Computer Science • 13 (4) 2012 http://dx.doi.org/10.7494/csci.2012.13.4.35

Michał Śmiałek
Norbert Jarzębowski
Wiktor Nowakowski

TRANSLATION OF USE CASE SCENARIOS TO JAVA CODE

Abstract *Use cases are usually treated as second class citizens in the software development chain based on models. Their textual descriptions (scenarios) are treated as informal input to more formal design models that can then be (semi-)automatically transformed down to code. In this paper we will show that use case scenarios can gain precise metamodel-based notation and semantics enabling automatic processing. What is more, we will show transformation algorithms that can transform use case scenarios directly to dynamic code in Java. The presented transformation can generate the full structure of the system following the MVP architectural pattern, including complete method contents for the application logic (Presenter) and presentation (View) layers. It also provides a code skeleton for the domain logic (Model) layer. The use case notation and the transformation were implemented within a sophisticated tool suite. Based on this, the paper discusses the evaluation efforts based on a case study.*

Keywords use cases, runtime semantics, model transformation

1. Introduction

Use cases are currently one of the most widely used notations in specifying functional requirements. Developing a use case model is normally the first step in the software development (SD) process, especially when model-based techniques are used. Use cases serve as the process drivers in many contemporary SD methodologies. For each of the use cases, specific design artifacts are developed (e.g. interaction diagrams) thus contributing to the overall system structure expressed through component diagrams and/or class diagrams. This finally leads to a code that implements the specific functionality defined through use cases.

The use case notation is standardized to some extent within the UML specification. However, this part of UML is very poorly defined, as discussed already in 1999 [18], and the situation has not changed up till now in UML 2. For this reason, use cases were always excluded from the main stream of model-driven transformation paths. Translation from use case to design and code was made manually, sometimes supported by specific best practice heuristics. However, use case descriptions tend to become more and more precise thus improving their applicability in the software development chain. This is done through applying precise quality-related guidelines for textual use case descriptions [14, 10] and proposing alternative visual notations [21].

By bringing precision to use cases we shift into their becoming "first class software citizens" within software development. This is associated with automating the translation of use case models and their descriptions into various other models. The first group of attempts at this issue consists in generating analysis models, like conceptual class diagrams and some dynamic diagrams (eg. state machines), as reviewed in [24]. The second group approaches to deriving design models. In [19] we can find a detailed set of rules for transforming into architectural design models, including dynamic sequence diagrams. The authors of [23] propose a simple meta-model for describing textual use case descriptions and transforming them into static component models. A similar approach, but extended with natural language parsing and generating service-oriented component models was proposed in [3] (see also [1]). In [17] there is proposed a set of heuristics and a resulting algorithm to generate detailed design class models from constrained natural language use case scenarios. These models are purely static but do contain complete sets of class operations for the application logic, and can become the start to implementation efforts.

Ultimately, this leads to relating use case descriptions to code. In some approaches, use case models (as such) are derived from code by applying certain reverse-engineering techniques (see [2], [6], [16], [7]). This is mainly used to better understand and trace from the source code of legacy applications. Though, recently there have emerged approaches to generate code directly from use case descriptions. In [5] there is proposed a framework to produce three-tier-based code. Textual scenarios are first translated into special trace scripts ("procases") and then into the contents of application logic layer class methods. Similarly, [20] proposes a set of rules to translate constrained natural language scenarios directly into Java code.

Recently, in RSL [8] there was introduced a new specification of the use case model, together with a precise notation for use case representations (scenarios). This starts an opportunity to introduce use cases as "first-class citizens" on the transformation path, thus introducing automatic translation from the use case models to more detailed models. Within the ReDSeeDS project (www.redseeds.eu) [22] there were introduced such transformations leading to detailed design models, including classes and interactions. What is more, this can lead to treating use case scenarios as a kind of programming language that can significantly speed-up the software development (evolution) cycles [20].

In this paper we extend the above results with a detailed description of the process of translation from the precisely defined RSL-based scenarios to Java-based code. The source language (RSL) is expressed according to the rules of software language engineering (see [11]). Based on this, the translation is done in accordance with the rules of model transformation (see e.g. [12]). We show that it is possible to translate scenarios written in semi-natural language into the fully operational dynamic code of the application logic and presentation layers. The resulting process is shown in Figure 1. The RSL scenario model is transformed into the UML design model with Java method bodies by using an automatic transformation in the MOLA language [9]. The UML model can then be easily generated into code using the standard mechanisms of CASE tools.

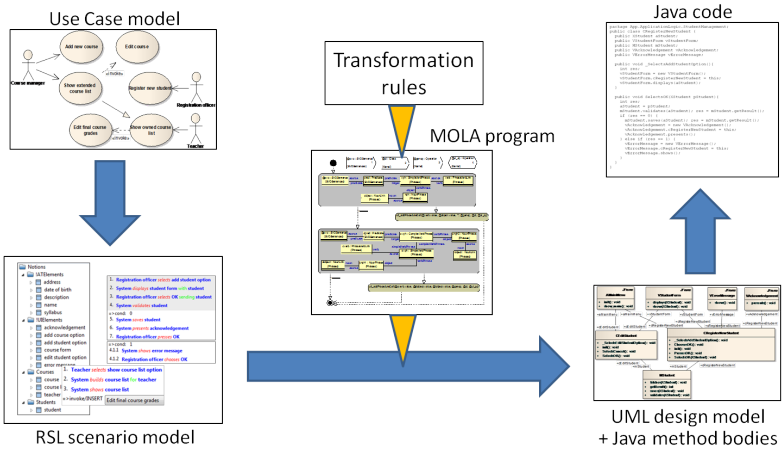

Figure 1. Overview of the translation process.

