SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 33–57 P.

A Model-Driven Path from Requirements to Code

Audris Kalnins, Elina Kalnina, Edgars Celms, Agris Sostaks
University of Latvia, IMCS, Raina bulv. 29, Riga, LV-1459, Latvia
Audris.Kalnins@lumii.lv, Elina.Kalnina@lumii.lv, Edgars.Celms@lumii.lv, Agris.Sostaks@lumii.lv

Although there is a lot of support for model-driven development, few approaches offer support for a complete model-driven path from requirements to code. The approach proposed in this paper offers such a path fully supported by model transformations. The starting point is semiformal requirements containing behaviour description in a controlled natural language. A chain of models is proposed, including analysis, detailed design, and platform-specific models. A particular architecture style is chosen by means of selecting a set of appropriate design patterns for these models. We show how the required transformations can be informally defined and then implemented in the model transformation language MOLA. Thus, a prototype of the system is obtained which can then be extended in a model-driven way.

Keywords: model-driven development, transformations, requirements, UML.

1 Introduction

The main goal of this paper is to demonstrate how transformations could be used to support the full path from requirements to code in a model-driven development. Requirements are specified in the requirement specification language RSL [1, 2], which has been developed as part of the ReDSeeDS project [3]. A significant part of RSL is the specification of requirements for system behaviour in a controlled natural language. In this paper we demonstrate how such requirements can be used as the basis for transformations to code via Analysis, Detailed Design, and Platform-Specific Models. Models are generated according to a particular architecture style, including selection of appropriate design patterns for these models.

The Model Driven Architecture (MDA) approach [4] has evolved significantly since its launch in 2001. Now it has become just one of the versions of model-driven development (MDD) [5]. From the original chain of three models – CIM, PIM, and PSM – only the last two are used frequently, and typical MDA transformations support only the generation of PSM from PIM. The CIM model has got significantly less attention. In this paper, we assume that the proper contents of CIM are requirements.

The ReDSeeDS approach [3] used in this paper covers a complete chain of models for model-driven development – from requirements to code. Each transition in this chain is to a great degree assisted by formal model transformations. Although a specific chain of models is described here, the approach could be applied to any similar setting of models.

The first in the chain is the Requirements Model built in a special semiformal requirement language RSL (described in Section 4). The required behaviour specification

in this controlled natural language is sufficiently precise; therefore, this specification can be processed by model transformations in order to generate initial versions of the next models.

The next models are built using appropriate subsets (and profiles) of UML 2. The first corresponds more to the Analysis Model in the standard OOAD [6] approach. Therefore, we call this model the Analysis Model. In our approach, the main content obtained in this step is an analysis-level class model (the Domain Model). The Analysis Model is described in more detail in Section 5.

The most important model in the proposed model chain is the PIM, which is very close to the corresponding model in the MDA approach. This model is built according to the selected design patterns and contains the description of structure and detailed behaviour of the would-be system in a platform-independent way. Transformations which generate the initial version of this model use both Requirements and Analysis as inputs. Only this way the most sophisticated analysis of requirements can be performed. In the whole chain of transformations, this step contributes most to the rich system functionality inferred directly from requirements. The contents of PIM are described in Section 6.

It should be noted that for our models we use a pre-selected consistent set of design patterns and other design rules called an architecture style in our approach (this concept is described in Sub-section 3.2). Transformations are adjusted to this style to get maximum results in extracting the required behaviour from RSL. The best results are obtained if requirements are specified in RSL in an appropriate way – the RSL profile associated with the architecture style is used (see Sub-section 4.2).

The next model is the Platform Specific Model (PSM) in a fairly standard MDA style (Section 7). It is built by transformations from PIM by adding platform-relevant details. The paper demonstrates the combination of Java and Spring/Hibernate frameworks as the target platform, but any similar platform can be used as well. Finally, PSM is transformed to code (annotated Java EE in this case). The main value of the approach is that a large fraction of a non-trivial prototype of the system can be obtained from requirements without manual extension of intermediate models. Certainly, a true model-driven development should follow, where in each step the required details of the real system are filled in manually.

All model-to-model transformations in our approach are implemented in the model transformation language MOLA [7], which appears to be very appropriate for the given kind of tasks. If selection of patterns and the architecture style are changed, the transformations should be rebuilt too. The emphasis on transformation readability in MOLA would significantly facilitate this task. Another issue to be solved by transformations is the inevitable modifications of models and the necessity to reapply the transformations and merge the results. Transformation development is discussed in Section 8.

2 Related Work

The MDA Guide [4] states clearly that CIM means requirements for the system, although no formalism is proposed for this model. There also is an alternative view [8]

that CIM is just a business model of the whole business environment of the system; in this case, no transformations to PIM are possible. Because of that opinion, there are few approaches similar to ours.

Requirements in a controlled natural language, in particular behaviour scenarios, are used as a starting point in [9, 10, 11, 12, 13]. The approach closest to ours is described in [9], which proposes the Natural MDA language for description of behaviour in use cases. This language uses a large set of keywords; therefore, it is much closer to programming languages than RSL, and the transformation-based approach is only partial. The approach described in [10] is based on the Language Extended Lexicon and does not use the behaviour description thoroughly. The approaches proposed in [11, 12, 13] require an initial semi-manual transformation of natural language requirements into a more formal notation, which then can be processed by model transformations. An interesting approach of this kind is proposed in [12], where the initial requirements in a natural language are manually converted into a list of semiformal functional features, which then can be transformed formally using the topological functioning model. In [11] a manual XML-based initial marking of requirements is used before a grammar-based processing can be applied. In [13] a manual conversion into behaviour trees is used. Thus, the direct processing of requirements in a controlled natural language by transformations is the innovation offered by our approach.

There is so much work on transforming PIM to PSM that we do not comment on this subject since it is not the main topic of our paper.

3 General Principles of the Proposed Approach

3.1 Models

In this section, we present a short rationale behind our selection of the specific model chain.

Requirements are specified in the requirement specification language RSL [1, 2] which is developed as part of the ReDSeeDS project [3] and is the basis for the approach. We are interested mainly in requirements for the system behaviour specified by use case scenarios and draft domain concepts (which are called notions in RSL).

Starting from requirements, a chain of models for a model-driven development of the software system is proposed. To a great degree, this chain is inspired by the classical MDA approach. However, the specific structure and construction principles of models in our approach are determined by the chosen architecture style which most importantly includes the set of selected design patterns. A more precise description of the concept of the architecture style is given in Sub-section 3.2. All the models are built in UML using an appropriate profile.

Initially the Analysis Model is extracted by transformations from requirements. This model has no direct counterpart in the classical MDA chain. In the Analysis Model the most important part is a class diagram describing the main concepts of the software system to be created. Stereotypes are used to distinguish different types of concepts according to the Analysis Profile.

The next model in this chain is the PIM model. In this model, the implementation structure is represented according to the behaviour extracted from use case scenarios. This model is platform-independent and could be used as a basis for development of a code on any enterprise platform (Enterprise Java, .NET, etc). This is the model where the selected design patterns and sophisticated analysis of requirements permit to generate a non-trivial part of solution behaviour.

The final model in the chain is PSM. From this model code fragments for the selected platform can be generated. Currently the chosen platform is Java in the Spring/ Hibernate framework. In this model stereotypes corresponding to Spring-specific annotations are used. Data from this model are transformed to Java code with Spring/ Hibernate annotations.

It should be noted that in ReDSeeDS project an alternative model naming is used – PIM is also called the Architecture Model and PSM the Detailed Design Model.

3.2 Design Patterns and the Architecture Style

Nowadays, as a rule, large enterprise systems are developed using a set of design patterns. There are two types of design patterns: platform-independent and platform-specific. The traditional GoF design patterns [14] represent the former type. The modern Java EE environments (based on the POJO [15] idea and declarative ORM) also share a large set of common enterprise patterns (and so do the latest .NET environments based on POCO [16]). On the other hand, low level patterns such as an adequate usage of Spring framework annotations are still platform-specific.

