Domain Modeling Based on Requirements Specification and Ontology

Iwona Dubielewicz, Bogumiła Hnatkowska, Zbigniew Huzar and Lech Tuzinkiewicz

Abstract Domain model plays an important role in software development. Typically, it is a primary input to elaboration of a system model which in turn is translated into source code and related database schemas. Effective development of domain model is a part of requirement engineering during which domain experts are employed to identify domain entities and relationships among them. We claim that this task can be supported by the use of domain ontologies from which interesting knowledge can be extracted. The starting point to knowledge extraction is an existing requirements specification. In this paper, we investigate how the form of requirements specification influences the quality of extracted model. Some measures allowing to assess the quality are introduced. A case study has shown that in the most cases the simplified version of a requirements specification is enough to obtain a satisfactory domain model, however if the domain is very complex, the extended version of requirements specification could be necessary.

Keywords Ontology · Requirements specification · Domain modeling · Knowledge extraction

I. Dubielewicz · B. Hnatkowska (✉) · Z. Huzar · L. Tuzinkiewicz
Faculty of Computer Science and Management, Wrocław University of Science and Technology, Wyb. Wyspiańskiego, 27, 50-370 Wrocław, Poland
e-mail: Bogumila.Hnatkowska@pwr.edu.pl

I. Dubielewicz
e-mail: Iwona.Dubielewicz@pwr.edu.pl

Z. Huzar
e-mail: Zbigniew.Huzar@pwr.edu.pl

L. Tuzinkiewicz
e-mail: Lech.Tuzinkiewicz@pwr.edu.pl

© Springer International Publishing Switzerland 2017

L. Madeyski et al. (eds.), *Software Engineering: Challenges and Solutions*,
Advances in Intelligent Systems and Computing 504,
DOI 10.1007/978-3-319-43606-7_3

1 Introduction

Incremental development is one of the approaches to software systems development. The software system is developed as a series of increments (builds, releases). Each new increment adds some functionality to the previous one. Specification, development, and validation are activities interleaved within single increment preparation. Incremental development seems to be the most common approach for software systems, and it is applied both in plan-driven (especially iterative) and agile methodologies. Its popularity owes to their advantages over classic waterfall model: reduced cost of implementation of changing requirements; feedback from customers during development work, and early delivery of running software.

Software development process is initiated by a real problem that occurs in a given application domain. The software system is intended to bring a solution or at least to support the problem solution. The software developer must first understand the problem, and, to this end, must have adequate domain knowledge. Usually, the needed knowledge is delivered by domain experts. In the paper, it is assumed that this knowledge can be extracted from a domain ontology. Recall that [12]: *An ontology is a formal* (*machine-readable*)*, explicit* (*consensual knowledge*) *specification* (*concepts, properties, relations, functions, constraints, axioms are explicitly defined*) *of a shared conceptualization* (*abstract model of some phenomenon in the world*). Further, we assume that it is available a consistent and complete domain ontology respective to the given problem.

The domain problem is usually informally outlined. The domain problem is described by requirements specification. The requirements specification is the base for system model design, and next for a code implementation. Symbolically, this process is depicted as follows (Fig. 1).

In the proposed approach, we demonstrate that the transition from the requirements specification to the domain model, which usually is performed by a business analyst in collaboration with domain experts, may be done semi-automatically with the use of a domain ontology. It means that the knowledge of the domain expert may be replaced by the knowledge contained in respective domain ontology provided that this ontology is consistent and complete. Of course, as in the traditional approach, the domain model should be validated by a domain expert. In the series of papers [3–5], we have presented the idea of this approach. Additionally, a set of programming tools, supporting the approach, have been developed [4, 7].

In this paper, we demonstrate how a form of requirements specifications may influence the resulting domain model. Our aim is to assess how the form and

$$\begin{bmatrix} requirements \\ specification \end{bmatrix} \xrightarrow{\substack{Domain \\ expert}} \begin{bmatrix} domain \\ model \end{bmatrix} \xrightarrow{\substack{Software \\ designer}} \begin{bmatrix} system \\ model \end{bmatrix} \xrightarrow{Programmer} \begin{bmatrix} source \\ code \end{bmatrix}$$

Fig. 1 Basic artifacts within development process

granularity of requirements specification influences the quality of the domain model obtained through knowledge extraction process from a given ontology. There are introduced some measures enabling assessment of this influence.

