DE GRUYTER OPEN

BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **17**, No 2
Sofia • 2017 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.1515/cait-2017-0025

An Environment for Automatic Test Generation

Nina Stancheva[1], *Asya Stoyanova-Doycheva*[1], *Stanimir Stoyanov*[1], *Ivan Popchev*[2], *Vanya Ivanova*[1]

[1]*Faculty of Mathematic and Informatics, University of Plovdiv "Paisii Hilendarski", Plovdiv, Bulgaria*
[2]*Faculty of Economics and Social Sciences, University of Plovdiv "Paisii Hilendarski", Plovdiv, Bulgaria*
E-mails: st.nina@abv.bg astoyanova@uni-plovdiv.net stani@uni-plovdiv.net, ipopchev@iit.bas.bg vantod@abv.bg

***Abstract*:** *This paper presents an environment which generates tests automatically. It is designed for assistance in the software engineering education and is part of the Virtual Education Space. The environment has two functionalities – generation and assessment of different types of test questions. In the paper, the architecture of the environment is described in detail. The test generation is supported by specialized ontologies, which are served by two intelligent agents known as Questioner Operative and Assessment Operative.*

***Keywords*:** *Virtual education space, e-Learning, automatic question generation, ontologies, intelligent agents.*

1. Introduction

The Distributed e-Learning Centre (DeLC) is a project implemented in the Faculty of Mathematics and Informatics at the University of Plovdiv and aiming at the development of an infrastructure for delivery of electronic education services and teaching content [1-3]. The environment is constantly expanding through the integration of new components such as, for example, intelligent agent supporting refactoring learning [4], middleware providing mobile e-Learning services [5], the electronic content editor Selbo2 [6]. Furthermore, a module for optimal deployment and access of information resources is implemented in the environment [7]. Having in mind the disadvantages of DeLC, the environment is being changed into a new infrastructure known as Virtual Educational Space (VES) [8, 9]. The VES is adapted for use in the secondary school [10]. In the latest years virtual testing has taken a significant place as a form of examining in the universities. In DeLC eTesting is one of the most widely used services provided by the environment. As a consequence, the virtual testing system in VES has been improving over the QTI 2.1 standard [11, 12].

To improve eTesting and the teachers' efficiency a new system is needed to automatize the process of creating questions and preparing tests. Since this process

takes a lot of time, efforts and attention, an automatic or semi-automatic method for generating tests is desirable. To accomplish this, a formal model and a supporting environment that generate tests are being developed and integrated in the virtual space. VES is an intelligent, context-aware, scenario-oriented and controlled infrastructure, which has various assistants and operatives that are maintaining it and executing its functionalities. The operative assistants provide various e-Learning services to the space users. These assistants support the execution of different scenarios and provide expert functionalities. The electronic resources are the main source for an e-Learning educational system. Thus, it is important to organize them in an appropriate structure – to make them easily accessible for the participants in the educational process or easily processed by the operating entities. Furthermore, VES also integrates different types of electronic resources, ontologies, Sharable Content Objects (SCO elements), e-packages in SCORM 2004 (Sharable Content Object Reference Model), data base with test questions, and statistics for the students. These resources are accessible by specialized operatives in VES.

This paper presents the environment for automatic test generation implemented as part of VES. The rest of the paper is structured as follows: Section 2 starts with a short literature overview, Section 3 introduces the environment in detail, and finally Section 4 concludes the paper.

2. Related works

Automated question generation as a component of intelligent test systems could be developed in different ways. In [13], an approach to question generation from structured data is presented. The supporting tool reads input data presented as tables and generates questions of various types. The question generation is completed in the following steps: pre-processing, entity recognizing, tuple generation, and question generation. The paper [14] discusses in detail the use of ontologies as a means of multiple choice question generation. Two new generation approaches are proposed there. An integer linear programming approach for static test generation is introduced in [15]. By help of this approach, test papers could be generated in a huge search space. In addition, a novel framework for web-based testing with automatic assessment is also discussed in this paper. The framework is tested in a mathematics question data base. In [16] is demonstrated a framework for automatic generation of multiple-choice questions in Chinese. In the framework, the generation process is decomposed into several sub-processes such as sentence identification, query term identification, and distractors generation. An eTesting system in two variations is described in [17] with standard graphical interface and with interactive images. In the paper, several question generation strategies using multiple-choice questions and for interactive images are presented as well. The strategies can be applied to develop a large set of questions. In [18], a vocabulary-learning scenario in a Basque-language is used to support teachers by generation of multiple choice questions. In this environment, multiple-choice questions are generated semi-automatically applying natural language processing techniques. In [19], an approach for semi-automatic question generation is described. The generator first extracts key phrases from

students' literature review papers. The extracted phrases are matched with a selected Wikipedia article and classified into various categories. In the next step, the generator constructs a conceptual graph structure representation for each phrase. Finally, the questions are generated using a conceptual graph.

