Ontology-Based Analysis and Design of Educational Games for Software Refactoring

Thorsten Haendler(✉) and Gustaf Neumann

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

Abstract. Despite being regarded as necessary to ensure a system's maintainability and extensibility, software refactoring is often neglected in practice due to difficulties and risks perceived by software developers. Still, refactoring received little attention by software engineering education and training so far. Educational games are a popular means for enhancing practical competences as well as increasing motivation of learners. For instructors, however, it is challenging to develop and apply games in order to address certain learning objectives, which is important to integrate the games into existing or planned training paths. In this article, we propose an ontology that aims to support the analysis and design of games for teaching and training software refactoring. In particular, we create 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 represented as a UML class diagram that reflects concepts and concept relations important for educational refactoring games. We describe ontology-based design options and demonstrate the use of the ontology by analyzing existing games for software refactoring. In addition, we also present an exemplary process for developing novel games based on the ontology and illustrate its applicability by designing a non-digital card game.

Keywords: Software refactoring · Game design · Game analysis · Gamification · Serious games · Domain ontology · Software engineering education and training

1 Introduction

Software refactoring is considered a useful technique to ensure a system's internal quality during software maintenance and evolution [45], but which is difficult to learn and master for software developers [15,43]. Due to the perceived difficulties and risks of performing it, refactoring is often neglected in practice [33,40,53]. Even though textbooks with rules and best practices are available [15,52], they

H. C. Lane et al. (Eds.): CSEDU 2019, CCIS 1220, pp. 602–628, 2020.
https://doi.org/10.1007/978-3-030-58459-7_29

can barely mediate the practical skills for reviewing larger code bases to identify refactoring opportunities or applying non-trivial refactoring techniques efficiently. Still, software refactoring received little attention by software-engineering education and training so far (see e.g. [25] and Sect. 2).

Games are in general a popular means for increasing motivation as well as mediating and improving practical competences by providing a playful and interactive training environment [26]. In recent years, several approaches for serious gaming and gamification in programming and software engineering have been proposed; for an overview, see [1,36,47]. In turn, in the field of software refactoring, so far only a few gaming approaches can be identified; see e.g. [12,23,49] and Sect. 5. However, for instructors there are no guidelines or tools helping to develop or apply game designs that address the objectives of a certain training situation (e.g. regarding aspects of competences or motivation), which is especially important for integrating the game into existing or planned training paths. In order to be able to systematically analyze and develop game designs for software refactoring, the conceptual field needs to be systematically structured.

In the conference paper presented at *CSEDU-2019* [21], we have developed a game ontology that aims to support the analysis and development of (educational) game designs for software refactoring. For this purpose, we reused and combined relevant concepts from three related domain areas, i.e. game design (a), software refactoring (b), and competence management (c). In particular, it was useful, to break these domain areas further down to five different domains. The resulting domain ontology has been 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 has been described in detail. This way, the ontology represents a step towards the structured design of games for software refactoring. This post-conference revision of the *CSEDU-2019* publication [21] incorporates important extensions, also in response to feedback by reviewers and conference attendees. In particular, this article provides the following additional contributions:

1. We extend the ontological model by integrating further relevant concepts, such as knowledge-categories for competence specification and player types (see Sect. 4).
2. We systematize the ontology-based analysis of existing games (e.g. by allocating addressed prerequisite and target competences) and include further recently published approaches into this analysis (see Sect. 5).
3. We present a process for the ontology-driven development of educational refactoring games. To demonstrate its applicability, we also provide a comprehensive example in terms of developing a non-digital card game. The game is then also classified according to ontology-based design options (see Sect. 6).

The applicability of the ontology is illustrated by two means. First, by performing an ontology-based analysis of five existing educational games for software refactoring. Second, by developing an exemplary game in terms of a non-digital card game for mediating basic competences for software refactoring. Figure 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 the development of novel game designs.

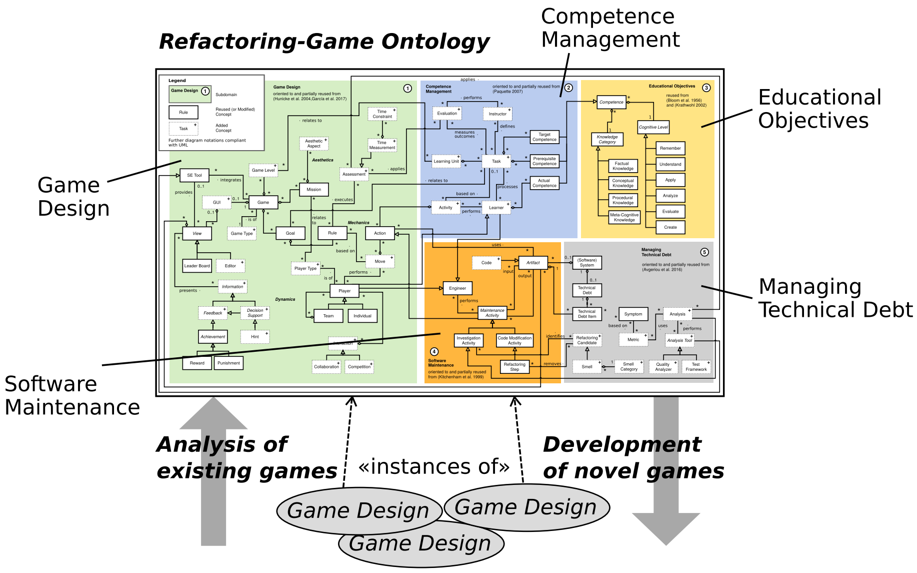


Fig. 1. Conceptual overview of the developed domain ontology, which combines concepts of domains relevant for the analysis and design of educational games for software refactoring [21]. The resulting ontological model is depicted in Fig. 2.

Structure. The remainder of this article is structured as follows. In Sect. 2, we reflect on background (and related work) in software refactoring, games for software development (and software refactoring in particular), and the use of ontologies (especially for game design). Section 3 details the process of creating the domain ontology containing concepts relevant for educational game designs in the field of software refactoring. In Sect. 4, we present the resulting ontological model and describe game-design options for its instantiation. Section 5 illustrates, how the ontology can be applied for analyzing and classifying existing approaches of games for training software refactoring. In Sect. 6, we focus on applying the ontology for developing novel refactoring games and illustrate the exemplary development of a card game for learning principles of software refactoring. In Sect. 7, we discuss the further potential of the presented approach. Sect. 8 concludes the article.

2 Background and Related Work

2.1 Challenges in Software Refactoring

Software refactoring is an important technique for improving a software system's maintainability and extensibility by restructuring the source code without changing the system's observable behavior [15,43]. Despite considered useful, refactoring is often neglected in practice [40,53]. The refactoring workflow can be roughly structured into the following two activities (also see [19]):

(A) identifying and analyzing candidates (a.k.a. opportunities) for refactoring in terms of bad smells or technical-debt items [33] and
(B) planning and performing the refactoring steps [15].