In Sect. 2 we present the forms of requirements specification assumed to be inputs to the knowledge extraction from existing ontologies. In Sect. 3 we shortly describe the extraction process as well as propose measures to access its results. Section 4 presents a case study, in which we try to answer the question if and how the form of requirements specification influences the quality of the resulting domain model. Section 4 gives the found conclusions and some comments on future works.

2 User Requirements Specifications

The notion of "requirement" has not unique meaning. According to IEEE [9], it is "*A condition or capability needed by a stakeholder to solve a problem or achieve objectives*". It can be found a variety of taxonomies for requirements, but the interesting one is this where requirements are defined for the product development and for the process which leads to this product development. As in the paper we concentrate on the product development process we consider taxonomy given in BABOK® [1]: business requirements, stakeholder (user) requirements, solution (or system) and transition requirements. Another requirements taxonomy is proposed in ISO/IEC 25030:2007 [10] but it is limited only to software system requirements.

In practice, the distinction between them remains problematic—while the business requirements are usually well discriminated from the others, distinguishing between user and system requirements is much more difficult.

Independently of software development approach, the requirements definition process is the of primary importance. In general, it consists of three sub-processes: requirement elicitation, analysis and specification (they all are generalized to the concept "requirement engineering") [2, 14]. Nowadays commonly used software development models are based on incremental and, more often adaptive approaches. Activities performed within one increment (release) form a kind of a mini-waterfall model. It means that the requirements for the release should be completely defined before the design of that release, and thus, the whole requirements elicitation process is performed iteratively.

Each new increment brings its releasable product (release). Within each increment, requirements are documented and analyzed. The specification of elicited requirements takes a form of plain text or sometimes more formal form—an use case model. They are expressed by a mixture of notions from the IT and business areas as in the iterative approach it is easy to cross the border between stakeholder and system requirements. The analysis aims at deciding whether the notions used in the requirements documentation are well-understood by all stakeholders and

whether these notions are consistent to the given application domain. In this analysis, a domain model elaborated on the base of the requirements specification and a respective domain ontology may be used.

A specific approach to requirement elicitation, analysis, and specification has been proposed in adaptive (agile) methodologies. The requirements specification takes a form called *User stories* or eventually *Epic* (just a large user story that needs to be broken down into) and represent a high-level description of a capability that the system needs to provide [2]. They are expressed only with the business notions and served as a basis for effort/cost estimation. After once picked up for being worked on, the user story is described with a lot of details given directly by the future user (and thus some IT notions can appear in its description). It only happens when they start to be subject to implementation. At that time also the acceptance criteria for the user stories are defined for they could be tested as "done" at the end of the iteration.

Another approach can be seen in the Use Case v.2.0 concept proposed by Jacobson [11]. For matching the incremental (precisely-agile) approach, he has suggested defining the use case scenarios progressively. Instead of specifying all scenarios at the beginning one can select (and define and implement) only some of them according to the order specified by the user ("a priority driven" approach). The scenarios specified in such a way has been called *use case slices*, and they correspond to the user stories defined with many details.

Further, we assume that the requirements specification is expressed in the form of user stories, usually presented in the context of a system vision [13]. User stories are presented as statements in a standardized "role-feature-reason" form. Exemplary variants of the forms are:

As a <type of user>, I want <some goal> so that <some reason>.
As a <role>, I want <goal/desire> so that <benefit>.

Description of each user story may be augmented with test cases. There are many schemas of test case description. We have chosen the one marked "*given-when-then*" which has the form:

GIVEN: Set up the object to be tested
WHEN: act on the object
THEN: Make claims about the object, its collaborators/parameters or global state.

Summarizing, we assume that requirements specification *rs* is defined as:

$$rs = \{ < h_1, tc_1 >, \ldots, < h_n, tc_n > \} \tag{1}$$

where $h_1, \ldots, h_n$ are user stories (histories), and $tc_1, \ldots, tc_n$ are sets of test cases. Both user stories and test cases may be presented using above notations.