3. Test environment

To facilitate the development of the system, a model has been developed that delivers a formal representation of the main concepts and constructs. The model aims to provide a common way of generating test questions, including the representation of the needed knowledge and the question forming. The following three levels described in detail in [20] constitute the model:

- *Domain level.* At this level, the basic building blocks are modelled as a repository consisting of interrelated teaching units.
- *Extractor level.* At this level, separate building blocks can be selected according the desired test structure. The supporting operators are known as extractors.
- *Generator level.* At third level, the generation of test questions using the already extracted building blocks is modelled.

In accordance with the model a prototype of test generation environment was developed and adapted for the domain of UML (Unified Modelling Language) which is detailed presented in this section.

3.1. Architecture

The environment consists of two operatives (intelligent agents) known as Questioner Operative (QO) and Assessment Operative (AO) respectively (Fig. 1). Both operatives share a knowledge base, a data base, and a Graphical User Interface (GUI). A database manager supports both operatives to save data about the tests. The operatives are implemented as intelligent agents which process structured educational content saved in the environment's knowledge base to achieve their tasks. Each of them has its own specific tasks, but they have a similar architecture. Both have separate sensors and effectors which are used to accomplish an interaction with their environment. In addition, the QO and AO access the GUI through the sensors and effectors. The Local Control manage and coordinate the work of all agent's components.

The Questioner Operative generates the test in general. To compose the test, it forms random questions on a given topic by its behaviours using the knowledge base.

The second operative in the environment is the Assessment Operative. Its role is to check the users' answers and to keep records of the results during the test. This operative has several behaviours that process, analyse and assess user's answer. It uses the UML ontology to confirm answers.

Although the Questioner Operative and the Assessment Operative have their own specific tasks, they need to communicate during the test to perform and coordinate their actions. In this way they can require some performance from the

other one or send information. This communication is implemented under the FIPA (Foundation for Intelligent Physical Agents) specifications.

The both operatives are developed in Java and JADE (Java Agent Development Framework) integrated in the Eclipse environment. JADE is powerful and convenient due to standardized middleware support of multi-agent systems complying with the FIPA specifications.

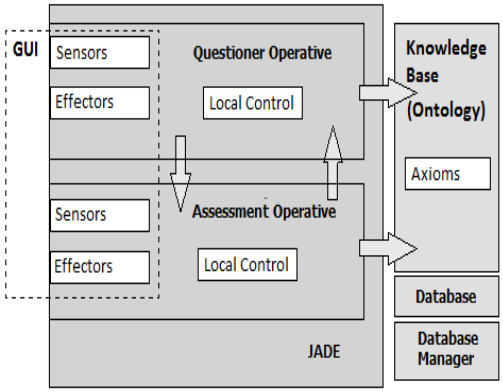

Fig. 1. Architecture of the test environment

The knowledge base stores structured educational content implemented as an ontology. The ontology represents the basic concepts, the relations between them and the significant rules building the UML language. The ontology approach is chosen because it's a better way to present the information, so that it can be processed by semantic means. Protégé-OWL is the editor that is used for creating the ontology. The OWL format, approved by W3C (Word Wide Web Consortium), is appropriate for the purpose. Another advantage is OWL API – a Java library for processing ontologies which is used in the realization of the operatives, so they can access and process the ontology. The prototype database is implemented using H2 Database Engine.

3.2. Knowledge base

Currently, the environment is adapted for use in the software technology discipline and in particular for the UML language domain [21]. The knowledge base stores and provides structured educational content for the basic UML concepts, elements and diagrams. The ontology is constructed from classes and properties. Classes show the elements in UML while properties present the relations between the elements. The semantic meaning of the classes is shown by their restrictions.

The ontology classes are organized in a hierarchy. In this way the type of a given UML element can easily be determined by finding its superclass. There are six classes, which are the main ones on the top level of the hierarchy (Fig. 2).

The created properties in the ontology, and the existential and universal constraints establish the relations between classes. Thus, we can say how one UML element could be connected to another one or what its function is. It is important that each property has its domain and range.