In this process, software developers are supported by several analysis tools, such as quality analyzers, e.g. *SonarQube* [9], and refactoring tools, e.g. *JDeodorant* [54] (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 activities in practice [40,53]. A way to address these challenges can be seen in fostering developers' motivation and practical competences in software refactoring.

2.2 Game Designs for (Training) Software Development

In recent years, several game designs for training skills in programming and software engineering have been proposed; for an overview of serious-gaming approaches, see e.g. [36], for gamification, see e.g. [1,47]. Besides other environments for training software refactoring (such as tutoring systems [24,50,51]), however, so far only a few games and game designs exist for software refactoring in particular. These approaches will be analyzed in detail (based on the developed ontology) in Sect. 5. A goal of this article is to identify design options based on ontological concepts for creating and analyzing (educational) game designs for software refactoring. 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 [27]. 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 and managing complexity (in a certain domain). In contrast to automatically processed ontologies such as web knowledge graphs [46], 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 [17]. Concepts are often represented as classes related to other classes, e.g. in terms of a UML class diagram [42].

3 Development of the Domain Ontology

In the development of the domain ontology, we are guided by the structured process proposed by [41] 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 Sect. 3.1, we describe the scope of the domain ontology and reflect the ontologies identified and considered for reuse. In Sect. 3.2, we explain how the classes and relations have been defined and what kinds of aspects had to be extended.

3.1 Ontology Scope and Reuse of Existing Ontologies

The scope of the ontology is defined by the question: ***How to design games that foster motivation and practical competences in software refactoring?*** 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 below.

Game Design. The *MDA framework* [27] for structuring the design concepts from the three perspectives *mechanics*, *dynamics* and *aesthetics* is quite popular and represents a quasi-standard for game design. 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 [27]. However, the framework reflects the important perspectives on game design, but does not represent the concepts in detail in terms of a domain ontology. The framework [16] builds upon these perspectives and refines them with the purpose of introducing *gamification* to 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 Sect. 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. [31] structures software-maintenance activities and covers important related aspects. 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. [3] represents the concepts relevant for managing technical debt. A combination of both covers some important concepts in the field of refactoring.

Competence Management. For the management of competences (and learning objectives), several ontologies and/or taxonomies with different purposes can be found. First, Paquette provides an ontology for *competence management* [44]. In this work, competences are specialized into *actual*, *prerequisite* and *target competences* (a.k.a. learning objectives). Complementing this ontology, Bloom's taxonomy [7] provides details on *complexity levels* for learning objectives in terms of six hierarchical levels for specifying cognitive processes. In [32], Krathwohl proposes a revised version of this taxonomy by extending the cognitive processes by four knowledge categories (orthogonal to the process dimension). This taxonomy makes it possible to specify competence levels more precisely than previous approaches by using a two-dimensional matrix. This revised version is especially popular for specifying (levels of) competences in engineering education and training [8,35]. Our domain ontology integrates this taxonomy applied to software refactoring.

3.2 Defining Classes and Relations

The five ontologies from different domains (see Sect. 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 Sect. 4.

4 Refactoring-Game Ontology

For documenting the ontological concepts for refactoring-game designs, we (1) structure the concepts in terms of a class diagram (see Sect. 4.1) and (2) describe options for instantiating the game ontology (see Sect. 4.2).

4.1 Ontology Representation

The domain ontology resulting from the applied development process (see Sect. 3) is documented as a class diagram of the Unified Modeling Language (UML2) [42]. Figure 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 [42]. 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. 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 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 (see above).

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 [31] (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. According to the revised version of Bloom's taxonomy [32], each kind of `Competence` can be characterized by two dimensions, i.e. the

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

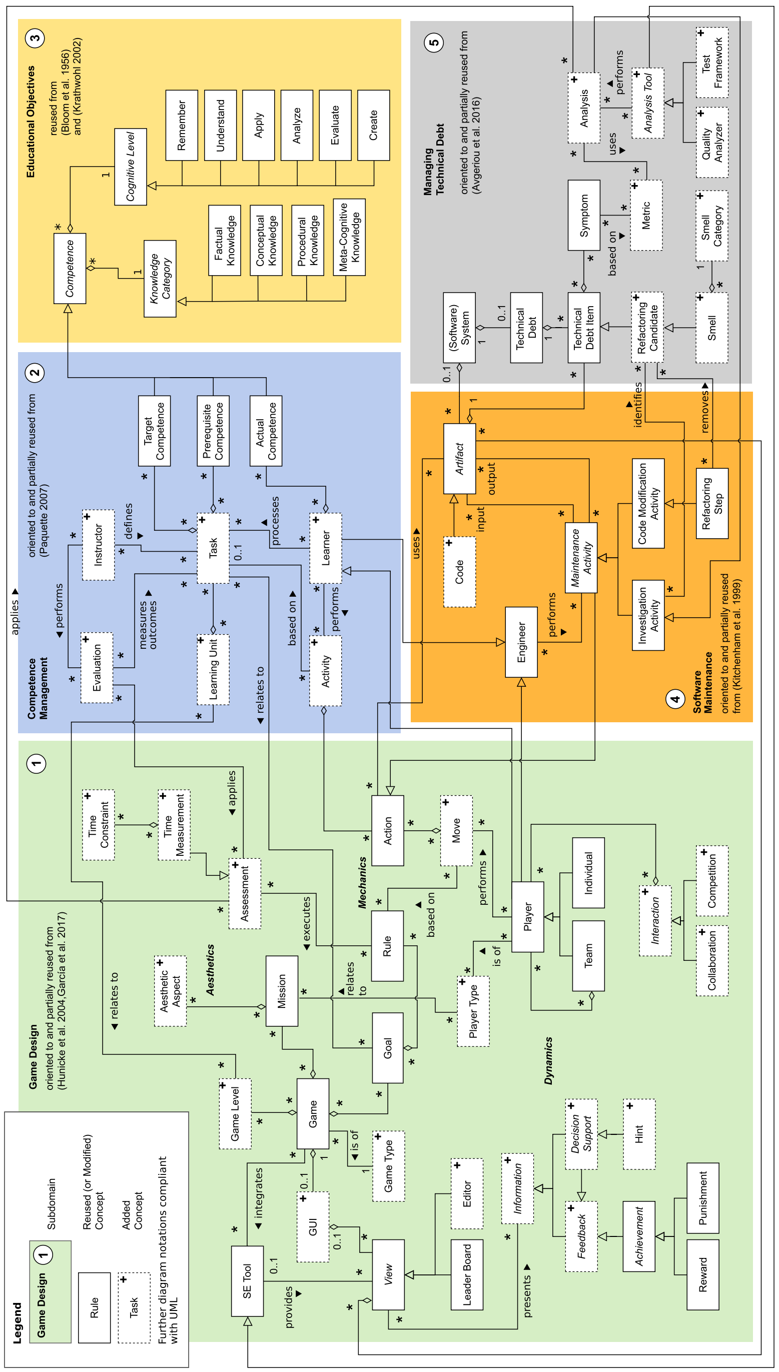


Fig. 2. Domain ontology for the analysis and design of educational games in software refactoring combining concepts from related sub-domains; this ontological model represents a revised and extended version of [21].