2. Precise notation for use case scenarios

The Requirements Specification Language (RSL) is a semi-formal language for specifying software requirements. RSL employs use cases for defining the system's behaviour. Each use can be detailed by one or more textual scenarios consisting of sentences in

constrained natural language. Notions and phrases used in scenario sentences are linked to elements of the domain model. These elements can have their definitions specific to the system's domain. Such notation, separating descriptions of the system's behaviour from descriptions of the domain problem, is easily understandable to different audiences, including end-users, thus allowing them to discuss and partially construct software through writing the application logic at the level of requirements. On the other hand, to facilitate automatic transformations from requirements to design-level models and code, RSL syntax is precisely defined through a meta-model in MOF [13]. The full RSL specification, including abstract syntax, concrete syntax and semantics, can be found in [8]. Here, we describe only some simplified language constructs, which are relevant to the process of translation of use case scenarios into code.

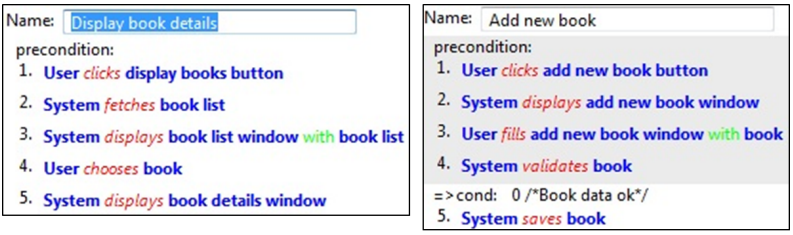

Figure 2. Concrete syntax for use case scenarios.

Figure 2 shows examples of use case scenario notation. Every scenario is a numbered sequence of actions that are performed either by an actor or the system and lead to success or failure in reaching the use case goal. Every such action is expressed by a single sentence in the SVOO grammar. Sentences in this grammar are composed of a subject, a verb and an object, optionally followed by a second indirect object. The subject indicates who performs the action (the "user" or the "system"). The objects in a sentence represent notions from the business or the system domain (eg. "book", "book details window"). The verb, in turn, is strongly relevant to the direct object (the VO part of a sentence) – it describes an operation that can be performed in association with that object (eg. "validate book", "display book details window"). The indirect objects (VOO part) can represent detailed data that is passed while performing actions (eg. "displays book list window with book list"). In addition to action sentences, also condition sentences can be used to define an alternate sequence of actions that is performed according to the condition defined. Conditions relate to system state or actor's decision.

Figure 3 shows a fragment of the RSL metamodel that deals with use case scenarios and sentences. Every Requirement (ie. a use case) in RSL can have at least one representation in the form of ConstrainedLanguageScenario. Scenarios, in turn, are composed of ordered sets of scenario steps (ConstrainedLanguageSentence metaclass) that can be of two types: SVOSentence (for expressing actions performed by the actors or the system) or ConditionSentence (for defining conditions for the alternative

scenario paths). An SVOSentence is composed of a Subject and a Predicate that are in fact hyperlinks to definitions of the proper phrases in a vocabulary.

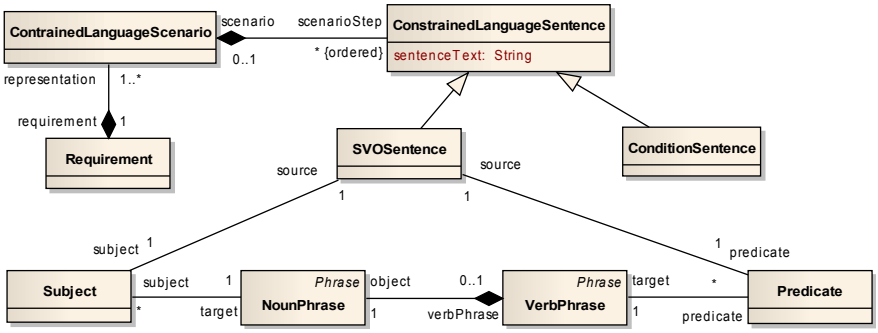

Figure 3. Abstract syntax of use case scenarios and sentences.

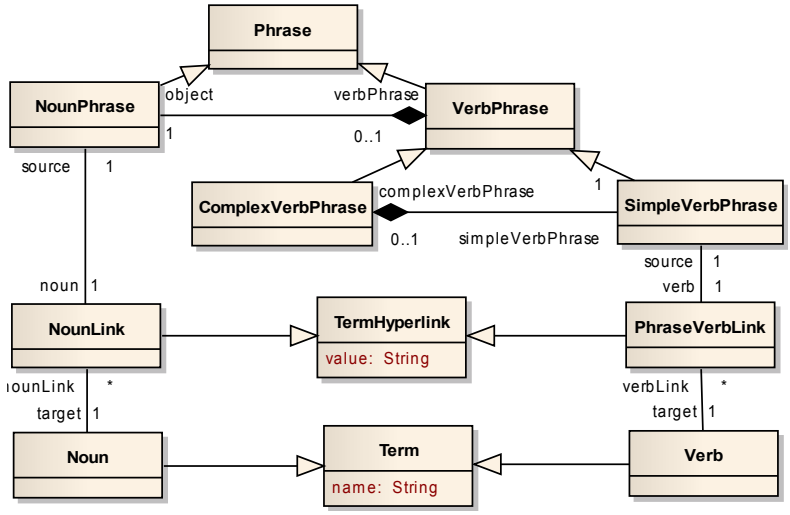

Figure 4. Abstract syntax of sentence phrases.