Usage of design patterns is vital to efficient application of MDD and transformations. However, patterns alone are not sufficient for deciding how the generated models look like. Therefore, we use the concept of *architecture style*, which includes the structure of the system and model, a related set of design patterns (with indications where they should be used), the applied general design principles, and finally, the rules by which model elements are obtained from models preceding in the development chain. This last feature is formalized by a model transformation set associated with the architecture style. The most important content of an architecture style is the selected set of design patterns, tied up to the chosen model structure. Namely patterns are the style element which helps most in specifying efficient transformation rules. In addition, for transformations supporting the given architecture style to produce maximum results, the requirements must be specified in an appropriate style too; therefore, the concept of RSL profile (associated with the given architecture style) is introduced.

In this paper, we propose an architecture style named *Keyword-Based Style*. The main goal of this style is to extract as much as possible behaviour from the requirements. The in-depth analysis of requirements is based on keywords to be found in RSL sentences which the style is named after. The RSL profile associated with the Keyword-Based Style is described in Sub-section 4.2.

We start the description of the Keyword-Based Style with the model and system structure and some general design rules. We have chosen four-layer architecture because it is the most popular and accepted information system architecture style today. We use the following layers: Data Access or Repository layer, Service or Business layer, Application Logic, and User Interface. We also have domain objects as data containers (available to any layer, former DTOs [17]). Another general principle is that our approach

is based on a declarative object-relational mapping (ORM). The particular ORM in our approach will be Hibernate [18]. Whenever possible, we use an interface-based design style for all layers, meaning there is an interface (where the operations are specified) and its implementation class. The selected design patterns for the style will be described in the next sub-section.

It should be noted that there is already an architecture style defined in the ReDSeeDS project from the very beginning named the *Basic Style*. The goal of the Basic Style is to prove the feasibility of the approach in which model-driven development starting from requirements is combined with requirement-based reuse of software. The initial version of ReDSeeDS tool support is also based on this style. However, the possibilities to extract behaviour from requirements in the Basic Style are significantly weaker than in the proposed Keyword-Based Style.

In no case the selected architecture style should be considered the sole possible solution; other styles are also possible. To a great degree, the choice of the most appropriate architecture style depends on the domain of the system to be created. The proposed Keyword-Based Style could be an adequate solution for simple web-based information systems. The selection of architecture style could be formalized on the basis of non-functional requirements for the system; however, this topic is completely out of the scope of this paper. Furthermore, it should be reminded that creation of a new architecture style also requires creation of an appropriate transformation set.

3.3 Selected Design Patterns for the Keyword-Based Style

In this sub-section, we will describe the design patterns chosen for the Keyword-Based Architecture Style. The patterns are grouped according to models and system layers chosen for the style. The patterns used at the PIM level are as much platform independent as possible. Since we have chosen Java + Spring + Hibernate framework as the target platform, the design patterns popular in the Spring community are used at the platform-specific level. This choice has also slightly influenced our PIM level, when we had to choose one of several equivalent options.

We use the DAO design pattern [19] at the Data Access layer. Data access objects are introduced as the main actors for explicit ORM-related actions. Therefore, each DAO has the basic CRUD and typical Find operations. A data access object is created for each persistent domain concept. DAO classes are assumed to have the standard transaction support for their operations.

For business logic, the main design pattern used is Manager (see [20] for its version in the .NET world). It means that for each domain concept participating in business logic, a class (and interface) is created, which encapsulates all business level operations related to this concept.

The application logic and user interface layers are governed by the MVC pattern, which is used in almost every four-layer architecture. In addition, for application logic, the façade pattern [14, 17] is used. For each Use Case in requirements, we create one application logic interface and an implementing class. This class implements all operations invoked by MVC controllers within this use case.

The UI part is kept as simple as possible. It contains only calls to the application layer. This research does not include the specific issues of building user interfaces

from requirements, which is a separate topic in the ReDSeeDS project (currently in development [21]).

We also use the domain object design pattern. It means we use domain objects as data containers, in other words, as standard "POJO" (not mandatory Java) objects. Persistent domain objects are treated as the basis for ORM definition; therefore, platform-independent ORM features such as identifying attributes and persistent relations are included.

The design in general relies on the Dependency Injection Pattern (which will appear later as platform-specific dependency annotations) for referencing other classes; therefore, the Factory Pattern is not used explicitly.

Platform-specific design patterns are used in PSM and in the code. It is domain objects that have the most of platform-specific features. The POJO pattern is used, adapted to the Spring style. We use the declarative ORM definition (Spring + Hibernate) based on annotations. Annotations are coded as appropriate stereotypes in PSM. The transactionality of relevant classes is also defined by annotations. For reference initialization, the dependency injection pattern is used.

For UI layer, the MVC design pattern is used in a standard ("Spring-Basic") way.

4 The Requirements Model

The development of a software system in ReDSeeDS starts with definition of requirements for it in the Requirements Model.

4.1 Requirements Specification Language in ReDSeeDS

The Requirements Specification Language (RSL) [1, 2] is a semiformal language for specifying requirements for a software system. We briefly sketch here those elements of RSL which can be directly transformed into the system design.

RSL employs use cases for defining precise requirements for the system behavior. Each use case is detailed by one or more scenarios, which in turn consist of special controlled natural language sentences. The main type of sentences is the SVO(O) sentence [2], which consists of a subject, verb, and direct object (optionally, also an indirect object). These sentences express the actions to be performed in the scenario. In addition to SVO(O), there can also be conditions, rejoin sentences ("gotos" to a point in the same or another scenario) and invoke sentences (invoke another use case). Alternatively, the set of scenarios for a use case can be visualized in a natural way as a profile of a UML activity diagram. SVO(O) sentences serve as the nodes of the diagram, and conditions and rejoins as control flows (in addition to the natural "next sentence" control flow).

Another part of RSL is domain definition which consists of actors (system users), system elements, and notions. Notions correspond to elements (classes) of the conceptual model of the future system. It is also possible to define notion generalization and simple associations between notions.

The precise syntax of RSL is defined by means of a metamodel [1]. The behavior and domain parts in a valid RSL requirements model must be strictly related. The subject of an SVO(O) sentence must be an actor or system element. An object (direct or indirect)

must be a notion. The informal meaning of each noun and verb must be defined in a vocabulary (currently, WordNet [22]).

For the first version of RSL, a ReDSeeDS tool support [23] has been built including an RSL editor. This tool version only supported the Basic Architecture Style, with an appropriate set of model transformations for generation of PIM and PSM from requirements included. The Enterprise Architect (EA) tool [24] in ReDSeeDS is used for UML support. At present the final version of ReDSeeDS tool support has been built [25]. It includes the basic support for an extended RSL version, the main new feature being the introduction of keywords in order to support the Keyword-Based Style as well. Another aspect is the extension of the domain part – adding attributes to notions and extending association features so that a fully-fledged class model can be obtained. This paper is based on the extended RSL version – see an RSL example in Fig. 1.

4.2 The RSL Profile

Transformations described in this paper can be applied to any valid set of requirements in RSL for a system. However, in order to ensure that these transformations generate a really substantial fragment of the software system to be built, some more constraints on the requirements should be put. Thus, a concept of the RSL profile is introduced. The profile defines the set of keywords with predefined semantics to be used in scenario sentences (verbs, nouns, and prepositions) and some rules on how these keywords should be used. In addition, there are constraints on the order of these sentences (or nodes in the activity form). All these rules are "soft" rules in the sense that requirements do not become invalid if they violate some of these rules; simply, the transformations can do less. At the same time, profiles are defined so that they never make requirements less readable to domain area specialists (however, more skills may be required by requirement engineers to create them). A profile is always associated with an architecture style so that the corresponding transformation set can produce the largest possible part of the PIM and PSM models from requirements.

In fact, the default Basic Architecture Style together with the default RSL profile and the corresponding transformation set has already been used in ReDSeeDS [23]. This profile has no keywords, only some constraints on sentences. The usage has confirmed the feasibility of the used technologies; however, the part of a system generated by transformations is small.

In this paper, we propose a profile for the Keyword-Based Style. In this profile, the verb keywords for SVO(O) sentences are *show*, *select*, *build*, *add*, and *remove*. The noun keywords are *form* and *list* – when used as parts of complex notion names (and, consequently, objects in SVO(O) as well). Conditions (which otherwise are arbitrary sentences in RSL) can contain the verb keyword *click* and noun keywords *button* and *link*. The adjective (modifier in RSL terms) *empty* is also treated as a keyword.