Knowledge Category and the Cognitive Level, which are combined orthogonally (see ③ in Fig. 2). In particular, four levels of knowledge are distinguished, i.e. (I) Factual, (II) Conceptual, (III) Procedural and (IV) Meta-Cognitive Knowledge, which can be processed on six hierarchical cognitive levels, which are (1) Remember, (2) Understand, (3) Apply, (4) Analyze, (5) Evaluate and (6) Create.

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 [31] 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 [16]) 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 location 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. [3,33]). 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 [15] 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. [14]) 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. [19]. Further details on concrete tools are provided in Sect. 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 `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 [27]. 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 run-time 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`, which address certain `Player Types` (see [5]).

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 game-design options, e.g. in terms of possible enumerations, a knowledge base can be built [41] that can serve as foundation for analyzing and for designing gaming approaches (see Sects. 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 [27]), and game types. Instances of other (more generic) concepts such as kinds of `Views` (e.g. bar charts 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. [16].

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 Sect. 2.1 and [19]):

(A) *identifying and assessing candidates for refactoring*, and
(B) *planning and performing refactoring steps.*

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

Fig. 3. Prerequisite (`PC`) and target competences (`TC`) of selected training exercises in a tutoring system (`Tutor`) [24] and a serious game (`Game`) [23] allocated to knowledge categories (I–IV; vertical) and cognitive-process levels (1–6; horizontal) of Bloom's revised taxonomy for educational objectives [32] applied to *identification and assessment of candidates* (activity A; top rows) as well as to *planning and performing refactoring techniques* (activity B; bottom rows). For details, see [20].

In Fig. 3, for each of these activities a two-dimensional matrix of four `Knowledge Categories` (vertical) and six `Cognitive-Process Levels` (horizontal) is depicted (also see ③ in Fig. 2 and [32]). Based on specifications for each category and level, the addressed competences for software refactoring can be precisely allocated, e.g. in terms of `Prerequisite` (required to perform an activity) and `Target Competences` (a.k.a. learning objectives; see [44] and ② in Fig. 2). Figure 3 also provides an allocation of competences addressed by exercises of two exemplary training environments. For further details, please see [20].

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 [2], such as the following: code smells [15], software design or architecture smells [52], test smells [6], requirements smells [13], model smells [37]. 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, e.g. the category Modularization smells as sub-group of software design smells, see [52].

Analysis and Assessment. According to the types of smells (see above) and their symptoms, different quality `Metrics` can be applied [29] 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* [54] or *DECOR* [38].
– *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* [9] *JArchitect* [10] or *NDepend* [55].

Table 1. Decision support (horizontal; run-time tests (RT), measured technical debt (TD), identified bad smells (BD), refactoring options (RO)) with addressed tasks (vertical; 1–4) [21].

Task	RT	TD	BS	RO
(1) Behavior preserving code modification	♦	–	–	–
(2) Identification of refactoring candidates	–	♦	–	–
(3) Planning and performing refactoring steps	♦	♦	♦	–
(4) Strategy selection	♦	♦	♦	♦

Table 2. Exemplary aesthetic aspects for game designs with description [27].

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

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 to ensure that no errors have been introduced while modifying the source code. As described above (Sect. 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 1). 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 1 of performing *code modifications* that do not change the external behavior of the system (i.e. refactoring) can be automatically assessed, which forms 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 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.

Aesthetics and Player Types. While the `Goal` of a refactoring game can be generally seen in increasing the system's 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` [27].

The MDA framework suggests 8 kinds of `Aesthetic Aspects` for game designs (see Table 2). 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 the code for identifying refactoring candidates. In addition, game users can be characterized by certain `Player Types`, which partially correspond to the aesthetic aspects above. A popular set is defined by Bartle, who distinguishes the following four types of players as corners along two axes (i.e. *from an emphasis on players to an emphasis on world* and *from acting to interacting*) [5]:

- *Achievers* focus on acting on the world by attaining status and achieving set goals (e.g. points).
- *Explorers* are interested in interacting with the (game) world by discovering the unknown (searching for surprises).
- *Socializers* are interested in interacting with other players by developing a network of friends.
- *Killers* are interested in acting on other players in terms of demonstrating their superiority. Thus they focus on rank and direct competition.

For example, in refactoring games, these player types can be addressed as follows. *Achievers* can be motivated by collecting points for identifying/removing smells via refactoring. Discovering opportunities for refactoring while reviewing the source code can address *explorers* (see the aesthetic aspect of *Discovery* above). By providing ways to work together, e.g. in a group, can especially motivate *socializers* (see *Fellowship*).

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 [48], 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 [28]. 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 [16]. Moreover, *serious gaming* means playing games with a certain purpose, e.g. educational or training games [34]. In addition (and in contrast to game-based learning), serious games are regarded to provide real-world conditions (e.g. via video-based simulations [11]) 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 Ontology-Based Analysis of Games

Here we illustrate by example how the ontology can be used for analyzing existing game designs. In addition, in Sect. 6, we also reflect on applying the approach to develop novel games. In research literature, only a few game approaches in the field of software refactoring be identified. Based on current systematic literature reviews (SLRs) on serious games in programming [36], gamification in software engineering and engineering education [1,47] and further review, only the following five 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) *CodeSmellExplorer: Tangible Exploration of Code Smells and Refactorings* [49]
(B) *A Game of Refactoring: Studying the Impact of Gamification in Software Refactoring* [12]
(C) *Impact of Gamification on Code Review Process: An Experimental Study* [30]
(D) *Serious Refactoring Games* [23]
(E) *CodeArena: Inspecting and Improving Code Quality Metrics using Minecraft* [4]

In Table 4, the results of analyzing these approaches are presented according to ontology-based game-design options and further concept values of the ontology as described in Sect. 4.2. In particular, the values for the following design aspects of approaches (A–E) have been allocated: *addressed refactoring tasks and actions, the game mission, the addressed aesthetic aspects, player types, addressed audience, the addressed (prerequisite and target) competences (allocated to competence levels according to the matrix in* Fig. 3), *the smell types, the artifacts modified by the users, the provided views and feedback mechanisms, the techniques of assessment and evaluation, the reward kinds, included tools or games, the user interactions, the kind of game progress and the type of game.*