3 Towards Domain Model

Practical approach to requirements specification begins with the initial list of user stories $h_1, \ldots, h_n, \ldots$. The stories are ordered according to their decreasing priorities. On the base of the arbitral decision of an specifier, first n stories with the highest priority are selected to form first user requirements specification. This specification usually contains only user stories, but for the sake of unique notation, it will be written as in (1), where the sets of test cases may be empty.

The requirements specification, as a set of documents in natural language, contains some words or phrases that represent notions related to a given problem within an application domain. In the presented approach [8], we concentrated on nouns and noun phrases as the most likely elements relating to a structural aspect of the considered problem. The set of these elements is called a set of initial domain notions.

The specification *rs* contains a set of initial domain notions *DSN*, which some of them are domain and other are systems' or native notions. *DNS* forms a glossary of terms within a considered domain. The expert should define a function $EQ: DNS \rightarrow ONT$, which maps glossary terms into their equivalent ontology concepts *ONT*. We assume that such domain ontology relevant to the considered problem is given. The ontology should be consistent; additionally, its completeness is strongly required.

The *EQ* function is usually a partial function because only the terms which represent domain notions may have equivalents in the set of ontology concepts; the domain of *EQ* (*dom*(*EQ*)) is marked in Fig. 2 by the gray oval within *DSN* while the codomain of *EQ* (*ran*(*EQ*)) is marked by the gray oval within ONT.

A well-defined domain ontology should rather not contain specific notions related to information systems.

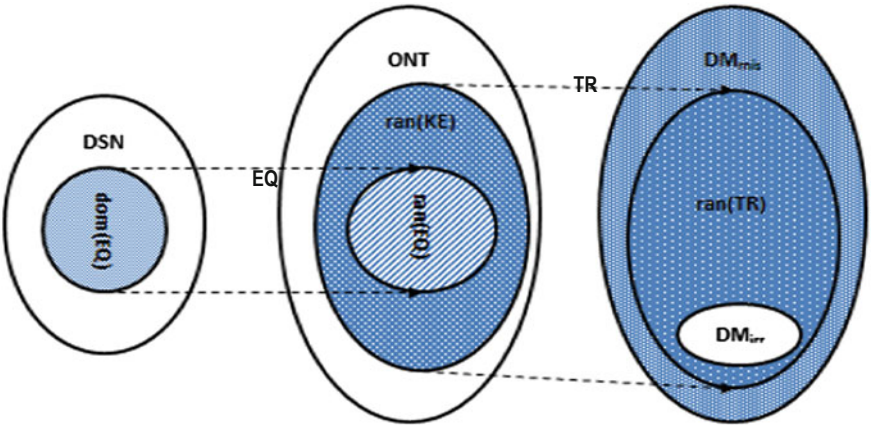

Fig. 2 Schema of transformations from a requirements specification to a domain model

The value:

$$coverage = \frac{card(dom(EQ))}{card(DNS)} \tag{2}$$

where $card(X)$ means cardinality of the set X, is the measure how terms of the glossary are covered by ontology notions.

The set of ontology notions defined by the EQ function, i.e. the set representing its range $ran(EQ)$, is the base for extracting additional knowledge from the ontology ONT. The additional notions are extracted by knowledge extraction algorithm [8], which abstractly is represented as a function $KE(ran(EQ), ont) \subseteq ONT$. The set of ontology notions extracted from the ontology is represented in Fig. 2 as a range of the function KE, i.e. $ran(KE)$. Informally, $ran(KE)$ may be called a sub-ontology of the ontology ONT. Of course, this set contains $ran(EQ)$ as its subset.

The value:

$$enrichment = \frac{card(ran(KE) \backslash ran(EQ))}{card(ran(KE))} \tag{3}$$

may be treated as a measure how the ontology enriches the initial mapping of glossary terms.

The set $ran(KE)$, represented by the dark gray oval in Fig. 2, is a base for automatic UML class diagram generation [3]. The algorithm generating a class diagram is abstractly represented as a function $TR(ran(KE))$. The generated class diagram $ran(TR)$ is represented in Fig. 2 by dark gray oval, a subset of DM. DM represents a final domain model, also represented in the form of UML class diagram, which is obtained by modification of $ran(TR)$. The modification is the result of two activities performed by an analyst/domain experts. Within these activities, it is decided which elements of the $ran(TR)$ on the class diagram are irrelevant, and which elements are missing with respect to the requirements specification.