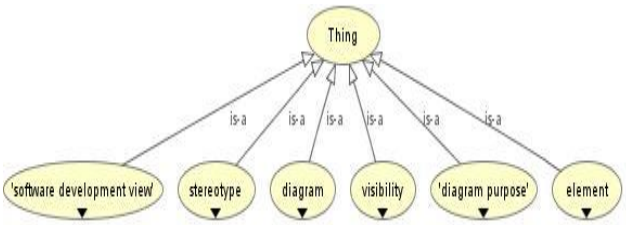

Fig. 2. Main classes in the UML ontology

The classes, their hierarchy relations, properties and restrictions form the axioms of the ontology. These axioms are rules, defining the UML domain. They are the key for semantic processing of the knowledge by the operatives. Each axiom contains a base class (the one that is described by the axiom) and a restriction. The restriction is formed by one or more properties and defining classes. The properties connect the main class with the restriction classes, showing some semantic meaning.

The meta-knowledge takes a significant part in the ontology. All classes and object properties have label annotations. They are necessary for the processing of the knowledge in an appearance, readable for the user. There are two user-defined annotations (Fig. 3) – declarative and interrogative. They are used to amplify the properties and their effective processing. The values of these two annotations give the correct words to use in the different sentence types in grammatical means. There is one more user-defined annotation, called test. It is used to annotate the axioms. Its purpose is to show in what test the knowledge, presented by the axiom, can be included. The value of this annotation is the topic of the test. This helps to generate thematic questions over the UML topic and thus to create different tests types.

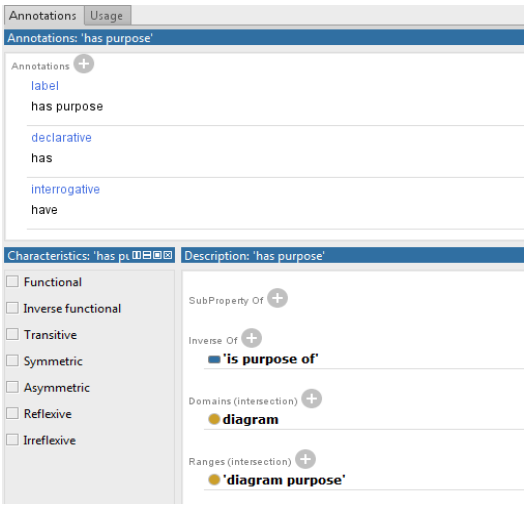

Fig. 3. Property annotations (Protégé-OWL view)

Our ontology contains over 850 axioms therefore it offers a variety of questions that can be asked by the QO.

3.3. Database

The database is developed as a relational database. It is needed to store data about the tests taken by the user such as test questions and results from the tests. Also, it offers the opportunity to realize the functionality that can provide a sequence of thematic tests, containing a different number of questions (Fig. 4).

The table STUDENTS is intended for data about the users. As it is assumed that the users are students who take exams or prepare for exams, this table stores the student's faculty number. The next table TESTS stores the data for the different tests which exist for the environment and that the students have to take. The NAME column stores the name of the test, which is its topic. The idea here is that some of the test annotations from the ontology will match this value. In this manner the knowledge presented by the different axioms can be assigned with the different test types. So a row from this table shows the test type, and the knowledge needed to generate this test can easily be found in the ontology. The QUESTIONS_COUNT column stores the number of the questions that have to be generated and asked in this test. SEQUENCE is a column that shows the priority of this test. The value of this integer column tells the sequence of giving the tests to a user. It defined which test has to be taken first, second and so on. The STUDENTS_TESTS_MAP stores data about the tests that a user has already taken. It maps the STUDENTS and the TESTS tables, and it also keeps data about the date of taking the test and its result. The last table is the STUDENTS_TESTS_QUESTIONS_MAP. It is needed for the data about the questions that were not answered correctly by the user. It has a foreign key, linking the rows to the STUDENTS_TESTS_MAP table. Hence, the wrongly answered question is related to the test and to the user who was taking it. This table stores the full question string and the class from the ontology that was used for generating the question. The ontology class shows the UML element that the question was about. Storing it gives the opportunity for the same or another question about this element to be asked later in another test. The idea is to have a final test after the thematic tests, where the asked questions refer to elements/subjects that were not learnt well before.