Now we briefly describe the meaning of keywords and some context rules in scenarios. The keyword *show* means that the system must display a form defined by the direct object of this sentence. This object, in turn, must correspond to a notion whose complex name ends with the noun keyword *form*. For example, the SVO(O) sentence "System *shows* reservable facility list *form*" specifies that the form "reservable facility list *form*" must be displayed at this point.

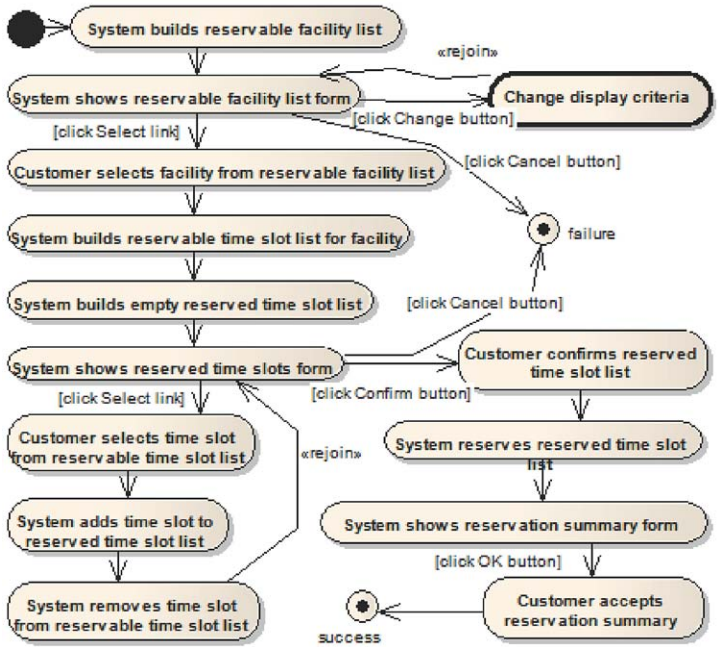

Fig. 1. Requirements – scenarios of the use case in a graphical form

Similarly, the sentence "System *builds* reservable time slot *list* for facility" uses the verb *build*, which means a data creation. The direct object "reservable time slot *list*" denotes a list, since the last noun in it is *list*.

The sentence "Customer *selects* facility from reservable facility *list*" means that the user has performed element selection from the data table in the form. The indirect object (after the preposition "from") specifies the data table contents ("reservable facility *list*", that is, a *list* notion), the selected element is an instance of the notion "facility".

The condition "*click* Select *link*" means that the user clicks on an active element (link) in a form table with selectable rows. Normally this condition should be on the control flow, which goes from the *shows* sentence/node (see the example above) to the *selects* sentence (the previous example). The condition "*click* Confirm *button*" means that the form button has been clicked.

The meaning of the remaining keywords is self-explanatory. The example in Fig. 1 completely complies with the rules described above.

The described profile for the Keyword-Based Style is supported in the current version of ReDSeeDS tools.

4.3 An Example of Requirements

The proposed ideas are illustrated on a fragment of an example of the Fitness Club system. One use case *Reservations* is taken – how a club customer can book regular access to the selected fitness facility of the club. The scenarios for this use case (in the form of one activity diagram) are defined in RSL (see Fig. 1). For this to be a correct requirements model, the relevant notions must also be defined (facility, reservable

facility list, etc). Fig. 2 presents two scenarios of this use case in textual form as they were entered using the RSL editor.

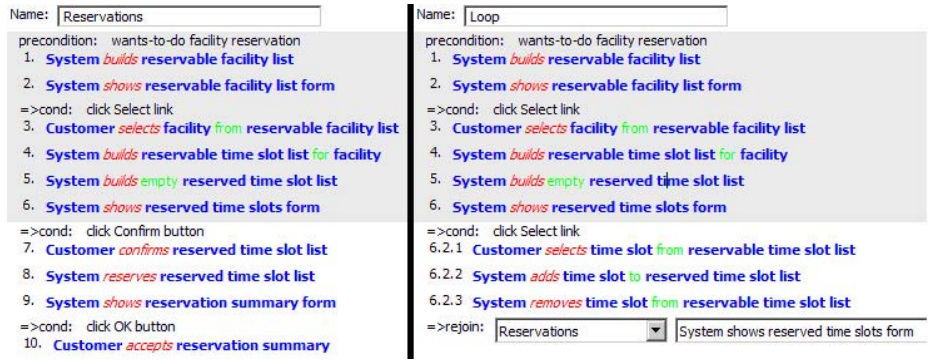

Fig. 2. Requirements – two scenarios in a textual form

The colour marking helps to distinguish more clearly the parts of SVO(O) sentences – subjects, verbs, and objects. Prepositions starting an indirect object are marked green. The whole continuous group of words marked blue is an object with a complex name (there must be an equally named notion in the domain part of the requirements). Note that in the textual syntax, each scenario is one continuous path in the diagram.

5 The Analysis Model

5.1 The Structure of the Analysis Model

The main part of the Analysis Model in the Keyword-Based Style is the Domain Model – a conceptual class model for the system to be built. The Domain Model is generated by appropriate transformations from the domain (notion) part of Requirements. It contains classes corresponding to all notions in Requirements. Class attributes and associations are also extracted from the notions part of Requirements (if they have been defined there). A special Analysis Profile is defined in ReDSeeDS which contains stereotypes to be applied to the Domain Model. Classes generated from persistent notions would have the <<entity>> stereotype (there also are some heuristic rules how to find persistent notions when they have not been properly marked in requirements). Other classes with the stereotype <<form>> would correspond to forms – notions with the suffix form in their names. In a similar way, collection classes (for example, ReservableFacilityList) will have the <<list>> stereotype. In the design stage, these classes will be converted into generic list classes. Control elements in forms (such as buttons and links) are also represented by stereotyped classes in the Domain Model, with stereotypes <<button>>, <<gridLink>>, <<link>>, and some others. Additional associations having a special meaning for the design model (e.g. aggregations linking a form to a list to be visualised as a data grid in this form) can also be generated. These associations are also given special stereotypes (<<owned>>, <<formElement>>, and some others). See more on the principles how the Domain Model is generated from Requirements by transformations

in Sub-section 5.2. Fig. 3 presents part of the generated Domain Model in the Fitness club example. It shows that the proposed approach can transfer a significant part of the intended semantics of requirements into the stereotyped Domain Model (this, in turn, will guarantee a rich behaviour to be generated into the PIM model).

The full strength of the transformations is revealed only if requirements are built in RSL according to the appropriate RSL profile (see 4.2). If requirements in RSL cannot provide sufficient information for building this Domain Model, it is highly recommended to extend this model manually in the Analysis Step. Only in this case the next steps will provide the desired results.

The structuring of the Domain Model is based on notion packaging (provided in RSL).

5.2 Transformation of Requirements to Analysis

The main task of this transformation is to create the Domain Model from the notion part of Requirements, taking into account some elements of scenarios as well. The basic transformation is very straightforward since notions, their attributes, and relationships in RSL actually are in one-to-one correspondence to the class model. The stereotypes <<*list*>> and <<*form*>> are added if the respective keywords are present in the notion names. An additional analysis is done for list classes. If an entity name is contained within the list notion name (such as "facility" within "reservable facility list"), the entity class is assumed to be the element of that list (a <<*listItems*>> association is generated).