An element $e \in ran(TR)$ is said to be irrelevant to the requirements specification if e is not used in the description of any user story or any test case. The set of irrelevant and missing elements are denoted by DM_{irr} and DM_{mis}, respectively. The following are measures of irrelevance:

$$irrelevance = \frac{card(DM_{irr})}{card(ran(TR))} \tag{4}$$

and missing:

$$missing = \frac{card(DM_{mis})}{card(ran(TR) \backslash DM_{irr}) + card(DM_{mis})} \tag{5}$$

In the incremental and iterative software life-cycle model, the requirements specification prepared in one iteration may be modified by extension or refinement in the next iteration. Requirements specification rs_1 may be developed in subsequent iterations. The specification rs_2 is an extension of rs_1 if $rs_2 = rs_1 \cup < h_{n+1}, tc_{n+1} > \ldots\}$. It means that new user stories are added to the previous specification rs_1. The specification rs_2 is a refinement of rs_1 if some test case tc_i assigned to the user story h_i ($i = 1, \ldots, n$) from rs_1 is replaced by $tc_i^{'}$ such that $tc_i \subset tc_i^{'}$. In practice, one modification step may contain both requirements extension and its refinement.

Summarizing, after a series of iterations, we have a sequence of requirements specifications: $rs_1, rs_2, \ldots, rs_n$ accompanying with a sequence of respective sub-ontologies: $ont_1, ont_2, \ldots, ont_n$, and a sequence of domain models: $dm_1, dm_2, \ldots, dm_n$.

The way how requirements specifications are constructed shows that $DNS_i \subseteq DNS_{i+1}$. Therefore also $ONT_i \subseteq ONT_{i+1}$, where ONT_i is the set of ontology notions for the sub-ontology ont_i, and $DM_i \subseteq DM_{i+1}$, where DM_i is the set of domain model notions for the domain model dm_i.

Now the question begs: how, in the context of the ontology, the form of rs influences the quality of dm. It is difficult to answer the question directly, but we may examine how the increment of the set of initial domain notions influences the set of ontology and domain model notions. For example, if for some $i = 1, 2, \ldots, n$ the condition holds:

$$\frac{card(DNS_{i+1})}{card(DNS_i)} > \frac{card(ONT_{i+1})}{card(ONT_i)} > \frac{card(DM_{i+1})}{card(DM_i)}$$

it may be interpreted that each subsequent requirements specification has a weaker influence on the subsequent domain model.

It should be noted that from the domain expert perspective, it would be not convenient to prepare a domain model in incremental approach by creating a sequences of sub-ontologies. It is rather suggested that the sub-ontology should be created only ones on the base of the most developed requirements specification. Of course, the postulate of incremental system model and software release development remains still valid.

4 Case Study

The aim of the case study is to answer the question if and how the form of requirements specification influences the quality of the domain model obtained through knowledge extraction process from existing ontologies. We would like to know if a domain model of acceptable quality can be retrieved from existing ontologies at early stages of software development, when requirements

specification doesn't contain many details, or if we should refine as many requirements as possible to obtain better results.

The study starts with a simplified version of requirements specification which consists of two user stories with accompanying test cases (see Sect. 4.1). The user stories refer to hotel reservation domain where a potential customer wants to browse a hotel offer and later, to check the availability of specific room type in a given period. Next, we follow the same extraction procedure twice, for two forms of requirements specification—the first contains only user stories (see Sect. 4.2), while the second also test-cases (see Sect. 4.3). The domain ontologies from which the knowledge is extracted are included into the following SUMO [15] files: Merge.kif, Mid-level-ontology.kif, Hotel.kif, Dining.kif, Catalog.kif. The first two files are upper ontologies while the last two are domain ontologies referenced by Hotel.kif. The models obtained in both cases are modified: missing elements (according to the test cases) are added, and irrelevant found out and marked. The proposed metrics are calculated what allows us to derive some conclusions.