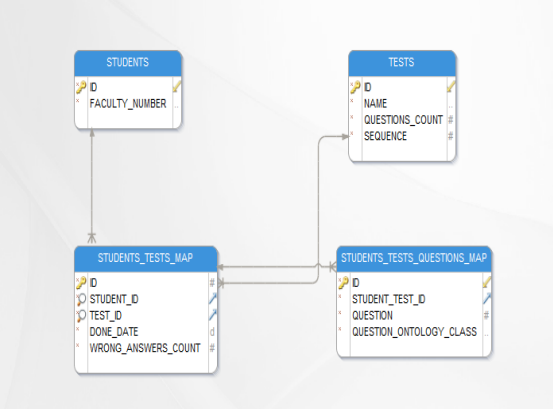

Fig. 4. Database tables

For our prototype purposes 11 rows were added in the TESTS table (Fig. 5) for the different tests types. The topics of the first 9 tests are the different UML packages. There is a specific test for the UML diagrams. The final test unites all previous topics and is supposed to ask questions about the elements that were problematic before. A manager class was created to interact with the database.

SELECT * FROM TESTS;

ID	NAME	QUESTIONS_COUNT	SEQUENCE
11	use case package	10	1
12	interaction package	10	2
15	class package	10	3
14	state machine package	10	5
13	activity package	10	4
17	composite structure package	10	6
16	component package	10	7
18	deployment package	10	8
19	relationship package	10	9
22	final package	20	11
21	diagram package	10	10

(11 rows, 8 ms)

Fig. 5. TESTS table rows in the prototype

3.4. Question generation

In this section, we discuss the automatic generation of different types of test questions by using an ontology. We use the QTI 2.1 Standard for defining different question types. The QTI 2.1 Standard specifies item types, which can be included in a test. These item types specify different ways to form a question. Research is done to include different question types in the environment that are suitable for generation by using an ontology. In the generation features different kinds of questions are incorporated and they correspond to some of the item types, pointed out in this standard.

The question generation is part of the Questioner Operative's capabilities. The QGFM is used to realize the question generation algorithm. Initially, we must say that each question is formed by one or more entities so the generation of each question starts with the generation of the sentences which it will contain. We define two types of sentences to generate – declarative and interrogative.

For both types we start by extracting a set of axioms. The axioms from this set are axioms from the ontology corresponding to criteria, which depend on the question type requirements and on the test type requirements (they will be discussed later). Only axioms from the Equivalent Classes or Sub Class types are used since they are the most suitable for sentence generation due to the similar order of the elements. One sentence is generated from one axiom from this set. Then, the different elements of the axiom are extracted – a base class and restrictions, and one of the restrictions is used for the generation. Currently, we support only existential and universal restrictions. The property and the defining classes are extracted from the restriction, if they are present. The restriction classes can be combined in different sets, but for the generation we use a restriction containing a union or intersection set types.

When all elements are extracted from the axiom, the algorithm separates in two branches. If the property is empty (the axiom doesn't contain any properties), then

the axiom presents hierarchical relations. Semantically, the UML element, represented by the base class, is a kind of the UML element/s, represented by defining classes. We show this relation in the generated sentence by the verb "is", so the empty property is filled with the "is the" string. If this is not the case and the axiom contains a property we need to extract the right annotation value that will present the property in the sentence. The annotation corresponds to the sentence type. For declarative sentences we use the declarative annotation and for interrogative sentences – the interrogative annotation.

In case of an interrogative sentence we extract the range of the property as well. The range is a class, which determines the type of the UML elements defining the element, represented by the base class. For the sentence the label annotation of the range class is used.

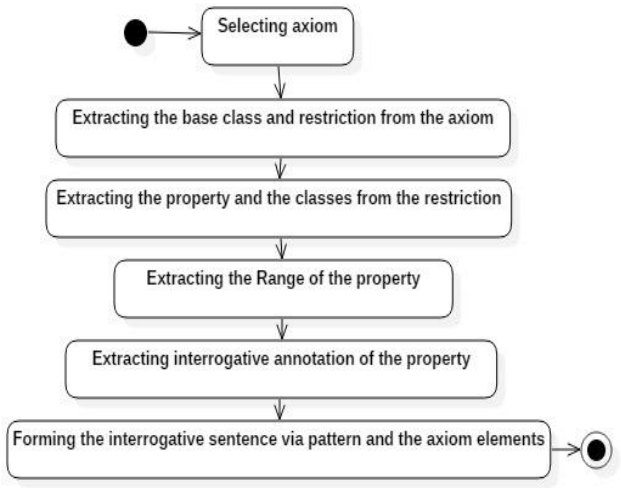

Fig. 6. Interrogative sentence generation

Generating interrogative (Fig. 6) and declarative sentences have similar algorithms. Both need the same axiom elements with the difference of the property annotation and the use of range. To form the sentence we use a different pattern according to the sentence type.