In particular, approach (**A**) [49] 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 4). The exercises (e.g. multiple-choice questions) are framed by a learning path. As decision support, tangible cards with general information on selected smell types are provided as well as an interactive screen with a visualization between smells and refactorings. Multiple-choice questions are assessed automatically, the code modifications by the instructor. For this game, the conceptual 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 competences. Approach (**B**) [12] 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) 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. The approach targets software developers. Approach (**C**) [30] proposes the use of gamification for performing code reviews to answer 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 (beginner and proficient). In approach (**D**) [22,23], we investigated serious games and proposed a game for software refactoring. After each game move (which 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, pre-existing analysis tools such as quality analyzers (QA) (e.g. [9] or [10]) 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, such as single- and multi-player modes either competing against a predefined benchmark (technical debt) or against other players. The game's focus is set on efficiently performing refactorings and on applying appropriate prioritization strategies. Targeted are also software developers in terms of beginner and proficient. Approach (**E**) [4] *CodeArena* is a 3D game built on *Minecraft Forge* that allows for fighting selected bad smells via code refactoring based on an additional editor view. The user's code base represented by a 3D world, in which the refactoring candidates (identified via a code analysis tool) are represented as monsters and can be accessed by walking through the world. The game especially aims at motivating novices to perform refactoring activities.

The analysis shows that existing refactoring games (as identified in recent SLRs and further research) only cover a few of the possible design options presented in the proposed domain-ontology. In particular, from a *motivational perspective*, it can be seen that certain aesthetic aspects (such as *sensation* and *fellowship*) and player types (such as *killers* and *socializers*) are not or barely addressed. Thus, potential for further games lays in addressing user interaction (e.g. collective activities or direct user competition) as well as providing a visually more sophisticated environment (such as video/3D games). In addition, regarding the *addressed competences*, it can be observed that games (B–E) already demand the *application* of at least *conceptual* and mostly *procedural* knowledge and focus on improving these practical competences. Lower process levels and knowledge categories are addressed by approach (A) only. In turn, there is little attention by games (except approach (D)) on *meta-cognitive* knowledge and reflexive cognitive processes (i.e. the levels of *analysis* and *evaluation*).

6 Ontology-Based Development of Games

In addition to the game analysis, the game-design options identified based on the developed domain ontology can support in designing novel (educational) games for software refactoring. In the following, at first, exemplary scenarios for developing game designs are discussed (Sect. 6.1. Then we demonstrate the ontology-based game development by designing an exemplary non-digital card game for software refactoring (Sect. 6.2).

6.1 Scenarios for Developing Games

As starting point for designing educational games (for software refactoring), we can consider the following three scenarios (also see Fig. 4):
ⓐ **From target competences to game.** *How to design a game that addresses certain competences?*
ⓑ **From existing game to target competences.** *How to use an existing gaming environment to address certain competences?*
ⓒ **From existing SE tools to game that addresses target competences.** *How to gamify a development environment in order to address certain competences?*

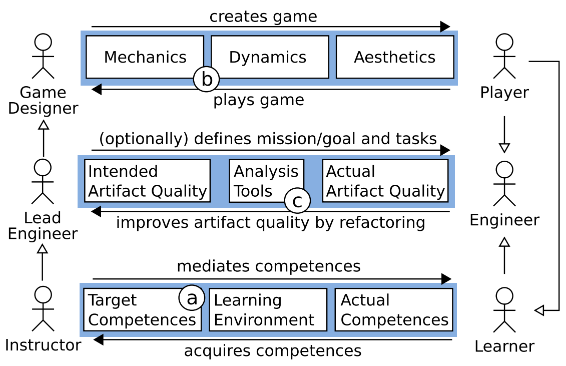


Fig. 4. Layers and stakeholders relevant for the development of game designs in software refactoring [21].

The developed ontology provides support for reasoning about game design from all three viewpoints. Figure 4 depicts an overview of the three layers relevant for game designs in software refactoring covered by the ontology. In case ⓐ (see Fig. 4), 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 integrate, to game dynamics, e.g. what kinds of information presented to the user, and also regarding user interaction. In case ⓑ, the instructor aims to *modify and adapt an existing gaming environment* in order to address certain (target) competences. Depending on 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 ⓒ describes the *typical gamification approach*, i.e. it is aimed to add gaming elements to existing SE tools and activities in order to increase fun in learning or performing the activity. Starting from the technical sub-domains in ④ and ⑤ in Fig. 2 (including the applied SE tools), she can reflect on design options for game mechanics, dynamics and aesthetics (see ①) in order to address certain (target) competences (see ② and ③).

6.2 Development of a Card Game for Software Refactoring

In order to demonstrate the applicability of the ontology for developing games for training software refactoring, we have developed a non-digital card game called Refactory, which aims at mediating basic principles of software refactoring to novices. The applied design process conforms to scenario ⓐ (see above and Fig. 4), which is driven by an analysis of actual competences possessed by players and the competences to be acquired by playing the game.

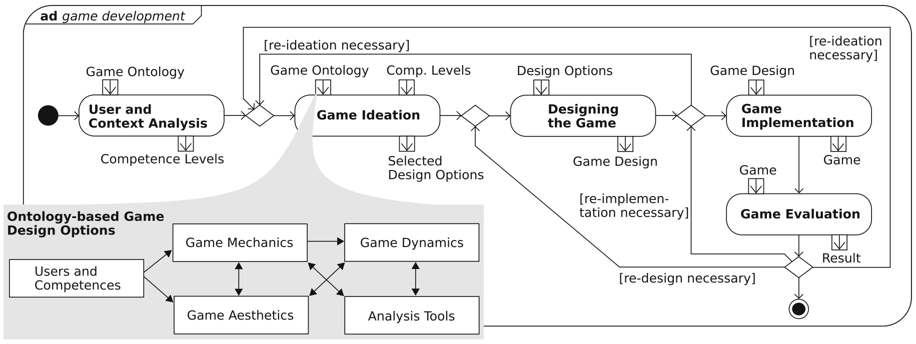


Fig. 5. UML activity diagram representing an exemplary process model for developing educational games (oriented to [39]; top) with a detail view on dependencies between groups of game-design options based on the domain ontology (bottom).

The development is guided by a process model for developing educational games depicted as a UML activity diagram in Fig. 5. The model is oriented to a method for designing gamification [39]. In the following, the process steps of *user and context analysis*, *game ideation*, *design*, *implementation* and *evaluation* are explained and illustrated for the development of the card game.

User and Context Analysis. According to scenario ⓐ (see Fig. 4), the game development can be driven by the objective to mediate certain target competences to an audience possessing certain actual competences. The card game addresses novices (especially students) that already possess a basic understanding of refactoring (i.e. in terms of *factual* knowledge). In particular, its objective is (1) to motivate the players for learning more about software refactoring and (2) to mediate basic refactoring-related competences in terms of being able to *conceptually* combine refactoring techniques to remove bad code smells as well as to weigh up costs and benefits of performing refactoring activities.