4.1 Case Study—Requirements Specification

This subsection contains the definition of two user stories accompanied by test cases. Test cases are written in Gherkin language—"a business readable, domain specific language that lets you describe software's behavior without detailing how that behavior is implemented" [6].

User Story 1: As a *potential customer*, I want to see information about *hotel,*[1] *hotel rooms*, *rooms' amenities* and *prices* so that I can decide whether to become a *customer*.

Test cases for User Story 1:
Scenario 1: Basic information about hotel
Given that a hotel is defined with: <*name*>, <*postal address*>, and <*category*>
When a customer navigates to the main hotel page
Then he/she should be informed about hotel <name>, <postal address>, and <category>.

Examples (basic information about hotel):

Name	Postal address	Category
Hostel	Bema Street 5, Wroclaw, Poland	***

[1]Domain notions are written in italics.

Scenario 2: List of hotel rooms
Given that a hotel rents rooms of types defined with: <*room type*>, <*capacity*>, <*price per night*>, <*amenities*>
When a customer navigates from main hotel page to the list of room types page
Than he/she should be informed about the hotel room types.

Examples (types of hotel rooms):

room type	capacity	price per night	amenities
single	1	230 zł	TV, sejf
double	2	300 zł	balcony, TV, sejf

User Story 2: As a potential customer, I want to check availability of selected *room type* (*room availability*) in a given *reservation period* so that to be able to decide if to make *reservation* or not.

Test cases for User Story 2:
Scenario 1: No room of a given room type is available in selected period
Given that a hotel has reservations for rooms of <*room type*>
And that <*room type*> is reserved from <*dateFrom*> to <*dateTo*>
And that the hotel has <*nr instances*> of rooms of <*room type*>
When a customer asks for the <*room type*> availability from <*given dateFrom*> to <*given dateTo*>
Than he/she is informed that no room of specific <*room type*> is available from <*given dateFrom*> to <*given dateTo*>.

Examples:

Input:
room type: single given dateFrom: 1-1-2016 given dateTo: 2-1-2016

		reservation period	
room type	no instances	dateFrom	dateTo
single	2	1-1-2016	3-1-2016
		31-12-2015	2-1-2016

Scenario 2: There is a room of a given room type available in selected period
Given that a hotel has reservations for <*room type*>
And that <*room type*> is reserved from <*dateFrom*> to <*dateTo*>
And that the hotel has <*nr instances*> of rooms of <*room type*>
When a customer asks for <*room type*> availability from <*given dateFrom*> to <*given dateTo*>
Then he/she is informed that a room of <*room type*> is available from <*given dateFrom*> to <*given dateTo*>.

Examples:

Input:
room type: single given dateFrom: 1-1-2016 given dateTo: 2-1-2016

		reservation period	
room type	no instances	dateFrom	dateTo
single	2	1-1-2016	3-1-2016
		31-12-2015	1-1-2016

4.2 Domain Model Built Basing on the User Stories

This subsection describes the inputs/outputs of the functions involved in the extraction process. We start from the definition of *DNS* set:

DNS = {*hotel*, *hotel room, room amenity, price per night, room availability, reservation, reservation period, customer, potential customer*}.

The notions that, according to a domain expert, describe business notions at that moment, are mapped by him to SUMO ontology. List of mappings from *DSN* to *ONT* (*EQ* function) is as follows:

{*hotel* → *HotelBuilding*, *hotel room* → *HotelRoom, room amenity* → *roomAmenity, price per night* → *price, room availability* → *reserved-Room, reservation* → *HotelReservation, reservation period* → *(reservationStart, reservationEnd)*}.

It can be noticed that two of *DNS* notions are treated as the system notions and are not mapped to the ontology notions.

Further the ontology notions are used to extract additional knowledge from ontology (with *EQ* function) and to translate extracted notions to a UML class diagram (with *TR* function). The translation maps each ontology element into an appropriate UML element (the mapping is one to one). Figure 3 presents the results of automatic UML class diagram generation. Please note, that *HotelUnitType*, and *PhysicalType* are UML *PowerTypes*.