The pattern for interrogative sentences is shown in Fig. 7. The parts marked in grey ("What", "may", "the") are the static parts. The ones marked in yellow are the dynamic elements, which are extracted from the axiom. It is important that the formed interrogative sentence asks a question related to a base class concept and it is always answered at least with the concepts represented by the defining classes.

Fig. 7. Constructing pattern for interrogative sentences

To set a pattern for the declarative sentences we use the axiom structure itself. We start with the base class, continue with the property and finish with the defining classes. Using the sentence generation algorithm, we can form the different question types. Below we discuss each question type.

3.4.1. Fill in the blank

The idea of this question is to present one sentence with missing words, which have to be filled in by the user. It consists of two parts: the known part and the blanks. The sentence is generated via the declarative sentence method. However, the defining classes are included in the question as blank fields, i.e., these are the answers, which the user has to fill in. The number of the blanks corresponds to the number of defining classes in the used axiom.

3.4.2. True/false question

The QO uses the declarative sentence to generate a true/false statement. For a true statement, a declarative sentence is formed by the given algorithm without any modifications.

To construct a false statement the QO replaces one of the defining classes with such one that doesn't satisfy the axiom. It is good for the new class to represent a similar meaning to that of the replaced class. We pick a false class among the disjoint ones of the original class from the axiom. To take the set of disjoint classes, we extract the Disjoint Classes axioms, defining the original class.

3.4.3. Multiple choice question

Multiple choice questions consist of two parts. The first one is an interrogative sentence, generated from the chosen axiom. It asks a question and the user has to select the correct answer from three options. The second part has the three possible answers – one correct and two false ones.

The QO needs to form the set of the possible answers, which will be presented to the user. The incorrect answers are generated in the same way as the false part from the True/False question. They are represented by disjoint classes of the correct ones.

3.4.4. Multiple response questions

The multiple response type of question is similar to the previous one. The difference is that the set of possible answers consists of two correct ones and two false ones. In this case the QO has to choose an axiom with a restriction, containing not only a class, but a set of classes – a union or intersection of classes. This will allow the creation of a question with more than one true answers.

3.4.5. Select text questions

To implement the Select text type of item we need a generation of text, which we can manipulate as different elements as well. The item should be represented as a text, which has parts that can be chosen by the user according to the question requirement. The question requirement could say that the user must select the mistake in the text or the correct element. Since we need to generate text and not separate sentences, the algorithm starts with choosing the base class and then extracting the set of axioms that will be used.

The QO chooses randomly for each axiom what kind of declarative sentence to generate – true or false. Each sentence is generated, using the algorithm for the true/false type of question. For each sentence the defining classes are marked as selectable, i.e., the QO controls the GUI so that each restriction class can be presented as a radio button. Having the text, the QO defines the task, which would be either "Select the mistakes" or "Select the correct elements".

3.5. Implementation of the different tests

The implementation and forming of the test is also realized in the QO. To do this it interacts with the database using the database manager. This way it can read, write and update the needed data.

As it was mentioned in the previous section for the question generation, a set of suitable axioms needs to be extracted from the ontology. This is done by certain criteria, which depend on the test that is being generated.

The first thing to do when an attempt to start a test is made, is the QO decision what type of test is needed to be given to the user. QO takes the needed data for the current user from the database and, most importantly, what is the last test done by the user. Having information about the sequence of the tests, the QO easily decides what the current test should be. In the common case the name of the current test, taken from the database, shows its topic and what the questions should be about. There is another particular case that refers to the final package test – in it there is no defined topic, but the questions are based on the user's previous results (this is the last test from the sequence). If the current test is not the final case, the criteria for picking the set of axioms for the generation are common for each question from the test. This criterion is the topic of the test. Having the name of test, extracted from the database, it is used for searching in the ontology among the axioms. Here is used the test annotation of the axioms. The QO finds and extracts all axioms from the ontology that have test annotation values matching with the current test name. These axioms form the set of axioms needed for the generation of the questions for the test.

In case the current test is the final one, the set of axioms is formed differently. Again, the QO needs to get data from the database not only for the test name, but also for the results of the previous tests that the user has done. The extracted data gives information about the questions that were mistaken before. The QO gets the names of the base classes from the axioms that were used for generating the mistaken question. Then all axioms containing at least one of these classes are extracted from the ontology. They form the set of axioms that will be used for the generation of the questions for the final package test. In this way the questions from this test refer to an UML element that was not learnt well enough by the user before.