Game Ideation. In order to provide activities that address the learning objectives defined above, ideas for designing the game are then developed, for which the ontology is applied. The bottom diagram in Fig. 5 illustrates the dependencies between groups of game-design options. It also shows the course from an *analysis of users and competences* to *game mechanics* (e.g. activities) and *game aesthetics* (e.g. fun factors). In dependency of these concepts, the *game dynamics* (e.g. feedback mechanisms) and *analysis tools* are investigated.

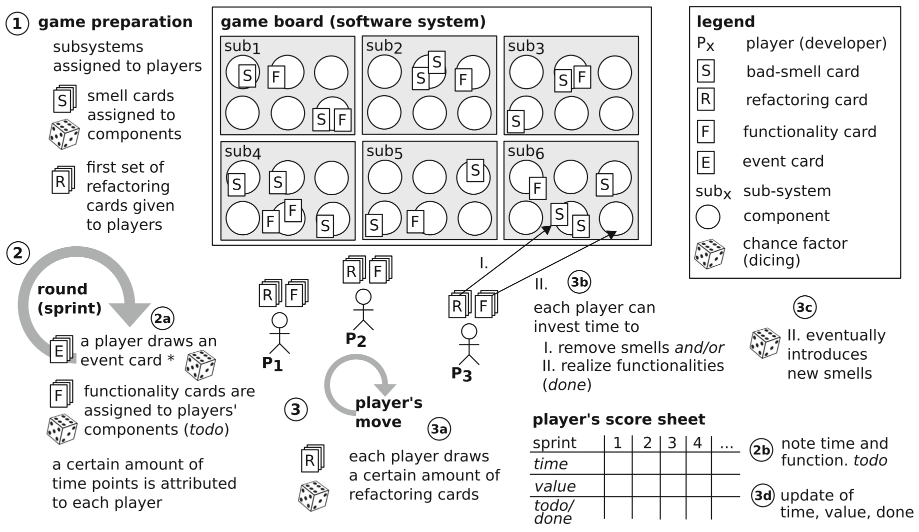
Game Mechanics. At first, the basic user actions and mechanisms in the game have been defined, which is to apply and combine refactoring techniques on a conceptual level. Moreover, players should be confronted with the dilemma for investing limited time, e.g. whether to extend a system by functionalities directly or to perform refactorings beforehand, which can in turn reduce extension costs (see e.g. [33]). By focusing on the conceptual level and certain aspects of refactoring, development-related complexities are avoided and the entry barriers are low. For this reason, a non-digital game including a game board and cards has been preferred (also compare the results of the analysis in Sect. 5).

Game Aesthetics. In dependency of the targeted audience and learning objectives as well as in accordance with the selected game mechanics, the aesthetics are then considered. The basic idea is to provide a conceptual simulation of a software project, where players in the role of software developers (i.e. *narrative* aspect) are confronted with a software system that is affected by technical debt concretely manifesting via bad code smells. In multiple sprints, the players' mission then is to increase the system's value by realizing given functionalities. In particular, the players compete against each other with regard to points they have earned for realizing the functionalities (i.e. addressing *achiever* players). Moreover, the system components are affected with smells, which makes is practically more costly (in terms of invested time points) to extend them by functionalities. So, the players have to balance between realizing functionalities and removing smells by combining refactoring techniques (*challenge* aspect). In addition, the discovery of options for combining refactoring techniques for removing bad smells also addresses the *discovery* aspect and *explorer* players.

Game Dynamics and Analysis Tools. Built on the game mechanics and in order to support the targeted aesthetic aspects (see above), the game run-time behavior and user interactions can be considered (*dynamics*; see [27]). During each round, which represents a sprint, functionalities are assigned by dicing to components, which are assigned to certain players. Moreover, each player selects a certain number of cards that represent refactoring techniques, which can be used in combination to remove bad smells (also represented as cards). For removing smells and realizing time points (given each round) are consumed. The resulting scores are noted in each player's score sheet. The player manually assess the correctness of applying a combination of refactoring techniques for removing a certain kind of bad smell. For this reason, the available techniques are listed on the back of the card. Thus, further analysis tools do not have to be included for this specific non-digital game.

Applied Game-Design Options. Table 3 reports the results of the ideation process by presenting the selected ontology-based design options applied in the card game (correlating to the comparison provided in Table 4).

Game Design. Figure 6 finally illustrates the resulting game design of REFACTORY in terms of a constellation of important elements and activities. A game board represents a software system consisting of multiple sub-systems (each comprising multiple software components in turn), which are assigned to players at start (see (1) in Fig. 6). Goal of the game basically is to increase the value of the given software system by realizing new functionalities assigned every sprint (see (2)). However, the players are impeded in realizing new functionalities by preexisting issues in the software design (see (1)), which complicate the modification



1 game preparation
subsystems assigned to players
smell cards assigned to components
first set of refactoring cards given to players
game board (software system)
sub1
sub2
sub3
sub4
sub5
sub6
legend
Px player (developer)
S bad-smell card
R refactoring card
F functionality card
E event card
subx sub-system
component
chance factor (dicing)
2 round (sprint)
2a a player draws an event card *
functionality cards are assigned to players' components (todo)
a certain amount of time points is attributed to each player
P1
P2
P3
I.
II.
3b each player can invest time to
I. remove smells and/or
II. realize functionalities (done)
3c II. eventually introduces new smells
3 player's move
3a each player draws a certain amount of refactoring cards
player's score sheet
sprint 1 2 3 4 ...
time
value
todo/done
2b note time and function. todo
3d update of time, value, done

Fig. 6. Constellation of the developed exemplary non-digital card game *Refactory* for learning principles of software refactoring.

Table 3. Ontology-based design options applied for designing the card game.

Design aspect	Card game *Refactory*
Tasks and actions	Investing time points to realize given functionalities and remove bad smells affecting software components by meaningfully combining refactoring techniques
Mission	Collectively (as developer team) increasing a software system's value by realizing given functionalities and removing identified bad smells
Aesthetic aspect	Narrative, challenge (and discovery)
Addressed audience	Novices (especially students)
Player type	Killers, achievers (and explorers)
Prerequisite competences (allocated according to [20])	*Understanding* basic terms of refactoring (*factual knowledge*; i.e. category I and level 2 of activity B)
Target competences (allocated according to [20])	*Applying* (combining) refactoring techniques to remove bad smells (on *conceptual* level; i.e. category II and level 3 of activity B) as well as *meta-cognitive knowledge aspects* in terms of strategies for refactoring expressed as balancing the time/costs and benefits of performing refactoring (i.e. category IV and levels 3 and 4 of activity B)
Smell types	Code/design smells (but generic concept)
Modified artifacts	Position of cards (representing functionalities and bad smells) located at components as part of a software system (represented by a game board)
Views/Feedback	Game board, game cards (representing smells, refactoring techniques, functionalities, and events)
Decision support	Cards with general information on smell types and refactoring techniques
Assessment/Evaluation	Manual assessment (e.g. whether the applied refactoring techniques match the addressed smell types)
Rewards	Value points
Tools or games integrated	–
User interaction	Competitive and indirect, e.g. by comparing value points in score sheets
Game levels/Learning units	–
Type of game	Serious game (project simulation)

and extension of components and thereby increase development time. These bad smells can be removed by combining cards representing refactoring techniques, which are also drawn in each round (see $\textcircled{3}$). In addition, events (such as code reviews or a feature check by the customer) impact the value scores (see $\textcircled{2a}$).