The structure of phrases is shown in Figure 4. All the phrases are sequences of hyperlinks (see NounLink and PhraseVerbLink) pointing at terms (see the Term metaclass and its subclasses: Noun and Verb) in a global terminology. These terms (with their forms, inflections, cases) are stored in a WordNet [4] repository. The Subject points to a NounPhrase that is linked to just one Noun. The Predicate points to one of the two VerbPhrase specialisations. If a sentence contains only the direct object (SVO sentences), the predicate points to a SimpleVerbPhrase. This subtype of VerbPhrase contains a link (see PhraseVerbLink) to one Verb that precedes the

NounPhrase (inherited from the VerbPhrase). Phrases containing both the direct and indirect object (SVOO sentences) are stored in a ComplexVerbPhrase. This subtype of the VerbPhrase links an instance of a SimpleVerbPhrase followed by an indirect object (see NounPhrase) inherited from the VerbPhrase.

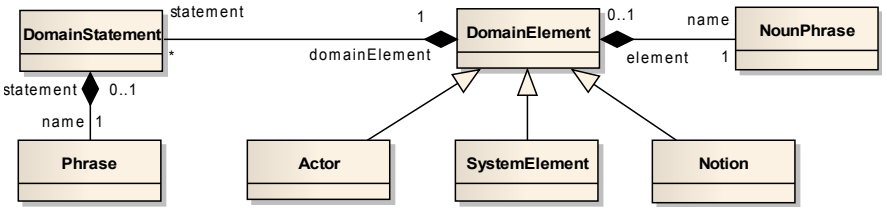

Figure 5. Abstract syntax of domain vocabulary elements.

All the phrases used in scenarios to describe system's behaviour need to be precisely defined in the context of the problem domain. In RSL, it is done by introducing domain elements that together form a domain vocabulary. The metamodel showing the structure of domain elements is presented in Figure 5. All phrases that refer to the same noun used in a scenario are grouped within a DomainElement, where a NounPhrase linked to that noun is the element's name. Other Phrases, (especially VerbPhrases) pointing to the noun are contained within a DomainElement as DomainStatements. All these DomainStatements have their definitions describing behavioural features of the related nouns. For example, "validate book" has a different meaning than "save book".

In order to distinguish data to be handled by the system from the actors interacting with its components, the DomainElements have three subtypes. Every part of the composite system that is referred to in scenarios is modelled as an instance of a SystemElement. All actors are modelled as instances of Actor. Other elements of domain vocabulary that are used in scenarios (domain entities, data, UI elements, etc.) and are not system elements or actors are modelled as instances of a Notion.

3. Equivalence of use case scenarios and Java

This section describes how requirement specification elements are transformed into code. The transformed code is written in Java and its structure is based on the MVP (Model-View-Presenter) [15] architectural pattern. Each generated class consists of the source object's name and a predefined prefix that emphasizes layer affinity:

- "V" for view layer classes. The view layer classes are GUI elements responsible for presenting information and interacting with the user.
- "M" for model layer classes. They are created based on the business objects stored in the system and contains all operations that can be performed on those objects.

- "C" for controller/presenter layer classes. The controller/presenter classes are responsible for conducting the system's observable behavior.
- "X" for data transfer object (DTO) classes. They are used to transfer data between layers, therefore they contain only attributes and no operations (except for the constructor).

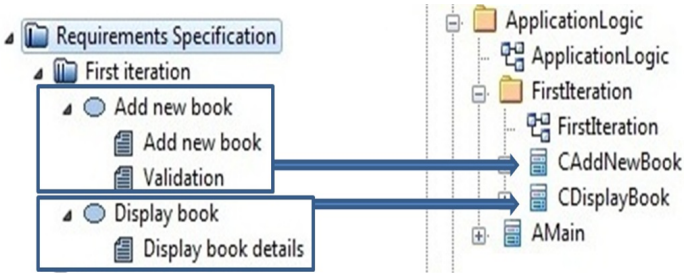

Figure 6. Generation of the controller/presenter layer classes.

Figure 6 presents an example transformation of use cases into classes. Each use case transforms into one class in the controller/presenter layer. The name of the class consist of the "C" prefix and the source use case's name. Scenarios do not transform directly into any objects, but their sentences are the basis for generating class operations. The precise rules of scenario sentence processing will be explained further in this paper. In turn, Figure 7 shows how the notions from the domain specification are transformed. Each notion transforms into one data transfer object class and one model layer class. To maintain the consistency of specification and code, the generated model classes are arranged in packages named as in the source model (cf. the "Books" package).

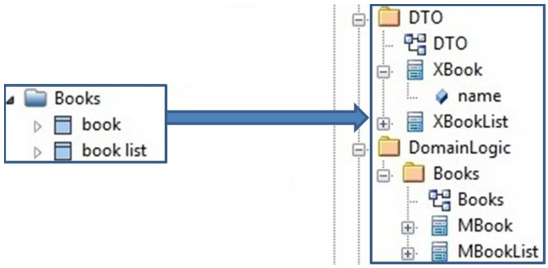

Figure 7. Generation of the model layer classes.

The notions that represent UI elements are placed in a special !UIElements package. They are transformed differently to "normal" domain elements as shown in Figure 8. Each such notion transforms into one class in the view layer. It is worth mentioning that notions from the !Buttons package do not transform into any new

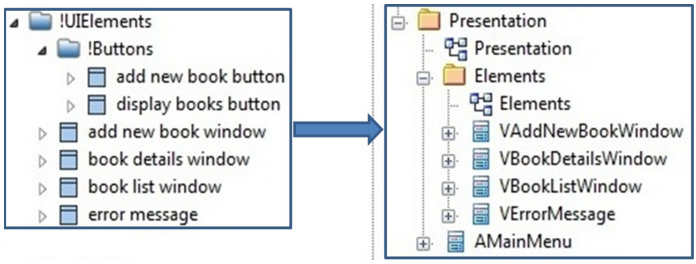

Figure 8. Generation of the view layer classes.

classes because buttons are always part of some window or form. Additionally a class representing the application's main window is generated.