To complete the test, the QO also needs to define what the number of questions is that it is supposed to ask. This information is also taken from the database.

3.6. Implementation of the assessment of the answers

The assessment of the users' answers is implemented through the Assessment Operative. To assess the answers of the question (Fig. 8), the first steps for the AO is to get the user's answers and process them in a suitable way for their assessment in

the ontology. The processing of the given answers is as important as the assessing, since it forms it in the semantics of the ontology.

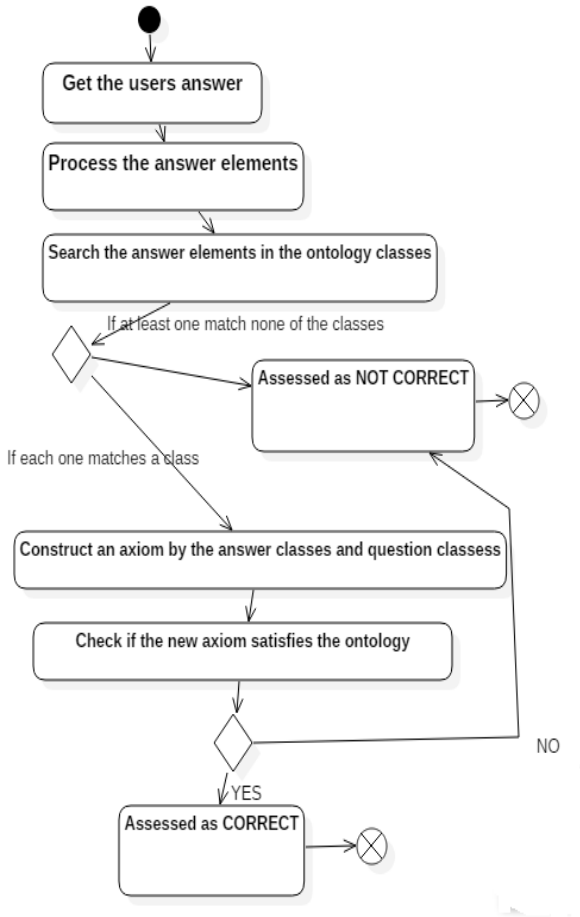

Fig. 8. Answers assessment algorithm

The answer itself could contain one or more elements, depending on how many answers are required by the question type (for example, the Fill in the blank question type could have more than one fields to fill in). The AO needs to process all of the elements of the answer, so they are in the form of an owl class of the ontology. The AO formats the string of each answer with respect to the ontology classes naming a convention. Afterwards, a search amongst the ontology classes is done not only amid the names of the ontology class, but also in their label annotations because thus it is more efficient. This is a good approach for searching in an ontology because the label annotations could give more names for one concept, if there are such in the ontology. The objective of the search is to find an ontology class for each of the answers given by the user. The results will be used to assess the answer.

If the process of the user's answer is successful, i.e., matches with the answers elements classes are found in the ontology, the assessment can proceed. The idea for

deciding if the answer is correct is to use the matching ontology classes to form an axiom ("answer axiom"). Then, it could be checked if this axiom satisfies the ontology knowledge. However, to construct this axiom, more information is needed. It will be taken by the axiom that was used for the generation of the question (the "question axiom"). The AO already has the question axiom because, as it was discussed, the QO sends this information when the question is asked. The needed elements are extracted from the question axiom – the base class, the property (if there is such), the type of the axiom and the type of the expressions in the axiom (in case there are any). Basically, the classes, formed by the answers elements, would be the defining classes in the answer axiom. They are the one used in the restriction.

Having extracted the base class and the property (if there is such in the question axiom) as well as the classes from the answer, the answer axiom is created. The type of the axiom is the same as the question axiom and so are the restriction type and the expression types. Is it checked if the answer axiom is contained in the ontology or if it satisfies the ontology rules. If this is true, the answer is assessed as correct.

Of course, there are other traditional conditions that depend on the answer assessment. If the answer is empty, i.e., no answer is given, the question is assessed as a wrong answer. In case the given answer can't be found in the ontology as a class or classes (depending on the answer elements), it is considered that it is not a UML concept and assessed as wrong.

It is important to mention that when the last question of the test is answered, AO updates the database by the database manager. The updated information is about the taken test – which test is taken by which user, and also the data about the mistaken questions is updated.