After generation, the domain model is investigated by a system analyst to find out missing and irrelevant elements. Missing elements are presented in Fig. 4 in bold (relations) or are written with Courier font (attributes, classes), while irrelevant elements are drawn by dashed line (classes) or by thinner lines (relations). The generated domain model doesn't contain many details that are necessary to realize all test cases, e.g. we lack the definition of properties like hotel name or category as well as the definition of some relationships, e.g. hotel building consists of hotel units. There is only one irrelevant element—we don't need to know who has defined a price for a hotel unit.

The values of basic measures for that scenario are listed in Table 1.

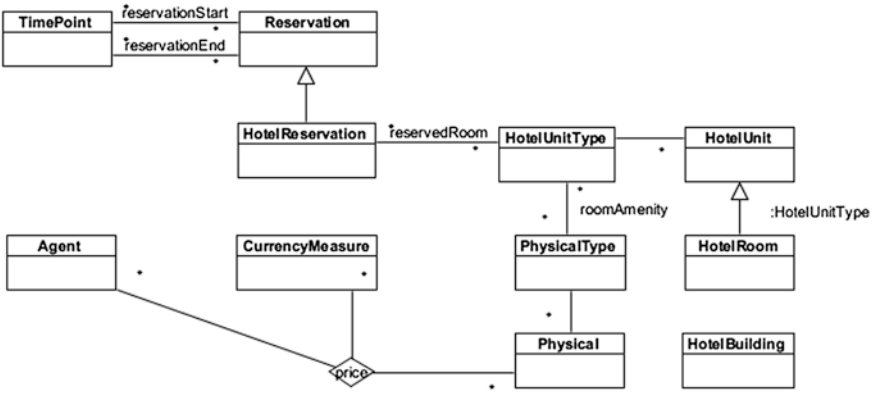

Fig. 3 Automatically obtained domain model (according to Scenario 1)

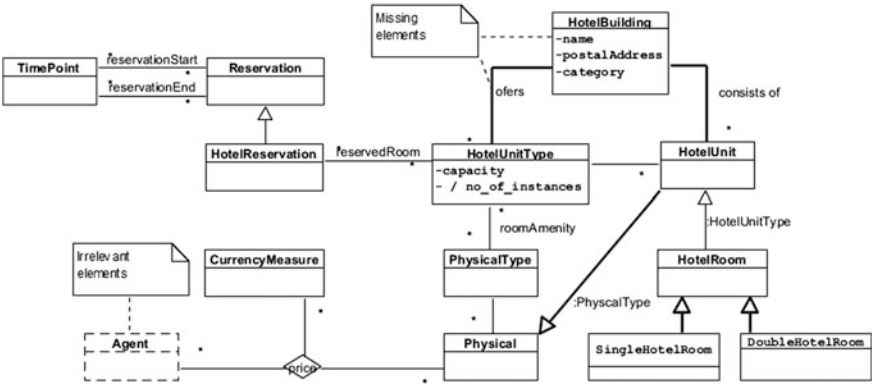

Fig. 4 Revised version of domain model (according to scenario 1)

Table 1 Basic measures counted for scenario 1 (columns refer to cardinality of defined sets)

DNS	dom (EQ)	ran (EQ)	ran (KE)	ran (TR)	DM_{irr}	DM_{mis}
9	7	8	20	20[a]	1	13[b]

[a]11 classes, 7 relations, 2 inheritance relationships
[b]2 classes, 5 attributes (1 unnecessary – \no of instances), 2 relations, 3 inheritance rel

4.3 Domain Model Built Basing on the User Stories and Test Cases

Similarly, to the previous section we start with the definition of *DNS* set. In comparison to the previous case, it contains additional notions, e.g. name, and address.