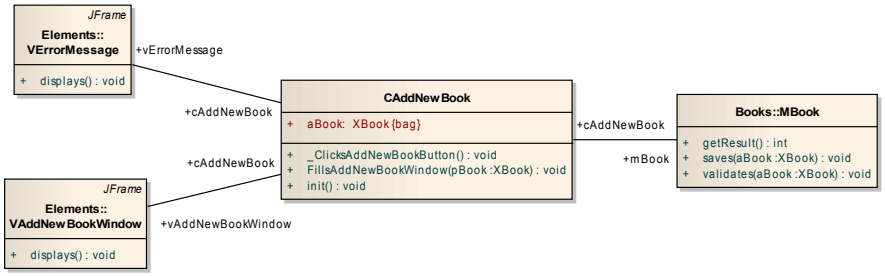

Figure 9. General structure of the generated code.

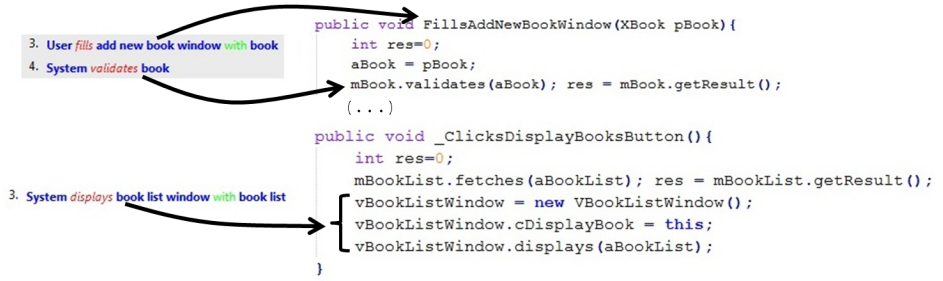

Figure 10. Dynamic code generated for SVO sentences.

To summarise the above rules, the general structure of the generated code for the Add new book use case from Figure 2 is presented in Figure 9. The operations in this model are generated based on the scenario contents. In this section we will give an example of such a generation, and in the next section detailed rules will be described (see sentence processing algorithm in Figure 14). Figure 10 presents two examples of code generated for SVOO sentences. The first example shows a code

generated for an "Actor-to-System" sentence. The operation name is constructed from a verb phrase and the (optional) operation parameter – from an indirect object. Then, the XBook object is assigned to a local aBook variable and passed to the validates operation in the MBook class. The second example shows code for a "System-to-Actor" sentence. The system constructs a new VBookListWindow object and then displays it with a previously fetched XBookList data (cf. aBookList).

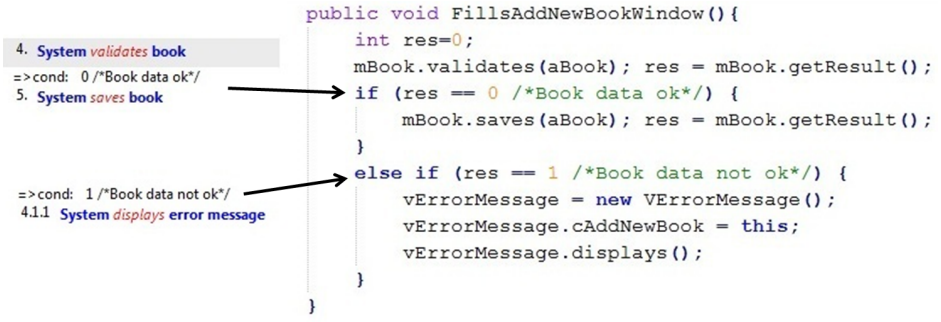

Figure 11. Dynamic code generated for condition sentences.

Figure 11 presents a code generated for the condition sentences. A condition sentence's text is transformed directly into the if-else structure code. It's a less user-friendly approach but allows the scenarios' author to better control the scenario flow. The body of the if-else structure is generated based on sentences occurring after the condition. In an example presented in Figure 11 the controller/presenter class assigns the result of validation to a local variable and then, depending on the validation result, the if-else structure decides whether to save the XBook object or to display an error message.

The full code (slightly abbreviated for clarity) generated for the CAddNewBook class is presented in Figure 12. It contains all the necessary declarations, imports (not shown in the Figure) and a constructor. It contains the operations presented in Figures 11 and 10. It can be also noted that appropriate references to the view and model layer objects are generated (see mBook, textsfaMainMenu, etc.).

4. Transforming use case scenarios to Java

This section precisely describes the mechanism for transforming requirements into code. It starts with general high-level algorithms written in the UML activity diagram notation and moves to discuss the actual MOLA procedures used to transform the RSL specification into code. Figure 13 shows the high-level algorithm for use case processing. The algorithm starts from the requirement specification level and processes each use case. While processing a use case, the algorithm takes and processes each of its scenarios and scenario sentences according to the algorithm presented in Figure 14.

```
public class CAddNewBook {
  public Xbook aBook;
  public VAddNewBookWindow vAddNewBookWindow;
  public Mbook mBook;
  public AMainMenu aMainMenu;

  public CAddNewBook() {
  }

  public void finalize() throws Throwable {
  }

  public void _ClickAddNewBookButton() {
    int res = 0;
    vAddNewBookWindow = new VAddNewBookWindow();
    vAddNewBookWindow.cAddNewBook = this;
    vAddNewBookWindow.display();
  }

  /**
   * @param pBook    redseeds_uid767171408326126930O-6917814001634927640-
   * 554483506756785203-8217886436700875814redseeds_uid
   */
  public void FillsAddNewBookWindow(XBook pBook) {
    int res = 0;
    aBook = pBook;
    mBook.validates(aBook);
    res = mBook.getResult();
    if (res == 0 /*Book data ok.*/) {
      mBook.saves(aBook);
      res = mBook.getResult();
    }
    else if (res == 1 /*Book data not ok.*/) {
      vErrorMessage = new VErrorMessage();
      vErrorMessage.cAddNewBook = this;
      vErrorMessage.displays();
    }

  public void init() {
    aBook = new Xbook();
  }
}
```

Figure 12. Full code of a generated controller/presenter layer class.