3.7. Implementation of the test generation environment

The architecture of the proposed test generation environment is decomposed and implemented in five separate Java packages (Fig. 9). Three classes serve the working with the database. One of them takes care of the connection with the database. The second one is intended for the authorization of the user. The third one is a manager class that provides the needed functions for reading and editing the database. This is the TestDbManager and it is created for a single user. This means that its instances are specific for each logged in user and they perform database operations only with data of this user. The ontology loading class (loader) uses the OWL API. For the GUI there are three classes developed – for the login frame, for showing the results and the main window – for taking the test. QuizStarted is the initial class of the system and it loads the login frame.

There is one more manager class and it is developed to manage the two JADE agents. It creates, starts and setups them, when the user is ready to start with the test. The two agents are developed by using JADE. These are the classes CheckerAgent (Assessment Operative) and QuestionAgent (Questioner Operative). Both have implemented inner classes that are actually their behaviors. They implement an agents' sensor and effectors over the GUI and their main functionalities.

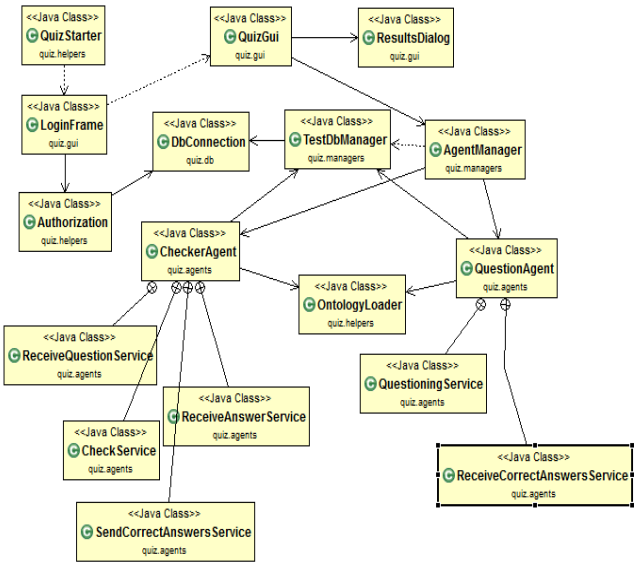

Fig. 9. Class diagram of test generation environment

4. Conclusion

The introduced environment generates tests, which consist of different question types in accordance with the QTI standard. This standard was chosen because it offers a well-structured specification for e-Learning materials such as structure of questions, assessments, and results. This structure is in XML format, which implies a suitable format for storage and exchange. Since the question types generated by the environment are referred to this standard, the most convenient and useful future work on the prototype is a new feature to generate these questions in XML format.

Currently, the environment is used in the real education at the Faculty of Mathematics and Informatics of the University of Plovdiv.

***Acknowledgments*:** The research is partially supported by the NPD – Plovdiv University, under Grant No MU17-FMI-001 "EXPERT (Experimental Personal Robots That Learn)", 2017-18 and Grant No FP17-FMI-008 "Innovative software tools and technologies with application in research in mathematics, informatics and pedagogy of education", 2017-2018.

References

1. Stoyanov, S., I. Popchev. Evolutionary Development of an Infrastructure Supporting the Transition from CBT to e-Learning. – Cyberbetics and Information Technologies, Vol. **6**, 2006, No 2, pp. 101-114.
2. Stoyanov, S., H. Zedan, E. Doychev, V. Valkanov, I. Popchev, G. Cholakov, M. Sandalski. Intelligent Distributed e-Learning Architecture – In: V. M. Koleshko, Ed. Intelligent Systems. InTech, 2012, pp. 185-218.
3. Stoyanov, S., I. Popchev, E. Doychev, D. Mitev, V. Valkanov, A. Stoyanova-Doycheva, V. Valkanova, I. Minov. DeLC Educational Portal. – Cybernetics and Information Technologies, Vol. **10**, 2010, No 3, pp. 49-69.