Game Implementation and Evaluation. Based on the game design, the actual game is implemented (e.g. as a software system or as a physical card and board game). An evaluation of the game can be performed by conducting survey with players asking for their impressions and experiences of playing the game, such as regarding its ability to promote competence acquisition or motivation. A more comprehensive way of evaluation can be seen in measuring the acquisition of competences [20]. Practically, the development process applied above and illustrated in Fig. 5 requires multiple iterations, especially in terms of refinements and adaptations from game ideation to game design. Further details on the card game including a prototype and a first evaluation in terms of a game-play study with players' feedback can be found in [18].

7 Discussion

In this article, we have developed a domain ontology for representing the concepts relevant for educational 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 and also for developing a novel game. 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 gaming approaches, especially in the sense of serious games that aim to acquire practical competences in refactoring [23]. Moreover, the exemplary ontology-driven development of a card game for learning principles of software refactoring has demonstrated that the ontology can support in reasoning about design potions available for game design. However, the development process also includes creative aspects to evolve ideas, especially when it comes to game mission (and fun aspects) and how game dynamics can support it. Basically, the development of the domain ontology was driven by the motivation to better understand the concepts relevant for designing games for mediating (especially) practical competences in software refactoring. Thus, its scope limited accordingly. For a broader application purpose, e.g. for designing and analyzing games in (other fields of) software engineering, certain domain-specific concepts need to be replaced or extended. However, we believe that the resulting ontology can be used and adapted for other technical domains. For instance, in case of designing or analyzing educational games for training requirements-engineering techniques, the concepts in sub-domains $\textcircled{1}$ to $\textcircled{3}$ in Fig. 2 can be reused. For this purpose, concepts in $\textcircled{4}$ and $\textcircled{5}$ need to be replaced by corresponding concepts for requirements engineering (e.g. activities, artifacts and tools).

Table 4. Comparison of five exemplary approaches for (educational) games in the field of software refactoring according to ontology-based design options; this table extends [21].

Design aspect	**(A)** *CodeSmellExplorer* [49]	**(B)** Refactoring Gamification [12]	**(C)** Code-Review Gamification [30]	**(D)** Serious Refactoring Game [23]	**(E)** *CodeArena* [4]
Tasks and actions	Identifying refactoring opportunities and perform corresponding refactorings in small code examples in the framework of a learning path (e.g. multiple-choice questions)	Identifying code smells and select appropriate refactorings in real-world code base and trigger the execution of *Eclipse* refactoring commands	Reviewing source code produced by peers for identifying code smells and bugs	Reviewing a code base (small to real-world) for bad smells and prioritizing their removal by refactoring	Selecting identified bad smells in 3 world represented as monsters and removing them by applying refactoring steps
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 refactorings	*Experience the use of gamification elements* while reviewing the code produced by peers for identifying code smells and bugs	Reducing the technical debt score by meaningfully selecting refactoring candidates and removing them efficiently via refactoring	Eliminate all monsters (representing bad smells) by refactoring the smelly code structures
Aesthetic aspect	Challenge, discovery, sensation	Challenge, fellowship, narrative	Challenge, (discovery)	Challenge, discovery, fellowship	Challenge, discovery, narrative, sensation
Player types	Explorers, sozializers	Achievers, explorers	Achievers (explorers)	Achievers, explorers, socializers	Achievers, explorers (killers)
Target audience	Novice (students)	Proficient	Beginner/Proficient	Beginners/Proficient	Novice (students)
Prerequisite competences (allocated according to [20])	*Understanding* the (basic) rules/symptoms for identifying selected smell types and performing refactoring steps (I/II, 1/2, A/B)	Using *Eclipse* refactoring tool to identify candidates and apply techniques for refactoring (III, 2, A/B)	*Understanding* and *applying* the rules/symptoms for identifying selected smell types (and bugs) *conceptually* (II, 2/3, A)	*Applying* and *analyzing procedural* knowledge for identifying and removing bad smells via refactoring (III, 3/4, A/B)	*Applying* procedural knowledge for applying refactoring techniques (III, 3, B)
Target competences (allocated according to [20])	Identify smells and refactoring options as well as perform refactorings (small examples; II/III, 2/3, A/B)	Assess and prioritize refactoring candidates in (real-world) code (IV, 4/5, A/B)	Identify code smells (and bugs) while reviewing the code (larger examples; III, 3, A)	*Meta-cognitive* knowledge for identifying candidates and performing refactorings (III/IV, 3/4, A/B)	*Applying procedural* knowledge for applying refactoring techniques (III, 3, B; *improvement*)
Smell types	Basic stylistic code and design smells, based on [15] (but generic concept)	Code smells according to *Eclipse's* code smell detector	Stylistic code smells (*clean code*; depends on used analysis tool)	Code and design smells (depends on used analysis tool)	Selected code smells (e.g. duplicated code, unit complexity and size; depends on used analysis tool)

(*continued*)

Table 4. (*continued*)

Design aspect	**(A)** *CodeSmellExplorer* [49]	**(B)** Refactoring Gamification [12]	**(C)** Code-Review Gamification [30]	**(D)** Serious Refactoring Game [23]	**(E)** *CodeArena* [4]
Modified artifacts	Small code examples	Real-world code base	Code base produced by peers (within of a programming exercise)	Code base (small to real-world)	Code base (small to real-world)
Views/Feedback	Editor/code, tangible cards/slides (with general refactoring information), graph/screen for visualizing refactoring options	Editor/code, leaderboard, activity feed, progress bar	Editor/code, leaderboard, activity stream	Editor/code, leaderboard	Editor/code, visualization via a 3D world with monsters (representing smells)
Decision support	Tangible cards with general information on selected smell types as well as dependency graph (between smell types and 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	Technical-debt score and results from run-time regression tests (and more, which depends on configuration)	Visual localization of bad smells
Assessment/Evaluation	Automated assessment of multiple-choice questions and manual 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)	Automated assessment via run-time tests and technical-debt analyzer	Automated assessment via code analyzer
Rewards	Several kinds of animations and sounds	Points and levels	Likes, badges	Reduced technical debt score	Points
Tools or games integrated	–	*Eclipse* refactoring tool	Code-review tool	Test framework, technical-debt analyzer (and smell detectors)	*Minecraft Forge* (gaming environment) and *Rascal* (code analysis)
User interaction	Informal, in terms of discussions between users	Competitive and indirect via shared leaderboards	Competitive and indirect via shared leaderboards	Competitive and indirect via shared leaderboards as well as collaborative	Planned
Game levels/Learning units	Path consisting of sequence of learning units	Documented progress (e.g. levels, bar charts)	–	*Planned*	–
Type of game	Game-based learning	Gamification	Gamification	Serious game/Gamification	Serious game