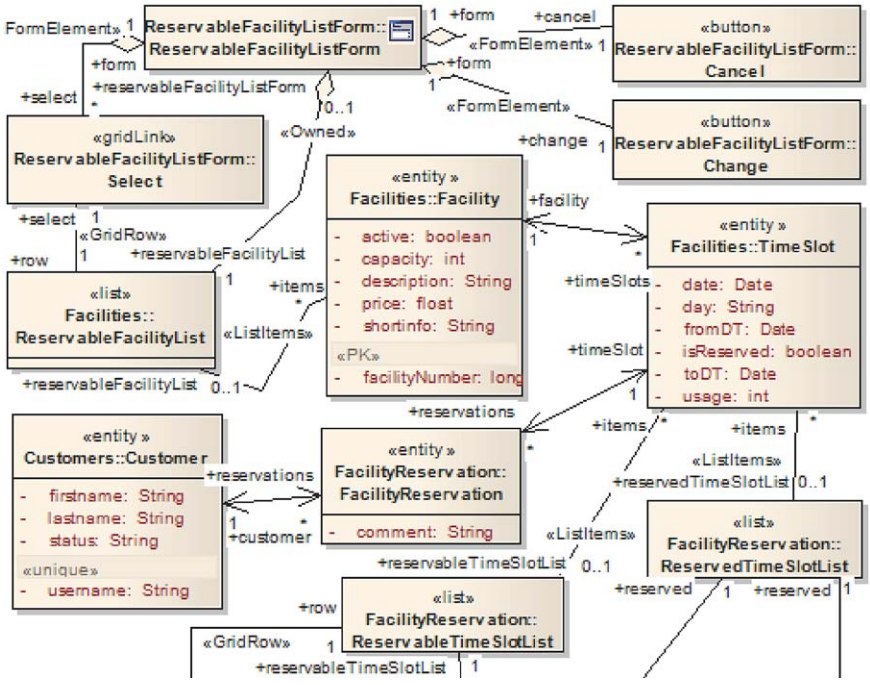

Fig. 3. Fragment of the generated Domain Model

Classes for control elements can be generated from scenarios. We are looking for a *click*-condition (*click … link* or *click … button*) which follows a *show*-sentence (… *shows … form*). If such (new) situation is found, a class is generated with the name equal to the name in the *click*-condition and the stereotype <<*gridLink*>> or <<*button*>>, respectively. The association (with the stereotype <<*formElement*>>) linking the control element to its form is also generated.

More form-related associations can be generated from scenarios. *Select*-sentences (such as ... *selects* facility from reservable facility *list*) let us conclude that the relevant form (that in the preceding *show*-sentence) permits to select elements exactly from this kind of list. Hence, this list (here, ReservableFacilityList) is visualized in the form (the <<*owned*>> association can be built), and each gridLink element in the form corresponds to a row in the list (the <<*gridRow*>> association is built).

Using these relatively simple principles, the domain model in the example in Fig. 3 can be generated from notions and the scenario in Fig. 1. The implementation of these transformations in the MOLA language is also quite straightforward.

6 The Platform-Independent Model

This model is the most important to our approach since all platform-independent functionality is generated in this model. This is done by revisiting the use case scenarios and analyzing them repeatedly, taking into account the (possibly manually extended) Domain Model from Analysis. In combination with the keyword-based sentence analysis, a significant part of application and especially business logic can be generated. This model is created according to the platform-independent design patterns described in Sub-section 3.3.

6.1 The Structure of the Platform-Independent Model

The main result of the PIM step is the design class model: packages and classes (and interfaces) with all attributes and operations. The operations will have all parameters defined. All the other data such as persistence info for ORM-related classes are coded by platform-independent stereotypes, which constitute the PIM profile.

The other essential results of this analysis are stored as sequence diagrams, also covering a significant part of business logic method bodies. All method invocations with appropriate parameters which can be generated are coded this way. Whenever possible, invocation logic up to the DAO level is documented. These sequence diagrams are kept in the *behaviour* package and are grouped in the same way as Use Cases in the Requirements Model. Some small practical extensions of sequence diagram syntax are used, for example, *FOREACH* iterator in *loop* fragments.

The design class model is split into the following packages: *applicationlogic*, *businesslogic*, *dataaccess*, *domainobjects*. The first three are further subdivided into *Interfaces* and *Implementation* parts containing interfaces and implementing classes, respectively. Each interface name has the prefix “I” added to the corresponding class name.

For application logic, the façade design pattern is used. For each use case, a class corresponding to this use case is generated (with the the suffix “Service” added to the

name). Further structuring of the *applicationlogic* package is done according to use case packages.

The content of *businesslogic* is generated according to the Manager Pattern. Here classes correspond to persistent classes (entities) whose usage in business logic can be inferred from sentences with keywords and the domain model. Classes/interfaces have the suffix "Service" added to the entity name.

For *dataaccess*, an updated version of the DAO pattern is used, and practically applicable methods are generated for DAO classes. Each class corresponds to a persistent domain object; the class name is generated from the object name with suffix "DAO". Classes are grouped in the same way as domain objects. For each class, CRUD and some typical find operations are generated. Bodies of these operations are similar in all classes, only types vary. Therefore, we propose to implement them once in a template class which contains parameterized types. All the other classes will inherit them from this template class (with parameters set to the relevant values in each case). We remind that this specialization of the classical DAO pattern is platform-independent since it can be directly implemented in most of typical platforms.

For the *domainobjects* package, the domain object design pattern is used. This package represents a platform-independent ORM (Object Relational Mapping) model for all entities, with platform-independent annotations. Associations (relations) are also included in a way typical of an ORM definition. A database schema for a specific platform can also be easily generated from this model (in the next PSM step). Names of domain objects are taken from the corresponding domain concepts. For each persistent class, a unique identifier attribute is defined as well.

6.2 Transformation of Requirements and Analysis to PIM

Transformations for building the platform-independent model are more complicated than for building the Domain Model in Analysis. They use the behavior part of the Requirements Model as input, as well as the updated Domain Model.

The transformation of domain objects is very straightforward. Domain classes are transformed to PIM domain objects, keeping all attributes. For each persistent class without primary key, an artificial primary key is created.

For each persistent domain class, a DAO class and its interface is created in the *dataaccess* package. They specialize the template-based implementation of CRUD and filter operations.

In the Business Logic layer, classes and interfaces have a similar structure as in DAO, with the exception that classes not having business level methods are excluded. The generation of business methods is done in the general context of behavior generation by analyzing scenarios in requirements.

In the Application Logic layer, for each use case, a class and interface is generated. For this interface/class, one "main" method is generated (which means invoking this use case from another one). Its name corresponds to the Use Case name. Other methods for this class are generated for UI-related sentences in the scenario. UI-related sentences are detected by analyzing the subject of the sentence. If the subject of the sentence is an actor, then it is Actor-system sentence (or UI-related sentence).

Now we will describe behavior generation. Behavior is grouped in the same way as Use Cases. For one Use Case, one or more sequence diagrams are generated by

processing its scenario. The behavior of a Use Case begins with invocation of the "main" method of the application logic class corresponding to the Use Case. In order to build an application logic method body, we look for consecutive scenario sentences with the subject System and recipient system (in other words, any verb other than "System *shows* …). All these sentences correspond to calls to the Business Logic layer. At first the verb used in this sentence is analyzed. If the verb is a keyword, the sentence is analyzed according to rules used for this keyword. If the verb used is not a keyword, the structure of the sentence is analyzed and object keywords are analyzed. Default behavior generation principles corresponding to the sentence structure are applied. The immediate recipient of this call depends on the sentence structure. If the indirect object (e.g., …*for* facility) is present, the call is directed to the manager of the corresponding entity (here, *FacilityService*). Another typical case is when an indirect object is absent and a direct object corresponds to a notion/class with the stereotype <<*list*>>. Then the invocation is created to the manager class corresponding to the entity class which is the list element. There also are some other "patterns" of sentences which correspond to business logic calls (or simple actions directly in the application layer). The grouping of the generated business logic calls is done in a simple way – all these calls up to the next UI call (corresponding to the next "System *shows* …" sentence) are included in the body of the current application logic method body (see Fig. 4). The "System *shows* … *form*" sentence generates a call to the user interface layer (to the controller of the relevant form), which completes the current body. The next sentence (which in fact follows the "*click* …" condition) corresponds to the invocation of another application logic method. Then building of the body of this method starts.