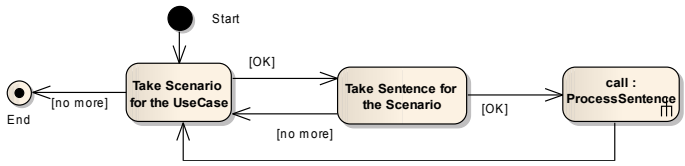

Figure 13. General algorithm for use case processing.

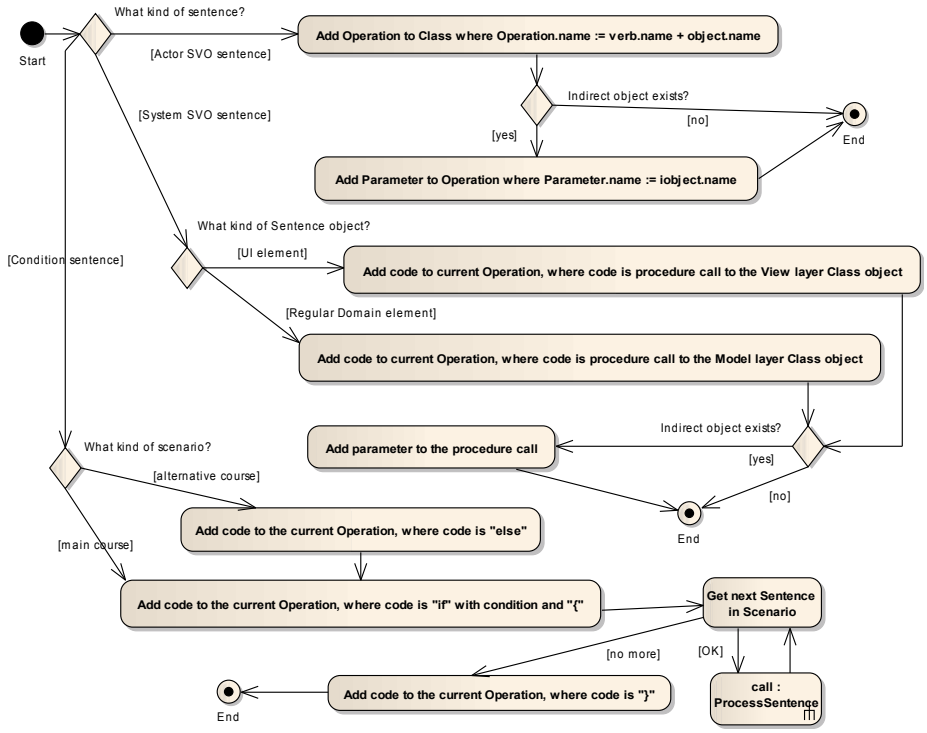

Figure 14. General algorithm for scenario sentence processing (ProcessSentence procedure).

Before we explain the algorithm, it is very important to identify the direction of control flow between an actor and a system. Depending on the flow direction, the target class for generating the appropriate operation's code is chosen according to the following general rules:

- Actor-to-System (subject is an actor) sentence generates an operation in the controller/presenter layer class.
- System-to-Actor (subject is a system, direct object in a UI element) sentence generates an operation in the view layer class.
- System-to-System (subject is a system, direct object is a regular domain notion) sentence generates an operation in the model layer class.

Having this in mind, we can go to Figure 14 that presents the general algorithm for sentences processing. The algorithm can also be described by the following textual rules:

- Each Actor-to-System SVO sentence is transformed into an operation. The operation's name is constructed from the VerbPhrase pointed by the predicate of the sentence.

- Each Actor-to-System SVOO sentence is transformed in the same way as the SVO sentence but additionally the operation's parameter (based on the indirect object) is added.
- Each System-to-Actor (UI element) sentence is generated into a code that calls a proper procedure from the view layer class object.
- Each System-to-System (normal domain element) sentence is generated into a code that calls a proper procedure from the model layer class object.

Finally, condition sentences are transformed differently to regular SVO sentences. Each condition sentence is transformed into the if-else structure where the operations created from the basic scenario sentences are added to the if structure and the operations created from the alternative scenario sentences are added to the else structure.