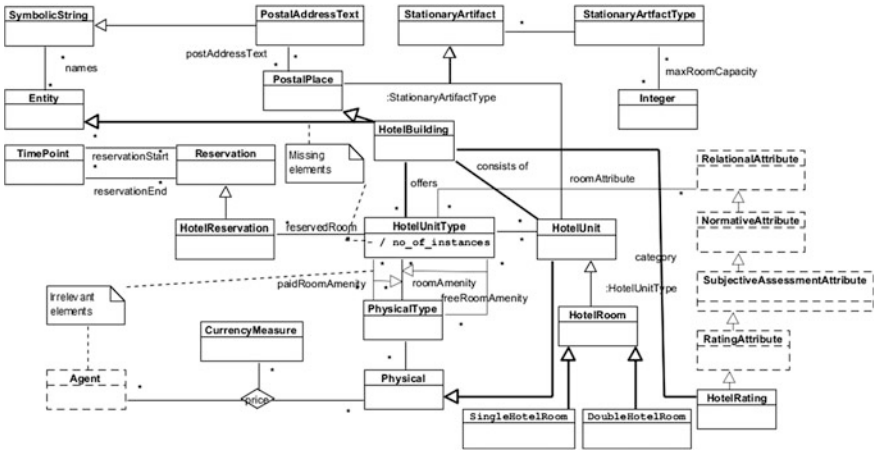

Fig. 5 Revised version of domain model (according to Scenario 2)

Table 2 Basic measures counted for scenario 2 (columns refer to cardinality of defined sets)

DNS	dom (EQ)	ran (EQ)	ran (KE)	ran (TR)	DM_{irr}	DM_{mis}
13	11	12	47	47[a]	10[b]	9[c]

[a]Classes, 14 relations, 10 inheritance (2 between relations)
[b]5 classes, 3 relations, 2 inheritance relations
[c]2 classes, 1 attribute (unnecessary: "\no of instances"), 1 relation, 5 inheritance relationships (most are defined in ontology)

DNS = {*hotel, hotel room, room amenity, price per night, room availability, reservation, reservation period, name, address, category, capacity*}.

The list of mappings from *DSN* to *ONT* (*EQ* function) is as follow:

{*hotel* → *HotelBuilding, hotel room* → *HotelRoom, room amenity* → *roomAmenity, price per night* → *price, room availability* → *reservedRoom, reservation* → *HotelReservation, reservation period* → *(reservationStart, reservationEnd), name* → *names, address* → *postAddressText, category* → *HotelRating, capacity* → *maxRoomCapacity*}.

Figure 5 presents a refined version of UML class diagram—both missing and irrelevant elements are marked here in the same way as in Fig. 4.

The values of basic measures for that scenario are listed in Table 2.

4.4 Comparison

In this section, we present the values of derived measures calculated for both scenarios of our case study—see Table 3.

Table 3 Derived measures calculated for both scenarios

Measure	Scenario 1: user stories	Scenario 2: user stories + test cases
Coverage	0.77	0.84
Enrichment	0.60	0.74
Irrelevance	0.05	0.21
Missing	0.41	0.19

When our *DNS* set contains more elements (scenario 2) more of them are considered to be domain notions (84 %) in comparison to the scenario 1 (77 %). It seems that this observation should be valid also in the context of incremental development, where, in an increment, a new user-stories are added or refined. Scenario 2 also allows extracting more elements from existing ontology (enrichment is equal to 74 % in comparison to 60 %). The extraction process is not ideal as it produces some irrelevant elements. The likelihood that irrelevant elements will appear is higher for scenario 2 (irrelevance is equal to 21 % in comparison to 5 %). On the other hand, the number of missing elements decreases when the input is richer (missing is equal to 19 % in comparison to 41 %).

The case study allows us to derive some preliminary conclusions. For the simple domains it should be enough to start with user stories only. A domain model resulting from the extraction procedure should be not complicated (only a few irrelevant elements), and relatively easy for extension by a domain expert. The missing elements can be added iteratively, on demand. For a complex domain, it is worth to invest with a more detailed requirements specification, which allows to obtain a domain model with less number of missing elements but possibly some out of scope. It is easier to cross out unnecessary parts and have an opportunity to familiarize deeper with the domain (from the perspective of a business analyst) than to ask domain experts for identification of all lacking elements.

5 Conclusions

The aim of the paper is to present an approach to the knowledge extraction from an ontology to create appropriate domain model and to evaluate the influence of the requirements specification on the quality of the resulting model. The main problems analyzed and discussed in this work are related to the evaluation of the possibility and potential benefits of using ontology in the process of creating domain models in the context of defined requirements (given in the form of the user stories). Defined measures: *enrichment*, *irrelevance*, and *missing*, allow to evaluate the quality of the extracted domain model.