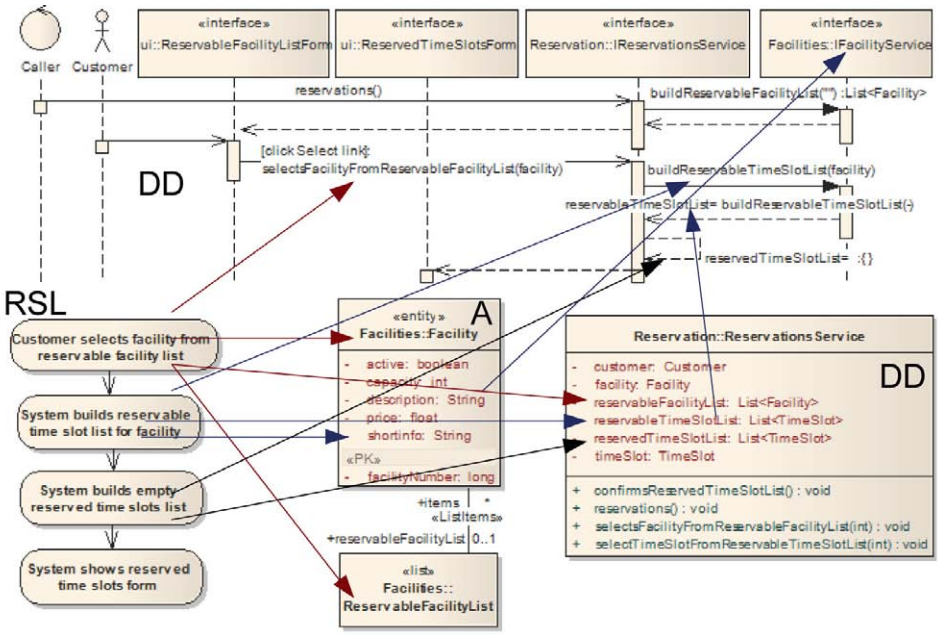

Fig. 4. An example of informal mapping describing transformations to Detailed Design

Fig. 4 illustrates in detail a typical application of the transformation rules described above by an informal "model mapping diagram", with arrows going from source model instances (bottom) to the corresponding target model instances (top). The first sentence in the scenario fragment ("Customer selects facility from reservable facility list") follows the "click Select link" condition; therefore, it implies the method invocation *selectFacili tyFromReservableFacilityList*() to the application logic class (*ReservationsService*). The next two sentences in the scenario correspond to the actions in the body of this application logic method. Fig. 4 shows detailed analysis of the first sentence. The sentence "System builds reservable time slot list for facility" implies the business logic method invocation *buildReservableTimeSlotList*(). According to the rules described above, there is an indirect object ("for facility"); therefore, the method must go to the corresponding manager class (to the class *FacilityService*). Because of *build*-semantics (*build* is a keyword) of the verb and *list*-semantics of the direct object, the return type of the method is *List<TimeSlot>*. The returned value must be stored in the attribute *reservableTimeSlotList* (of the same list type) of the invoking application class (*ReservationsService*). The next sentence corresponds to an action in the body (assignment to the attribute *reservedTimeSlotList*) because of the semantics of the keyword *empty*. Note that all lifelines correspond to interfaces because any invocation goes via the corresponding interface in our style (certainly, body behavior relates to the relevant class).

There are some more rules in the approach quite similar to those explained in the example. We do not examine the interaction with the UI layer in more detail.

7 The Platform-Specific Model and Code

This model is a specialisation of the platform-independent model to a specific platform. Java with Spring + Hibernate 3 was chosen, with declarative (annotation-based) style as much as possible.

7.1 The Platform-Specific Model

For this platform, the model is quite similar to the platform-independent model. The class structure in the PIM more or less corresponds to the required structure in PSM. The main task is to convert annotations to the specific style required by Spring and Hibernate. However, some new model elements should be added as well.

A new model is the database diagram generated from the domain objects. This is a typical database design diagram (with tables, columns, PK, FK, etc) in EA.

Domain objects themselves are "copied" with the same package structure. They are used to describe Hibernate-specific ORM functionality. All Hibernate- and Spring-specific annotations are added (coded as stereotypes) to domain classes, attributes, and operations. The relevant getters/setters and some predefined methods are added to classes. Traceability links between PIM and PSM elements are generated by transformations and used to maintain various annotations related to mappings between different parts of the model.

For each DAO class, the annotation <<*@Repository*>>is added. These classes also have annotations describing the transactional mode, by default "required" is used. The template-based mechanism is directly taken from PIM.

Application logic layer classes are included in the Business Logic layer. Classes in these layers are given the annotation <<*@Service*>> (to mark them as Spring beans). The annotation <<*@Autowired*>> is used to initialize references to other beans.

The structure of PSM corresponds directly to the potential Java class structure typically used in Spring (with packages *domain*, *repository*, *service*). These packages are further structured in accordance with the already defined model structuring.

In order to have a more or less complete design class structure and behaviour in sequence diagrams, some elements in the UI area also have to be specified. The basic source for that – forms, attached data, and actions (buttons and links) are available in the Analysis Model. Currently a rudimentary solution directly based on Spring MVC is proposed. In this solution, we can use JSP for data visualisation and controllers to manage user actions. We use one controller per form, with a method for each user action in the form. Typically a controller method directly calls the appropriate application logic method. Nevertheless, this should be treated only as a "stub" which can be replaced by a more appropriate UI feature definition. Such a prototype form structure definition could be incorporated in requirements since RSL language contains features for that purpose. Currently some experiments in this direction have been performed.

Sequence diagrams defining behaviour within method bodies are also refined according to Spring requirements. The most significant changes refer to the user interface part. At this level, a simple version of UI and application logic interaction can be precisely defined. In particular, a special "executable" solution (including DAO methods) could be provided for finding the object selected by the user via a data grid in a form. This way, the form behaviour sufficient for simple prototyping could be provided. We do not describe the UI aspects of PSM in more detail since tool support for them has not been fully implemented.

7.2 The Java Code

The provided PSM can be used for Java code generation. This generation is quite straightforward – at first all information must be transferred into a properly stereotyped class model using MOLA transformations (the body behaviour must also be transferred from sequence diagrams to code sections of operations in EA). Then properly modified EA Java code generation scripts can be used. The main issue of modification is to add scripts for processing all relevant annotations.

The structure of the Java code directly corresponds to the structure of PSM. Methods are generated according to the model. For some methods, predefined method bodies are generated. This is widely used for domain objects (almost all methods are generated). In particular, bodies of getters, setters, *hashCode*, *equals*, *toString* are generated. A template-based generator is used and the method body vary according to object properties for which the method is generated.

Predefined method bodies of the TemplateDAO class are also generated. Concrete DAO classes extending the TemplateDAO class with appropriate types are also generated. Appropriate Hibernate configuration file describing, for example, data base connection is also necessary. An initial version of this file can be generated. It should be noted that a data base script can also be generated from PSM.

Business logic- and application logic-related functionality is generated according to the class structure. The behaviour (described in sequence diagrams) is also generated.

For the UI part, currently only a placeholder is generated (due to reasons explained in Sub-section 8.1).

The generated Java project can be inserted into an Eclipse IDE project template containing references to the required Spring and Hibernate libraries. Thus, a ready-to-compile project is obtained. All this constitutes a significant part of a simple prototype – mainly the UI part has to be added manually. However, if the complete set of transformations described here was implemented, a "near to executable" prototype would be obtained.

Here are some examples of the generated Java code. The first example shows part of the code generated for the *Facility* entity.

```
@Entity
@Table(name="facility")
public class Facility {

private Boolean active;
private Boolean capacity;
private String description;
private String facilityNumber;
private String id;

@Override
public boolean equals(Object obj){
if (this == obj) return true;
if (!super.equals(obj)) return false;
if (getClass() != obj.getClass()) return false;
Facility other = (Facility) obj;
if (active == null) {
  if (other.active != null) return false;
} else if (!active.equals(other.active)) return false;
if (capacity == null) {
  if (other.capacity != null) return false;
} else if (!capacity.equals(other.capacity)) return false;
if (dscription == null) {
  if (other.description != null) return false;
} else if (!description.equals(other.description)) return false;
if (facilityNumber == null) {
  if (other.facilityNumber != null) return false;
} else if (!facilityNumber.equals(other.facilityNumber)) return
false;
 return true;
 }

@Column(name = "active", nullable = false)
public Boolean get_Active(){
return active;
}

...

public void set_Active(Boolean p){
active=p;
}

...

}
```

The next code fragment shows the code generated for Application logic methods. These are three methods for the Application logic class *ReservationsService*. To understand the context, one sequence diagram from the PSM model is shown in Fig. 5. There are three method invocations on the *ReservationsService* lifeline (*reservations*, *selectsFacilityFromReservableFacilityList*, and *selectsTimeSlotFromReservableTimeSlotList*). Methods invoked within the corresponding fragments of the lifeline (until the return) appear within the corresponding body.