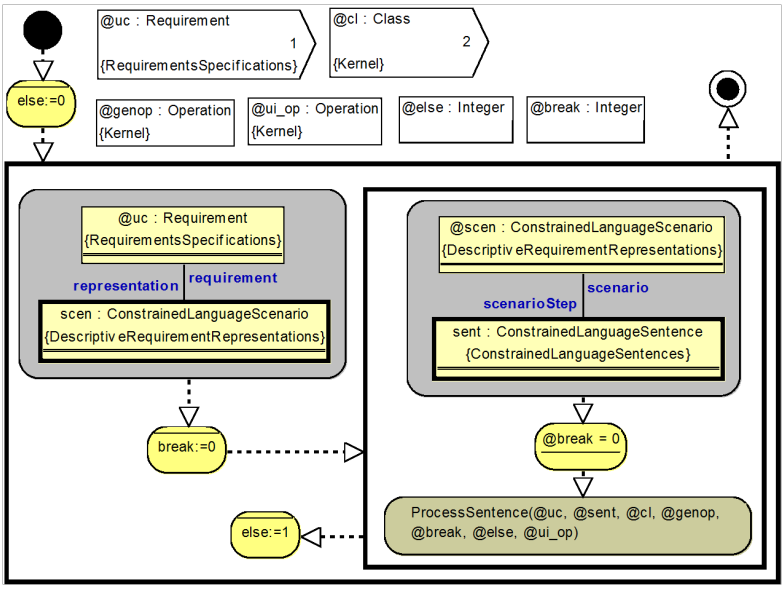

Figure 15. Main use case transformation procedure.

Based on the above described general algorithm we can now construct the transformation program. Figure 15 presents the main procedure written in MOLA (MOdel transformation LAnguage, [9]). MOLA is a graphical transformation language based on pattern matching at the metamodel level (see appropriate language tutorials at mola.mii.lu.lv). It is capable of modifying a model that complies to a specific abstract syntax written in MOF or transforming it into another model in a similarly defined different language. The syntax is generally similar and has similar control-passing semantics as UML activity models. An elementary instance transformation statement in MOLA is called a rule. Rules are represented by grey rectangles with rounded corners. Their main role is to check whether the elements in the model match

the metamodel-related objects presented in the rule. Other important MOLA elements are loops. They are represented by rectangles with thick edges and controlled by a thick edged element inside. In Figure 15, the procedure iterates and matches each ConstrainedLanguageScenario representing a Requirement and takes each of its scenario's (iterates over) ConstrainedLanguageSentence for processing. Note that the metamodel definition for ConstrainedLanguageScenario and ConstrainedLanguageSentence is described in Figure 3.

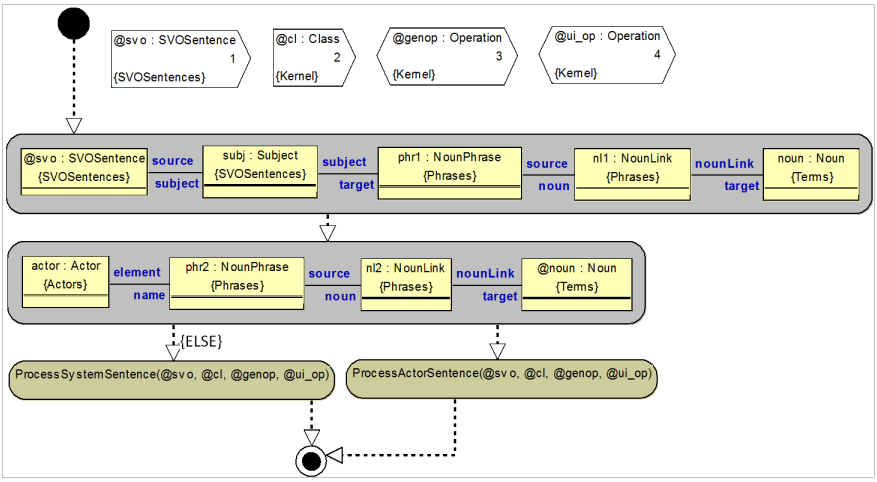

Figure 16. SVO sentence processing procedure.

The procedure presented in Figure 16 checks whether the sentence's subject is an actor or a system and decides which procedure to choose next (compare the object structure in the rules with the metamodel in Figures 3 and 4). The procedure ProcessActorSentence for processing Actor-to-System sentences is presented in Figure 17. The first two rules checks whether the sentence is an SVO or an SVOO sentence. Regardless of the result, the procedure sets the str variable to the verb and the direct object names. If the sentence is an SVOO sentence, the procedure sets the param1 variable to the indirect object name. The following rules checks if an operation with the same name already exists. If not, the transformation creates a new operation (with the Comment containing the initial method body) with the name based on the str variable and a parameter based on the param1 variable. The last element of the procedure it to call the utl_genJButton procedure that creates a new button within the proper application window class (more precisely: within the ui_op window constructor). The procedure ProcessSystemSentence is much simpler as it just generates appropriate procedure calls, and will be omitted for brevity.

Finally, Figure 18 describes how the condition sentences are processed. The procedure creates the if-else structure by generating the "if" or the "else if" code fragment, depending on the else variable. The condition's text is transferred directly into code.