8 Conclusion

In this article, we have 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 UML class diagram is presented. The ontology provides an overview of concepts relevant for educational games in software refactoring. In addition to the model, possible game-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 to support game designers and instructors in the development of novel games for software refactoring. In particular, five refactoring games identified via current systematic literature reviews and further research have been analyzed based on the game-design options provided by the ontology. The analysis indicates potential for games addressing further aspects in order to promote practical competences and motivation in software refactoring. In turn, the development of a non-digital card game for learning principles of software refactoring has shown by example that the identified ontology-based design options can also support the reasoning and planning of game designs.

A future research direction can be seen in extending the scope of the ontology-based analysis by investigating whether and to what extent the approach is applicable for analyzing and designing games in other fields of software engineering or other domains. Another interesting direction is to systematize the process of developing games and game designs based on the domain ontology and provided game-design options.

References

1. Alhammad, M.M., Moreno, A.M.: Gamification in software engineering education: a systematic mapping. J. Syst. Softw. **141**, 131–150 (2018). https://doi.org/10.1016/j.infsof.2014.08.007
2. Alves, N.S., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Seaman, C.: Identification and management of technical debt: a systematic mapping study. Inf. Softw. Technol. **70**, 100–121 (2016). https://doi.org/10.1016/j.infsof.2015.10.008
3. Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C.: Managing technical debt in software engineering (dagstuhl seminar 16162). In: Dagstuhl Reports, vol. 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016). https://doi.org/10.4230/DagRep.6.4.110
4. Baars, S., Meester, S.: CodeArena: Inspecting and improving code quality metrics using minecraft. In: Proceedings of the Second International Conference on Technical Debt (Tool Demos). IEEE (2019). https://doi.org/10.1109/TechDebt.2019.00023
5. Bartle, R.: Hearts, clubs, diamonds, spades: players who suit muds. J. MUD Res. **1**(1), 19 (1996). http://mud.co.uk/richard/hcds.htm

6. Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., Binkley, D.: An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: Proceedings of ICSM 2012, pp. 56–65. IEEE (2012). https://doi.org/10.1109/ICSM.2012.6405253
7. Bloom, B.S., et al.: Taxonomy of Educational Objectives. Cognitive Domain, vol. 1, pp. 20–24. McKay, New York (1956)
8. Britto, R., Usman, M.: Bloom's taxonomy in software engineering education: a systematic mapping study. In: 2015 IEEE Frontiers in Education Conference (FIE), pp. 1–8. IEEE (2015). https://doi.org/10.1109/FIE.2015.7344084
9. Campbell, G., Papapetrou, P.P.: SonarQube in Action. Manning Publications Co. (2013). https://www.sonarqube.org/. Accessed 7 Aug 2019
10. CoderGears: JArchitect (2018). http://www.jarchitect.com/. Accessed 7 Aug 2019
11. Connolly, T.M., Boyle, E.A., MacArthur, E., Hainey, T., Boyle, J.M.: A systematic literature review of empirical evidence on computer games and serious games. Comput. Educ. **59**(2), 661–686 (2012). https://doi.org/10.1016/j.compedu.2012.03.004
12. Elezi, L., Sali, S., Demeyer, S., Murgia, A., Pérez, J.: A game of refactoring: studying the impact of gamification in software refactoring. In: Proceedings of the Scientific Workshops of XP 2016, p. 23. ACM (2016). https://doi.org/10.1145/2962695.2962718
13. Femmer, H., Fernández, D.M., Wagner, S., Eder, S.: Rapid quality assurance with requirements smells. J. Syst. Softw. **123**, 190–213 (2017). https://doi.org/10.1016/j.jss.2016.02.047
14. Fontana, F.A., Braione, P., Zanoni, M.: Automatic detection of bad smells in code: an experimental assessment. J. Object Technol. **11**(2), 5:1–5:38 (2012). https://doi.org/10.5381/jot.2012.11.2.a5
15. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (1999). http://martinfowler.com/books/refactoring.html. Accessed 7 Aug 2019
16. García, F., Pedreira, O., Piattini, M., Cerdeira-Pena, A., Penabad, M.: A framework for gamification in software engineering. J. Syst. Softw. **132**, 21–40 (2017). https://doi.org/10.1016/j.jss.2017.06.021
17. Guizzardi, G., Herre, H., Wagner, G.: On the general ontological foundations of conceptual modeling. In: Spaccapietra, S., March, S.T., Kambayashi, Y. (eds.) ER 2002. LNCS, vol. 2503, pp. 65–78. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45816-6_15
18. Haendler, T.: A card game for learning software-refactoring principles. In: Proceedings of the 3rd International Symposium on Gamification and Games for Learning (GamiLearn@CHIPLAY) (2019)
19. Haendler, T., Frysak, J.: Deconstructing the refactoring process from a problem-solving and decision-making perspective. In: Proceedings of the 13th International Conference on Software Technologies (ICSOFT), pp. 363–372. SciTePress (2018). https://doi.org/10.5220/0006915903970406
20. Haendler, T., Neumann, G.: A framework for the assessment and training of software refactoring competences. In: Proceedings of 11th International Conference on Knowledge Management and Information Systems (KMIS). SciTePress (2019)
21. Haendler, T., Neumann, G.: Ontology-based analysis of game designs for software refactoring. In: Proceedings of the 11th International Conference on Computer Supported Education (CSEDU), vol. 1, pp. 24–35. SciTePress (2019). https://doi.org/10.5220/0007878300240035