A code fragment for *ReservationsService* class.

```
@Service("ReservationsService")
public class ReservationsService implements IReservationsService {

@Autowired
private IChangeDisplayCriteriaService iChangeDisplayCriteriaService_;
@Autowired
private IFacilityService iFacilityService_;
@Autowired
private IReservedTimeSlotListService iReservedTimeSlotListService_;
private List<Facility> reservableFacilityList;
private List<TimeSlot> reservableTimeSlotList;
private List<TimeSlot> reservedTimeSlotList;

...

public void reservations(){
reservableFacilityList=iFacilityService_.
buildsReservableFacilityList();
}

public void selectsFacilityFromReservableFacilityList(Facility
facility){
reservableTimeSlotList=iFacilityService_.buildsReservableTimeSlotLis
tFor(facility);
reservedTimeSlotList= new ArrayList<TimeSlot>();
}

...

public void selectsTimeSlotFromReservableTimeSlotList(TimeSlot
timeslot){
reservedTimeSlotList.add(timeslot);
reservableTimeSlotList.remove(timeslot);
}

}
```

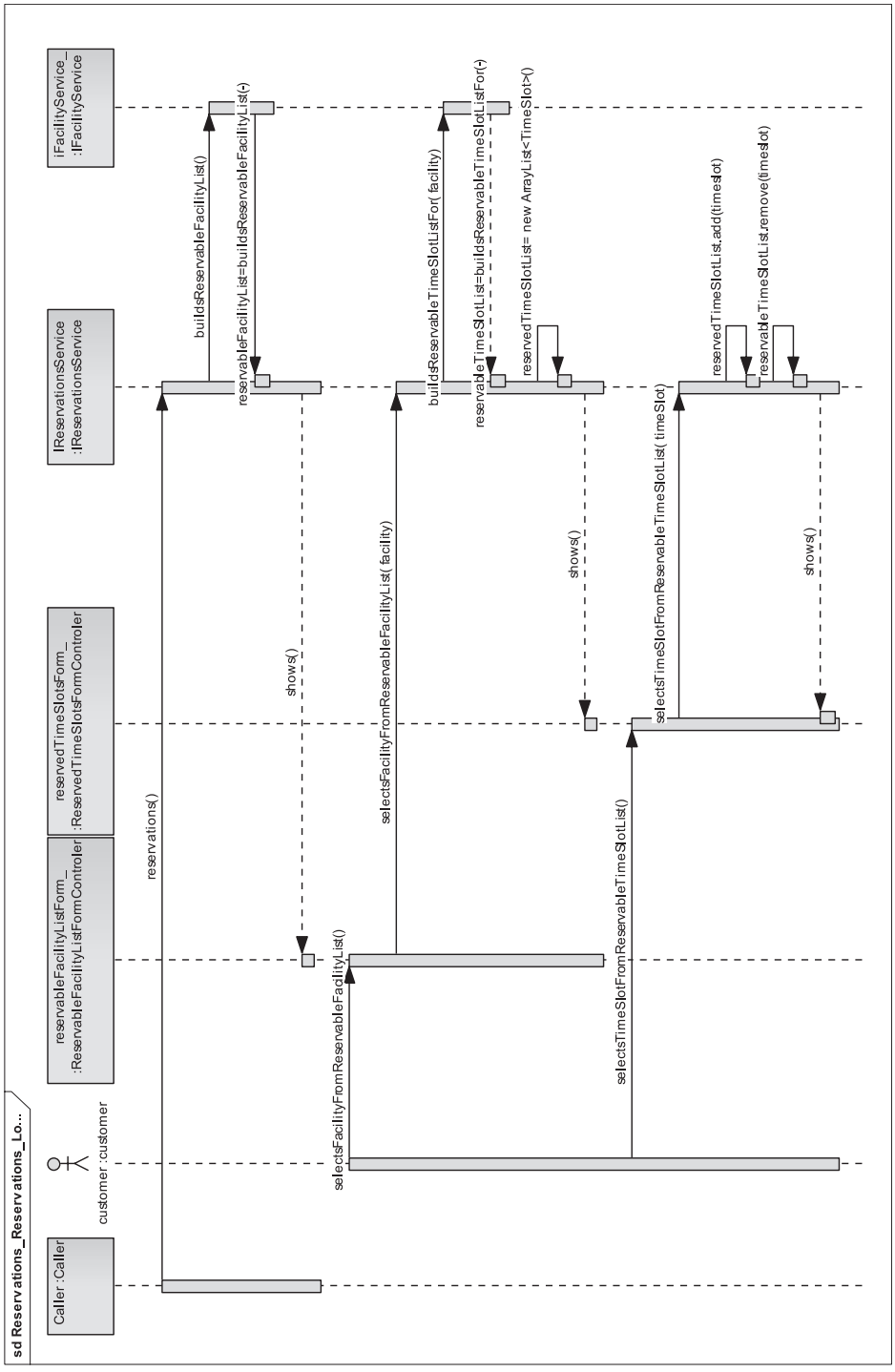

Fig. 5. An example of a sequence diagram for *ReservationsService* class

8 Implementation of Transformations

8.1 Model-to-Model Transformation Implementation

In this section, we briefly describe the implementation of transformation algorithms for building the chain of models in the Keyword-Based Style. The transformation language MOLA [7, 26] is used to define the transformations.

Although the current version of ReDSeeDS tools supports both the Basic Style and the Keyword-Based Style, not all new model transformation features described in previous sections are implemented in this version, mainly the features related to generation of UI functionality. For example, analysis of condition sentences in scenarios is not implemented since in the chosen RSL profile, conditions are mainly related to user interaction with forms. Similarly, the Analysis Model is created using only keyword-based analysis of notions, scenario-based analysis is not implemented in the current version. This again is related to the fact that scenario sentences can mainly contribute to finding relations between forms and their contained controls. The delay of transformation support for UI functionality is due to the fact that it would be natural to combine the generation of UI features from scenarios with direct specification of UI structure in RSL (as is usually done during requirements specification). Although this possibility is in the RSL language, as already said, currently there is minimum tool support for this.

Consequently, the UI part in generated models is implemented minimally; only some basic UI classes and interfaces are created. All the remaining details of UI such as form elements are not generated in the current version. Therefore, code generation for the UI part is not supported either although generation of some code skeletons is technically feasible.

One deviation from clean usage of UML in models is also visible in some of the examples. Assignments in sequence diagrams are emulated by message text and some tagged values because this feature is defined in UML in a very complicated way and supported in virtually no UML tools. This workaround has made some transformations more complicated.

The transformations are implemented using the MOLA tool [26]. Several different kinds of transformations are developed. Firstly, there are basic transformations supporting each software case development step: from RSL to Analysis, from Requirements and Analysis to PIM, from PIM to PSM, and from PSM to the PSM Code Model. There also are some technical transformations: export to EA, import from EA, keyword analysis, RSL scenario visualization by UML activity diagrams, and Simple Merge. Some transformation rules are re-used in several transformations.

The metamodel used for transformations is the same as for other ReDSeeDS tool components – it consists of an RSL metamodel merged with relevant parts of the standard UML metamodel and extended by special traceability elements. Transformations also build the relevant traceability links in every step.

Some non-trivial aspects of transformation implementation are described below.

Transformation for keyword analysis (which is the first to be applied in the chain) scans nouns, verbs, and modifiers used in scenario sentences, and fills in the keyword field of relevant RSL elements. This permits to specify the same keyword with several synonyms. It could be improved further by including Wordnet-based meaning analysis in this transformation.

The next transformation is from RSL to the Analysis Model. The logic of this transformation is relatively simple – it analyses the notion model in RSL and transforms it directly into a UML class diagram, adding stereotypes based on keywords set by the previous transformation.

The most important transformation is from the requirements and analysis model to PIM. This transformation has two logical parts. The first part is the creation of a static structure – package hierarchy, classes, and interfaces. The second part is the creation of behavior stored as UML sequence diagrams.

For creation of a static structure, a universal "package hierarchy copier" is used. This package hierarchy copier receives as input root of the source package hierarchy, the target package, and the copy mode. The package copier copies a hierarchy of packages and their elements (classes, interfaces, etc) in a way specific to the given mode. For example, it is possible to define that for some mode, a suffix should be added to the class name. It is also possible to define that for some mode class attributes should be ignored, etc. The universal package hierarchy copier is used in several contexts during creation of PIM and PSM models. In PIM Data Access objects and Business Logic objects are based on Analysis class diagram. In PIM Data Access class should be created for each persistent class in the Analysis Model. This is ensured using an appropriate copy mode. The same copy package hierarchy mechanism is even more widely used in creation of PSM since it is based on the PIM model with some modifications.