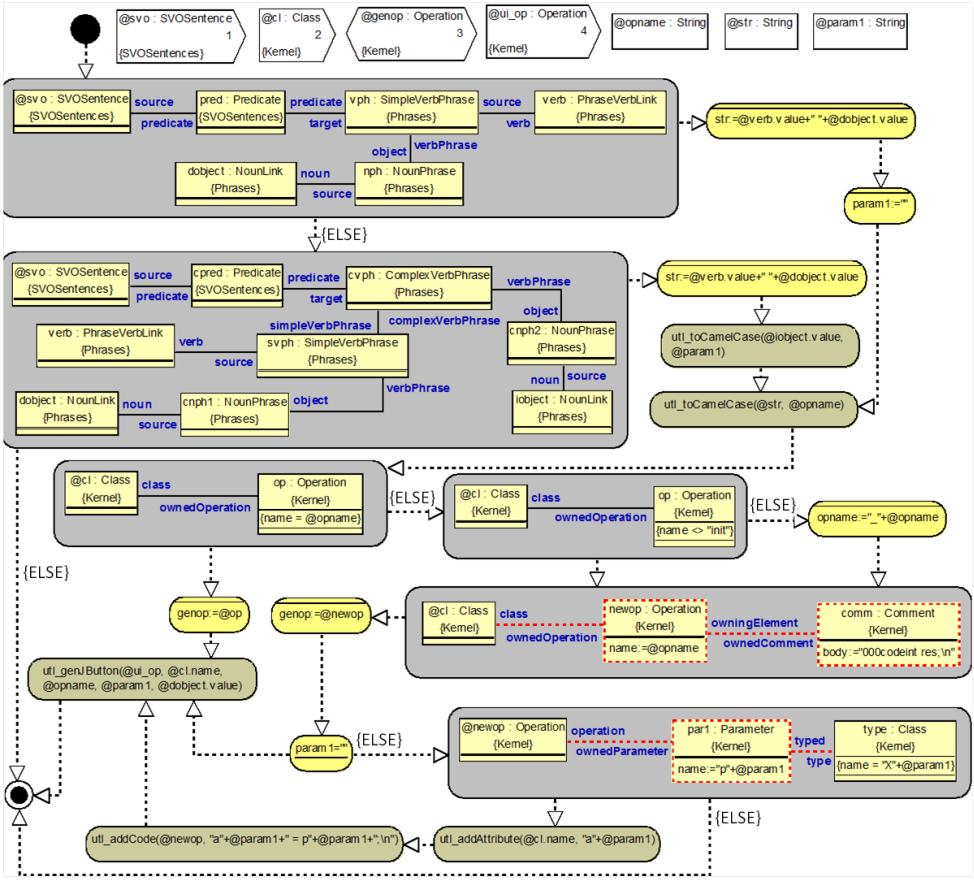

Figure 17. Actor-to-System sentence processing procedure.

The ProcessIf procedure is responsible for processing the sentences of the scenario that follow so that the code for those sentences is generated inside the if-else structure. It is similar to the main procedure in Figure 15.

5. Initial validation and conclusions

In order to validate the presented transformation, an appropriate tooling framework has been prepared. The framework consists of the ReDSeeDS tool developed within the ReDSeeDS project [22] that provides a comprehensive RSL editor. This editor allows writing use case scenarios in accordance with the rules of the RSL language grammar presented in Section 2. The use case scenarios are stored in a model repository, based on the RSL metamodel. The ReDSeeDS tool, thanks to the built-in transformation engine, also enables performing transformation programs written in

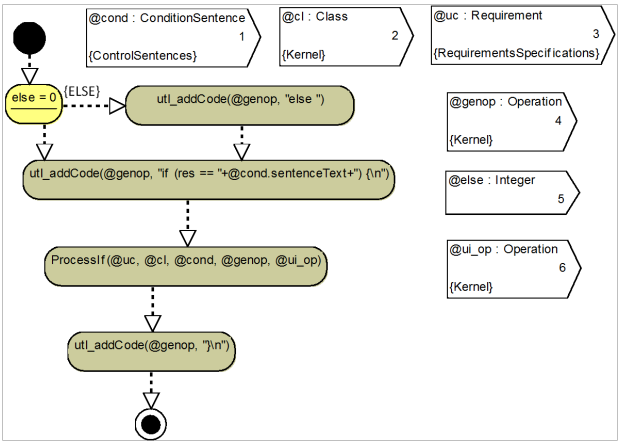

Figure 18. Condition sentence processing procedure.

MOLA. The transformation program, implementing translation rules described in this paper, has been developed to be used with ReDSeeDS. This transformation takes an RSL model stored in a model repository as an input and translates it into Java code. This, however, is not done directly. First, the transformation creates a UML class model with embedded textual Java-compliant contents for the class operation descriptions. This Java-enhanced UML model can be easily generated into Java code using standard capabilities of a UML tool integrated within the framework (currently it is Enterprise Architect by SparxSystems). Since these two steps do not involve any manual intervention, in future implementations this can obviously be substituted by a single step. Future work also includes a simple natural language parser that would additionally facilitate determining scenario sentence parts. Currently, the sentence parts need to be marked manually by the developers.

This framework has been used in an experiment conducted during laboratory sessions with students. The students were instructed on RSL story constructs and had previously gained knowledge about constructing MVC/MVP style systems, using UML and Java. During the classes, they were formed into 8 groups consisting of 3-4 students each. All the groups were assigned a ready use case model of a Campus Management System, containing 12 use cases. The first assignment consisted in writing scenarios for the use cases. Four groups wrote the scenarios using the RLS editor, while four other groups used a structured use case editor built into Enterprise Architect (EA). The EA editor did not enforce any syntax for the scenario sentences, although allowed for almost identical structure of scenarios with conditions and notation for alternatives.

The students had 4 hours (2 lab sessions) to write their scenarios and were asked to write them only during the classes. All the groups managed to write good quality scenarios for all the assigned use cases. There were no significant differences between

the groups using EA and ReDSeeDS. The groups produced from 121 to 159 story sentences of all types. Based on this, the groups were asked to implement their systems in Java having 10 hours (5 lab sessions). Each story sentence was treated as complete if the system managed to pass appropriate data between layers and output "debug" messages. Two of the groups used the use case to Java transformation, two groups used a standard ReDSeeDS generator that produces ready three-tier class diagrams and sequence diagrams. The remaining four groups used manual translation into UML and then code generation within the EA. The first two groups of students managed to implement almost half of the functionality, where on average 68 out of 141 sentences were implemented. It has to be noted that these two groups had extended acceptance criteria where the "debug" messages for the presentation layer were substituted with Swing-style GUI forms. The last four groups of students managed to implement 21 sentences on average. The groups that used the ReDSeeDS standard transformation performed somewhat better with the average of 28 sentences.