4. S a n d a l s k i, M., A. S t o y a n o v a-D o y c h e v a, I. P o p c h e v, S. S t o y a n o v. Development of a Refactoring Learning Environment. – Cybernetics and Information Technologies, Vol. **11**, 2011, No 2, pp. 46-64.
5. S t o y a n o v, S., I. G a n c h e v, I. P o p c h e v, M. O'D r o m a. An Approach for the Development of InfoStation-Based e-Learning Architectures. – Compt. Rend. Acad. bulg. Sci., Vol. **61**, 2008, No 9, pp. 1189-1198.
6. M i t e v, D., S. S t o y a n o v, I. P o p c h e v. Selbo2 – An Environment for Creating Electronic Content in Software Engineering. – Cybernetics and Information Technologies, Vol. **9**, 2009, No 3, pp. 96-105.
7. S t o y a n o v, S., I. G a n c h e v, I. P o p c h e v, I. D i m i t r o v. Request Globalization in an InfoStation Network. – Compt. Rend. Acad. bulg. Sci., Vol. **63**, 2010, No 6, pp. 901-908.
8. D o y c h e v, E., A. S t o y a n o v a-D o y c h e v a, S. S t o y a n o v, V. I v a n o v a. AgentBased Support of a Virtual e-Learning Space – In: ICCCI'16, Halkidiki, Greece, 28-30 September 2016.
9. S t o y a n o v, S., D. O r o z o v a, I. P o p c h e v. Virtual Education Space – Present and Future, Jubilee International Research Conference "The New Idea in Education" on the Occasion of the 25th Anniversary of the Establishment of Burgas Free University, 20-21 September 2016.
10. S t o y a n o v, S., V. V a l k a n o v a, I. P o p c h e v, I. M i n o v. A Scenario-Based Approach to Creating a Virtual Environment for Secondary School Instruction. – Cybernetics and Information Technologies, Vol. **8**, 2008, No 3, pp. 86-96.
11. G r a m a t o v a, K., S. S t o y a n o v, E. D o y c h e v, V. V a l k a n o v. Integration of e-Testing in an IoT e-Learning Ecosystem. – In: Virtual e-Learning Space (BCI'15), 2-4 September 2015, Craiova, Romania, 2015, ACM. ISBN 978-1-4503-3335-1/15/09.
12. IMS QTI Standard.
www.imsglobal.org/question/index.htm.
13. S h i r u d e, A., S. T o t a l a, S. N i k h a r, V. A t t a r, J. R a m a n a n d. Automated Question Generation Tool for Structured Data. – In: International Conference on Advances in Computing, Communications and Informatics (ICACCI'15), Kerala, IEEE, 2015, pp. 1546-1551**.**
14. V i n u, E. V., K. P. S r e e n i v a s a. A Novel Approach to Generate MCQs from Domain Ontology: Considering DL Semantics and Open-World Assumption. – Journal of Web Semantics: Science, Services and Agents on the World Wide Web**,** Vol. **34,** 2015, pp. 40-54**.**
15. N g u y e n, M. L., S. C. H u i, A. C. M. F o n g. Large-Scale Multiobjective Static Test Generation for Web-Based Testing with Integer Programming. – IEEE Transactions on Learning Technologies, Vol. **6**, 2013, No 1, pp. 46-59.
16. C h u, M.-H., W.-Y. C h e n, S.-D. L i n. A Learning-Based Framework to Utilize E-HowNet Ontology and Wikipedia Sources to Generate Multiple-Choice Factual Questions. – In: Proc. of IEEE Conference on Technologies and Applications of Artificial Intelligence, Tainan, 2012, pp. 125-130.
17. G u s e v, M., S. R i s t o v, G. A r m e n s k i. e-Testing Question Development Technologies and Strategies. – In: IEEE Global Engineering Education Conference, Tallinn, 2015, pp. 52-560.
18. A l d a b e, I., M. M a r i t x a l a r. Semantic Similarity Measures for the Generation of Science Tests in Basque. – IEEE Transactions on Learning Technologies, Vol. **7**, 2014, No 4, pp. 375-387.
19. L i u, M., R. A. C a l v o, A. A d i t o m o, L. A. P i z z a t o. Using Wikipedia and Conceptual Graph Structures to Generate Questions for Academic Writing Support. – IEEE Transactions on Learning Technologies, Vol. **5**, 2012, No 3, pp. 251-263.
20. S t a n c h e v a, N., A. S t o y a n o v a-D o y c h e v a, S. S t o y a n o v, I. P o p c h e v, V. I v a n o v a. A Model for Generation of Test Questions. – Compt. Rend. Acad. bulg. Sci., Vol. **70**, 2017, No 5, pp. 619-630.
21. S t a n c h e v a, N., A. S t o y a n o v a-D o y c h e v a, I. P o p c h e v, S. S t o y a n o v. Automatic Generation of Test Questions Using Ontologies. – In: Proc. of IEEE 8th International Conference on Intelligent Systems, Sofia, Bulgaria, 4-6 September 2016, pp. 741-746.