Another important part of PIM is the behavior description using UML sequence diagrams. In this case RSL scenarios are analyzed and sequence diagrams are created. For each scenario, one UML sequence diagram is created. The content of this sequence diagram depends on RSL sentences used in this scenario. Objects generated from a sentence depend on the kind of the sentence. There are three kinds of sentences: an "Actor-System" sentence defines interaction of an actor with the system. It can be recognized by the subject of the sentence – an Actor. The Subject of the two other kinds of sentences must be a system element. The next kind is a "System-Actor" sentence. Such sentence typically means that the system shows something to the user or asks for some input from the user. The third kind is "System-System" sentences. These sentences are used to describe internal actions of the system, typically some business logic. There are different subkinds of these sentences, depending on keywords used in the sentence.

The sequence diagram elements generated from a sentence depend on the kind and subkind of the sentence. At first the subkind of the sentence is determined; then elements of sequence diagrams are created. Since the UML sequence diagram metamodel is quite complicated, procedures for basic element creation are used. The procedure for one subkind of a sentence consists of calls to procedures for creating/finding basic sequence diagram elements. Fig. 6 demonstrates an example of procedure creating sequence diagram elements for "System-System" SVO sentence without keywords. At first the lifeline corresponding to the object is found or created. Then a message to this lifeline is created. Then operation corresponding to this message is found or created. Then this operation is associated with the message created. Then a return message is created. Each of these tasks is implemented as a MOLA procedure invoked by the given procedure. These procedures for sequence diagram element processing are used as building blocks. The content of one such MOLA procedure is shown in Fig. 7, which demonstrates the search of lifeline in a sequence diagram depending on the object used in the verb phrase.

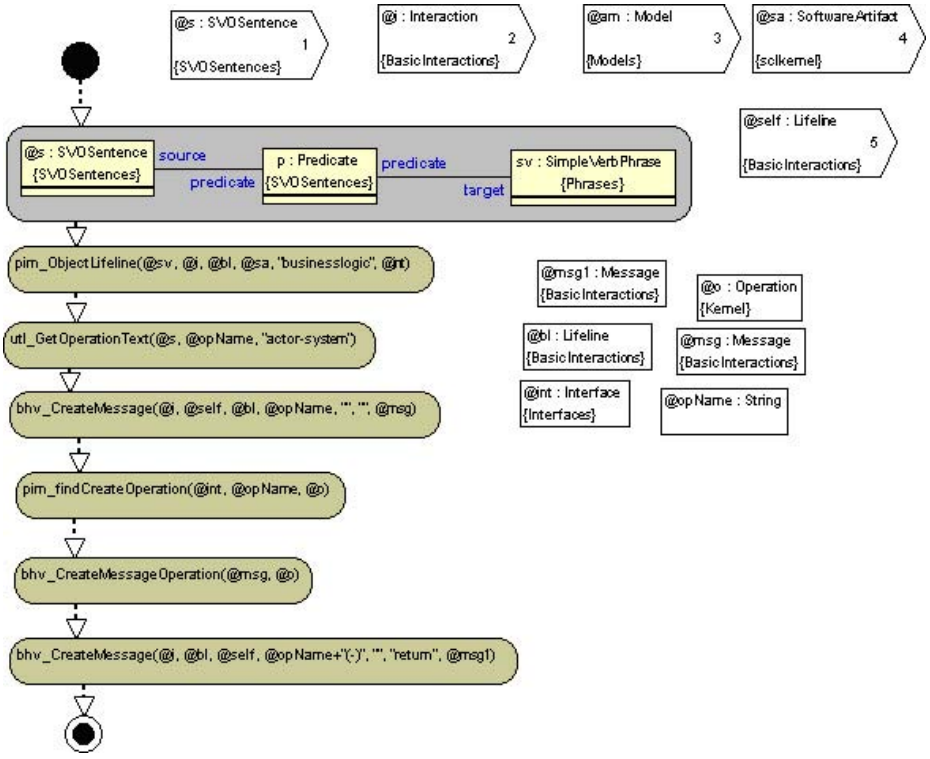

Fig. 6. Creation of a message for a "System-System" sentence without an indirect object

In the first rule, the notion corresponding to the noun used in a verb phrase is found (the long chain of associations necessary to locate this correspondence is implied by the RSL metamodel [1]). Then it is determined whether this notion or its parent should be used. Then the interface corresponding to this notion is found (it has been created during static structure generation). In this case, the Business Logic interface is found. Finally, lifeline for this interface is found or created. This procedure is very typical of transformation implementation in ReDSeeDS – it demonstrates the strength of MOLA patterns in finding complicated correspondences between model elements (such complicated correspondences are enforced by the structure of RSL and UML metamodels).

The next step in the chain is transition from PIM to PSM. For the creation of PSM, the package hierarchy copier described above is widely used. Only appropriate modes are defined.

The transformation from PSM to the initial code analyses sequence diagrams and creates the initial code. The code is attached to each relevant method. All messages from a lifeline starting from a method invocation on the lifeline to the return message (a message describing return to the caller of this message or a message to UI) are transformed to actions in the code for this method. For storing code corresponding to an operation, UML comments are used (the initial code is not a standard UML metamodel element). The transformation for code creation iterates through all messages in the sequence diagram. The search is performed in a recursive way (based on a stack). When

it detects a call of some operation, it means the following messages will constitute the body of this operation. If call to another operation follows this operation, the call to this other operation is added to the code body of this operation and this operation is added to the stack; and the newly created operation is set to be the current. If return from this operation to the previous operation is detected, the previous operation is popped out from the stack. If self messages are detected, an appropriate code is simply added to the message body. The stack is implemented using UML comments since it was not possible to extend the metamodel with temporary classes (due to requirements of other tool components).

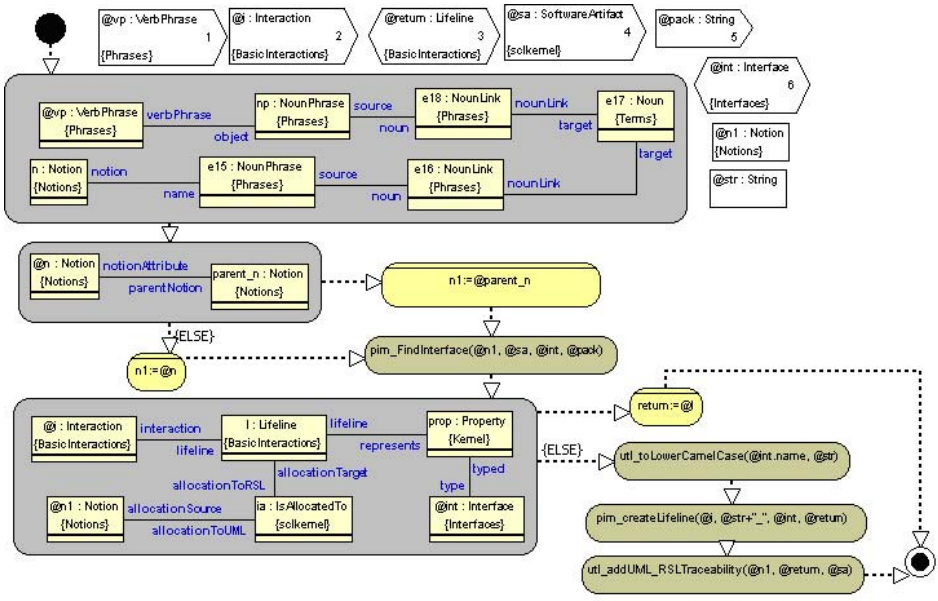

Fig. 7. The procedure of finding a lifeline in a sequence diagram depending on the object used in the verb phrase

Implementation of these transformation rules in the Keyword-Based Architecture style took approximately 3 person months. Implementation of these transformation rules consists of about 140 MOLA procedures (of size similar to those shown in Fig. 6 or 7). Thus, MOLA happened to be a very appropriate solution for implementation of transformations of such kind. The very low error rate during the development has to be singled out. Implementation of rules currently missing would be a small part of the existing code.