The above simple experiment shows significant improvement in productivity when using the presented scenario translation. It also shows that quite inexperienced programmers and software designers (students) can benefit from removing the need to structure their code and write the application logic part. Instead of finding technological ways to implement the end-user functional requirements, they can concentrate on writing the data processing (domain logic) code. The presented approach allows for generating a ready skeleton for such code.

Acknowledgements

This research has been carried out in the REMICS project and partially funded by the EU (contract number IST-257793 under the 7th framework programme), see `http://www.remics.eu/`.

References

[1] de Castro V., Marcos E., Vara J. M.: Applying CIM-to-PIM model transformations for the service-oriented development of information systems. *Information and Software Technology*, 53:87–105, 2011.

[2] di Lucca G., Fasolino A., de Carlini U.: Recovering use case models from object-oriented code: a thread-based approach. In *Proc. 7th Working Conf. on Reverse Engineering*, pp. 108–117. IEEE, 2000.

[3] Ding Z., Jiang M., Palsberg J.: From textual use cases to service component models. In *Proc. 3rd Int. Workshop on Principles of Engineering Service-Oriented Systems*, pp. 8–14. ACM, 2011.

[4] Fellbaum C., editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[5] Franců J., Hnětynka P.: Automated generation of implementation from textual system requirements. In *Software Engineering Techniques*, volume 4980 of *Lecture Notes in Computer Science*, pp. 34–47. Springer; Berlin / Heidelberg, 2011.

[6] Grechanik M., McKinley K. S., Perry D. E.: Recovering and using use-case-diagram-to-source-code traceability links. In *Proc. 6th Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. on Foundations of Software Eng.*, pp. 95–104, 2007.

[7] Hirschfeld R., Perscheid M., Haupt M.: Explicit use-case representation in object-oriented programming languages. In *Proc. 7th Symp. on Dynamic Languages*, pp. 51–60. ACM, 2011.

[8] Kaindl H., Śmiałek M., Wagner P., Svetinovic D., Ambroziewicz A., Bojarski J., Nowakowski W., Straszak T., Schwarz H., Bildhauer D., Brogan J. P., Mukasa K. S., Wolter K., Krebs T.: Requirements specification language definition. Project Deliverable D2.4.2, ReDSeeDS Project, 2009. www.redseeds.eu.

[9] Kalnins A., Barzdins J., Celms E.: Model transformation language MOLA. *Lecture Notes in Computer Science*, 3599:14–28, 2004. Proc. of MDAFA'04.

[10] Kamalrudin M., Hosking J., Grundy J.: Improving requirements quality using essential use case interaction patterns. In *Proc. 33rd International Conference on Software Engineering*, pp. 531–540. IEEE, 2011.

[11] Kleppe A.: *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.

[12] Kleppe A. G., Warmer J. B., Wim B.: *MDA Explained, The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Boston, 2003.

[13] Object Management Group. *Meta Object Facility Core Specification, version 2.0, formal/2006-01-01*, 2006.

[14] Phalp K. T., Vincent J., Cox K.: Improving the quality of use case descriptions: empirical assessment of writing guidelines. *Software Quality Journal*, 15:383–399, 2007.

[15] Potel M.: Mvp: Model-view-presenter the taligent programming model for C++ and java. Technical Report, Taligent Inc., 1996.

[16] Qin T., Zhang L., Zhou Z., Hao D., Sun J.: Discovering use cases from source code using the branch-reserving call graph. In *Proc. 10th Asia-Pacific Software Eng. Conf.*, pp. 60–67. IEEE, 2003.

[17] Sarkar S., Sharma V. S., Agarwal R.: Creating design from requirements and use cases: bridging the gap between requirement and detailed design. In *Proc. 5th India Software Engineering Conference*, pp. 3–12. ACM, 2012.

[18] Simons A. J. H.: Use cases considered harmful. In *Proc. of the 29th Conference on Technology of Object-Oriented Languages and Systems-TOOLS Europe'99*, pp. 194–203, Nancy, France, June 1999. IEEE Computer Society Press.

[19] Śmiałek M.: *Software Development with Reusable Requirements-Based Cases*. Publishing House of the Warsaw University of Technology, 2007.

[20] Śmiałek M.: Requirements-level programming for rapid software evolution. In J. Barzdins, M. Kirikova, eds., *Databases and Information Systems VI*, chapter 3, pp. 37–51. IOS Press, 2011.

[21] Śmiałek M., Bojarski J., Nowakowski W., Ambroziewicz A., Straszak T.: Complementary use case scenario representations based on domain vocabularies. *Lecture Notes in Computer Science*, 4735:544–558, 2007. Proc. of MODELS'07.

[22] Śmiałek M., Kalnins A., Ambroziewicz A., Straszak T., Wolter K.: Comprehensive system for systematic case-driven software reuse. *Lecture Notes in Computer Science*, 5901:697–708, 2010. Proc. of SOFSEM'10.

[23] Šimko V., Hnětynka P., Bureš T.: From textual use-cases to component-based applications. *Studies in Computational Intelligence*, 295:23–37, 2010.

[24] Yue T., Briand L. C., Labiche Y.: A systematic review of transformation approaches between user requirements and analysis models. *Requirements Engineering*, 16(2):75–99, 2011.

Affiliations

Michał Śmiałek
Warsaw University of Technology, Warsaw, Poland, `smialek@iem.pw.edu.pl`

Norbert Jarzębowski
Warsaw University of Technology, Warsaw, Poland, `jarzebon@iem.pw.edu.pl`

Wiktor Nowakowski
Warsaw University of Technology, Warsaw, Poland, `nowakoww@iem.pw.edu.pl`

Received: 8.03.2012
Revised: 26.06.2012
Accepted: 3.09.2012