Examples of knowledge extraction described in the case study show that the created domain model can contain redundant elements (*irrelevance* > 0). On the other side, one cannot assure that the resulting model fully meets user expectations (*missing* > 0). Everything depends on the quality of requirements specification as

well as the completeness of the selected ontology. In most cases the simplified version of requirements specification is enough to obtain a satisfied domain model, however, if the domain is very complex, an extended version of requirements specification could be necessary.

Expected benefits of the proposed approach is to reduce the commitment and costs of participation of experts in the process of domain knowledge extraction. It is possible due to a partial automation of domain modeling process using ontology, and the permanent access to the knowledge domain by IT developers. Experts responsibilities may be limited to:

- mapping of *DNS* notions to ontology concepts,
- validation and possibly extension of created domain model.

The other benefit is a better quality of resulting domain model, which—by construction—is consistent with ontology from which it was created.

Future works are concentrated on experiments aiming at improving effectiveness and precision of the knowledge extraction algorithm from ontology which is used in the process of generating domain models. Moreover, the results of experiments with SUMO ontology challenged our assumption about the completeness of this ontology and therefore it is also planned utilization of OWL ontologies in further research.

References

1. A Guide to the Business Analysis Body of Knowledge® (BABOK® Guide) v2
2. Cobb, Ch.G.: Making Sense of Agile Project Management: Balancing Control and Agility. Wiley (2011)
3. Dubielewicz, I., Hnatkowska, B., Huzar, Z., Tuzinkiewicz, L.: Domain modeling in the context of ontology. Found. Comput. Decis. Sci. **40**(1), 3–15 (2015)
4. Dubielewicz, I., Hnatkowska, B., Huzar, Z., Tuzinkiewicz, L.: Development of domain model based on SUMO ontology. In: Zamojski, W., et al. (eds.) Proceedings of the 10th International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, pp. 163–173. Springer (2015)
5. Dubielewicz I., Hnatkowska B., Huzar Z., Tuzinkiewicz L.: Problems of SUMO-like ontology usage in domain modelling. In: Nguyen, N.T., et al. (eds.) 6th Asian Conference, ACIIDS 2014, Lecture Notes in Computer Science, vol. 8397, pp 352–363, Springer (2014)
6. Gherkin: http://docs.behat.org/en/v2.5/guides/1.gherkin.html
7. Hnatkowska, B.: Towards automatic SUMO to UML translation. In: Kościuczenko, P., Śmiałek, M. (eds.) From Requirements to Software, Research and Practice, pp. 87–100. Polskie Towarzystwo Informatyczne (2015)
8. Hnatkowska, B., Dubielewicz, I., Huzar, Z., Tuzinkiewicz, L.: Conceptual modeling using knowledge of domain ontology. In: Nguyen, N.T., et al. (eds.) 8th Asian Conference, Intelligent Information and Database Systems, Proceedings, Part II, pp. 558–566. Springer (2016)
9. ISO/IEC/IEEE 29148-2011—Systems and software engineering—Life cycle processes—Requirements engineering. 2011
10. ISO/IEC 25030:2007 Software engineering—Software product Quality Requirements and Evaluation (SQuaRE)—Quality requirements

11. Jacobson, I.: Use Case 2.0. A Guide to Succeeding with Use Cases. https://www.ivarjacobson.com/sites/default/files/field_iji_file/article/use-case_2_0_jan11.pdf. Accessed 10 Apr 2016
12. Studer, R., Benjamins, V., Fensel, D.: Knowledge engineering: principles and methods. Data Knowl. Eng. **25**, 161–197 (1998)
13. Wikipedia: Vision document—wikipedia, the free encyclopedia, 2015. Accessed 4 Apr 2016
14. Wikipedia: Requirements engineering—wikipedia, the free encyclopedia, 2016. https://en.wikipedia.org/wiki/Requirements_engineering. Accessed 4 Apr 2016
15. Wikipedia: Suggested upper merged ontology—wikipedia, the free encyclopedia, 2016. Accessed 10 Apr 2016