One more aspect of transformation implementation should be pointed out. All transformations in the chain must support repeated runs – the requirements ever change. What is even more important, for the same transformations to be applicable to manual model-driven development, all models in the chain should allow for manual modification. Therefore, support for various result merge actions must be included in the transformation set. In our approach, this support mainly relies on traceability links.

Currently one kind of the merge procedure – the so-called Simple Merge is implemented, but more sophisticated merge procedures could be implemented too.

8.2 Model-to-Code Transformation Implementation

Many MDD-based tools offer code generation from UML models. The Enterprise Architect (EA), the modelling tool used in the ReDSeeDS project, has the Code Template Framework (CTF) which also provides code generation features. The CTF consists of a number of code generation templates which generate a code for the most popular programming languages like Java, C++, etc. Each template transforms particular aspects of the UML to corresponding parts of the target language. Just like most of code generation tools in the MDD world, the EA does not provide full code generation, but code skeletons (classes, interfaces, field and operation declarations) can be obtained. Only packages, classes, and interfaces are used by these templates, other UML elements are ignored. These templates are called base templates. The latest versions of EA (not used in the project) provide some code generation features for behavioural UML diagrams as well (sequence, state).

Since the ReDSeeDS project uses the EA for UML support, there is a possibility to re-use all CTF capabilities of code generation. It is a significantly easier way to obtain a code than to generate a Java model as the first step and then convert this model to a proper code.

Base templates can be used directly for the default architecture style. These templates are applied to a detailed design model of this architecture style. The package hierarchy, declarations of all classes (DAO, DTO, etc), and methods are included in the generated code. Bodies of obtained methods should be filled in manually since the detailed design model in this style contains no behaviour.

For the Keyword-Based Architecture Style, significantly more code can be generated, including the behaviour aspects. Base templates do not generate the declarative annotations used in the Keyword-Based Architecture Style. We remind that these annotations are specified in the platform-specific model as appropriate stereotypes of classes, attributes, and associations. However, code generation templates are defined using the model-to-text language (the CTF language) in EA. Thus, it is possible to customize the way in which CTF generates a source code. The extension of the Java code generation template for Spring framework has been built. The generated code contains Spring annotations obtained from the stereotypes.

Although behavioural diagrams cannot be properly used for code generation in EA, they can be processed by model transformations before the code generation step. For example, a MOLA transformation converting a message and action sequence in a sequence diagram into part of the code of the appropriate method body has been implemented using an intermediate model. Then such an enriched intermediate model can be further processed by code generation templates in EA. Since such pre-processing is done, a great portion of the code (for example, method invocations from sequence diagrams) is being generated using EA. This way a meaningful executable prototype code could be obtained directly from requirements. If the models in the software platform-independent and platform-specific models have been extended manually, a true model-driven development can be carried out by this approach.

9 Conclusions

The paper shows the feasibility of a transformation-supported path from semiformal requirements to code in a model-driven way. The key aspects that have enabled this are selection of an appropriate architecture style (the general structure and an appropriate set of design patterns) for the system and an associated style for requirements – a profile for the requirement language. Then a corresponding set of transformations can be defined that can extract maximum facts from requirements and convert them into appropriate elements of models in the development chain. The most crucial of models in the chain is the Platform-Independent Model. To build this model, most sophisticated analysis of requirements has been done. The next model – PSM – is adapted to the selected platform – Java, Spring, and Hibernate. For models in the chain – Analysis, PIM, and PSM – appropriately defined UML profiles are used. The models obtained by this approach serve as the basis for further model-driven development, using the same transformations for support. All the transformations are implemented in the model transformation language MOLA.

We want to conclude with some thoughts on software design languages. Sequence diagrams used in our approach are the most natural UML facility for describing the required behavior. This notation is excellent for describing the way methods of another class are invoked. However, a more detailed data flow description soon becomes very awkward in this notation. An alternative could be the usage of UML activity notation with basic actions included, but then the class interaction is clearly less readable. Thus, the UML possibilities are slightly unclear. It may be that a special DSL for software design should be developed.

Acknowledgments. This work is partially funded by the EU Project "Requirements-Driven Software Development System (ReDSeeDS)" (contract No. IST-2006-33596 under 6FP). The authors would like to thank ReDSeeDS partners for valuable discussions. Special thanks to the ReDSeeDS partners from Warsaw University of Technology. The authors would also like to thank Oskars Vilitis for Spring-related consulting.

References

1. H. Kaindl, M. Smiałek, D. Svetinovic et al. Requirements specification language definition. Project Deliverable D2.4.1, ReDSeeDS Project, 2007. Available: www.redseeds.eu (http://redseeds.iem.pw.edu.pl/index.php?option=com_remository&Itemid=7&func=fileinfo&id=58).
2. M. Smialek, J. Bojarski, W. Nowakowski et al. Complementary use case scenario representations based on domain vocabularies. *LNCS*, 4735, 2007, pp. 544–558.
3. Requirements-Driven Software Development System (ReDSeeDS) Project. EU 6th Framework IST Project (IST-33596). Available: http://www.redseeds.eu.
4. J. Miller, J. Mukerji et al. *MDA Guide Version 1.0.1*, omg/03-06-01. OMG, 2003.
5. T. Stahl, M. Voelter. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley, 2006.
6. C. Larman. *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design.* Prentice Hall, 1998.
7. A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA. *Proceedings of MDAFA 2004*, LNCS ,Vol. 3599, Springer, 2005, pp. 62–76.
8. A. Queralt, E. Teniente. A platform-independent model for the electronic marketplace domain. *Springer SoSym*, Vol. 7, No. 2, May 2008, pp. 219–235.

9. L. Leal, P. Pires, M. Campos. Natural MDA: Controlled Natural Language for Action Specifications on Model Driven Development. *Proceedings of OTM 2006*, LNCS 4275, pp. 551–568.
10. M. C. Leonardi, M. V. Mauco. Integrating natural language-oriented requirements models into MDA. Workshop on Requirements Engineering, WER, 2004, pp. 65–76.
11. B. B. Bryant, R. R. Raje, M. Auguston et al. From Natural Language Requirements to Executable Models of Software Components. *Proceedings of the Monterey Workshop on Software Engineering*, 2003, pp. 51–58.
12. J. Osis, E. Asnina, A. Grave. Computation Independent Modeling within the MDA. ICSSTE07, pp. 22–34.
13. R. G. Dromey. Formalizing the Transition from Requirements to Design. In: Jifeng He and Zhiming Liu (Eds.) *Mathematical Frameworks for Component Software – Models for Analysis and Synthesis*. World Scientific Series on Component-Based Development, 2006, pp. 156–187.
14. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, 1995.
15. R. Sriganesh, G. Brose, M. Silverman. *Mastering Enterprise JavaBeans 3.0*. Wiley Publishing, 2006.
16. J. Nilsson. *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET.* Addison Wesley, 2006.
17. F. Marinscu. *EJB Design Patterns*. John Wiley, 2002.
18. C. Bauer, G. King. *Java Persistence with Hibernate*. Manning, 2007
19. C. Richardson. *POJOs in Action*. Manning, 2006.
20. V. P. Mehta. *Pro LINQ Object Relational Mapping with C# 2008.* Apress, 2008.
21. S. Kavaldjian, H. Kaindl, K. S. Mukasa, J. Falb. *Transformations between Specifications of Requirements and User Interfaces.* 4th Int. Workshop MDDAUI 2009, pp. 37–40.
22. Fellbaum, C. (ed.) *WordNet: An Electronic Lexical Database*. MIT Press, 1998.
23. M. Rein, A. Ambroziewicz, J. Bojarski et al. Initial ReDSeeDS Prototype. Project Deliverable D5.4.1, ReDSeeDS Project, 2008. Available: www.redseeds.eu.
24. Sparx Systems, Enterprise Architect tool. Available: http://www.sparxsystems.com.au/.
25. M. Rein, A. Ambroziewicz, J. Bojarski et al. Final ReDSeeDS Prototype. Implementing the ReDSeeDS Engine prototype – 2nd iteration. Project Deliverable D5.4.3, ReDSeeDS Project, 2009. Available: www.redseeds.eu.
26. UL IMCS, MOLA pages. Available: http://mola.mii.lu.lv/.