22. Haendler, T., Neumann, G.: Serious games for software refactoring. In: Proceedings of Software Engineering and Software Management, pp. 181–182. GI, Springer (2019). https://doi.org/10.18420/se2019-58
23. Haendler, T., Neumann, G.: Serious refactoring games. In: Proceedings of the 52nd Hawaii International Conference on System Sciences (HICSS), pp. 7691–7700 (2019). https://doi.org/10.24251/HICSS.2019.927
24. Haendler, T., Neumann, G., Smirnov, F.: An interactive tutoring system for training software refactoring. In: Proceedings of the 11th International Conference on Computer Supported Education (CSEDU), vol. 2, pp. 177–188. SciTePress (2019). https://doi.org/10.5220/0007801101770188
25. Haendler, T., Neumann, G., Smirnov, F.: RefacTutor: an interactive tutoring system for software refactoring. In: International Conference on Computers Supported Education, Revised Selected Papers of CSEDU 2019. Springer (2019)
26. Hamari, J., Koivisto, J., Sarsa, H.: Does gamification work?-A literature review of empirical studies on gamification. In: Proceedings of 47th Hawaii International Conference on System Sciences (HICSS), pp. 3025–3034. IEEE (2014). https://doi.org/10.1109/HICSS.2014.377
27. Hunicke, R., LeBlanc, M., Zubek, R.: MDA: a formal approach to game design and game research. In: Proceedings of the AAAI Workshop on Challenges in Game AI, vol. 4, pp. 1–5. AAAI Press, San Jose (2004)
28. Huotari, K., Hamari, J.: Defining gamification: a service marketing perspective. In: Proceedings of the 16th International Academic MindTrek Conference, pp. 17–22. ACM (2012). https://doi.org/10.1145/2393132.2393137
29. Kan, S.H.: Metrics and Models in Software Quality Engineering. Addison-Wesley Longman Publishing Co., Inc. (2002)
30. Khandelwal, S., Sripada, S.K., Reddy, Y.R.: Impact of gamification on code review process: an experimental study. In: Proceedings of the 10th Innovations in Software Engineering Conference, pp. 122–126. ACM (2017). https://doi.org/10.1145/3021460.3021474
31. Kitchenham, B.A., et al.: Towards an ontology of software maintenance. J. Softw. Maintenance Res. Pract. **11**(6), 365–389 (1999). https://doi.org/10.1002/(SICI)1096-908X(199911/12)11:6<365::AID-SMR200>3.0.CO;2-W
32. Krathwohl, D.R.: A revision of Bloom's taxonomy: an overview. Theor. Pract. **41**(4), 212–218 (2002). https://doi.org/10.1207/s15430421tip4104_2
33. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: from metaphor to theory and practice. IEEE Softw. **29**(6), 18–21 (2012). https://doi.org/10.1109/MS.2012.167
34. Landers, R.N.: Developing a theory of gamified learning: linking serious games and gamification of learning. Simul. Gaming **45**(6), 752–768 (2014). https://doi.org/10.1177/1046878114563660
35. Masapanta-Carrión, S., Velázquez-Iturbide, J.Á.: A systematic review of the use of Bloom's taxonomy in computer science education. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education, pp. 441–446. ACM (2018). https://doi.org/10.1145/3159450.3159491
36. Miljanovic, M.A., Bradbury, J.S.: A review of serious games for programming. In: Göbel, S., et al. (eds.) JCSG 2018. LNCS, vol. 11243, pp. 204–216. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02762-9_21
37. Misbhauddin, M., Alshayeb, M.: UML model refactoring: a systematic literature review. Empirical Softw. Eng. **20**(1), 206–251 (2015). https://doi.org/10.1007/s10664-013-9283-7

38. Moha, N., Gueheneuc, Y.G., Duchien, L., Le Meur, A.F.: DECOR: a method for the specification and detection of code and design smells. IEEE Trans. Software Eng. **36**(1), 20–36 (2010). https://doi.org/10.1109/TSE.2009.50
39. Morschheuser, B., Hassan, L., Werder, K., Hamari, J.: How to design gamification? A method for engineering gamified software. Inf. Softw. Technol. **95**, 219–237 (2018). https://doi.org/10.1016/j.infsof.2017.10.015
40. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Trans. Software Eng. **38**(1), 5–18 (2012). https://doi.org/10.1109/TSE.2011.41
41. Noy, N.F., McGuinness, D.L., et al.: Ontology development 101: a guide to creating your first ontology (2001). https://protege.stanford.edu/publications/ontology_development/ontology101.pdf. Accessed 7 Aug 2019
42. Object Management Group: Unified Modeling Language (UML), Superstructure, Version 2.5.1, June 2017. https://www.omg.org/spec/UML/2.5.1. Accessed 7 Aug 2019
43. Opdyke, W.F.: Refactoring Object-oriented Frameworks. University of Illinois at Urbana-Champaign, Champaign (1992). https://dl.acm.org/citation.cfm?id=169783
44. Paquette, G.: An ontology and a software framework for competency modeling and management. Educ. Technol. Soc. **10**(3), 1–21 (2007). https://www.jstor.org/jeductechsoci.10.3.1?seq=1
45. Parnas, D.L.: Software aging. In: Proceedings of 16th International Conference on Software Engineering, pp. 279–287. IEEE (1994), http://portal.acm.org/citation.cfm?id=257734.257788
46. Paulheim, H.: Knowledge graph refinement: a survey of approaches and evaluation methods. Semant. Web **8**(3), 489–508 (2017). https://doi.org/10.3233/SW-160218
47. Pedreira, O., García, F., Brisaboa, N., Piattini, M.: Gamification in software engineering-a systematic mapping. Inf. Softw. Technol. **57**, 157–168 (2015). https://doi.org/10.1016/j.infsof.2014.08.007
48. Prensky, M.: Digital game-based learning. Comput. Entertainment (CIE) **1**(1), 21–21 (2003). https://doi.org/10.1145/950566.950596
49. Raab, F.: CodeSmellExplorer: Tangible exploration of code smells and refactorings. In: 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 261–262. IEEE (2012). https://doi.org/10.1109/VLHCC.2012.6344544
50. Rolim, R., et al.: Learning syntactic program transformations from examples. In: Proceedings of the 39th International Conference on Software Engineering, pp. 404–415. IEEE Press (2017). https://doi.org/10.1109/ICSE.2017.44
51. Sandalski, M., Stoyanova-Doycheva, A., Popchev, I., Stoyanov, S.: Development of a refactoring learning environment. Cybern. Inf. Technol. (CIT) **11**(2) (2011)
52. Suryanarayana, G., Samarthyam, G., Sharma, T.: Refactoring for Software Design Smells: Managing Technical Debt. Morgan Kaufmann (2014). https://dl.acm.org/citation.cfm?id=2755629
53. Tempero, E., Gorschek, T., Angelis, L.: Barriers to refactoring. Commun. ACM **60**(10), 54–61 (2017). https://doi.org/10.1145/3131873
54. Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A.: JDeodorant: Identification and removal of type-checking bad smells. In: Proceedings of 12th European Conference on Software Maintenance and Reengineering (CSMR 2008), pp. 329–331. IEEE (2008). https://doi.org/10.1109/CSMR.2008.4493342
55. ZEN PROGRAM: NDepend (2018), http://www.ndepend.com/. Accessed 7 Aug 2019