Modeling Language (UML), Superstructure, Version 2.5.0. http://www.omg.org/spec/UML/2.5 [March 25, 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.

Paulheim, H. (2017). Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web*, 8(3):489–508.

Pedreira, O., García, F., Brisaboa, N., and Piattini, M. (2015). Gamification in software engineering–a systematic mapping. *Information and Software Technology*, 57:157–168.

Prensky, M. (2003). Digital game-based learning. *Computers in Entertainment (CIE)*, 1(1):21–21.

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

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.

ZEN PROGRAM (2018). NDepend. http://www.ndepend.com/ [March 25, 2019].