Towards UML-Intensive Framework for Model-Driven Development

Darius Silingas[1,2], Ruslanas Vitiutinas[1,3]

[1] No Magic, Inc., Lithuanian Development Center
Savanoriu av. 363-IV, LT-44242 Kaunas, Lithuania

[2] Kaunas University of Technology, Information Systems Chair
Studentu 50-313a, LT-51368 Kaunas, Lithuania

[3] Vytautas Magnus University, Faculty of Informatics,
Vileikos 8-409, LT-44404 Kaunas, Lithuania

{Darius.Silingas, Ruslanas.Vitiutinas}@nomagic.com

Abstract. The paper describes a conceptual framework for model-driven development based on concise application of UML and modeling tool functionality. A case study of modeling software for library management is presented as an illustration of how to apply the proposed framework. Modeling tool features such as model transformations, code generation cartridges, model validation, dependency matrix, model metrics, model comparison, and model refactoring are presented as enablers for efficient model-driven development. The presented ideas and samples are based on industrial experience of authors who work as trainers and consultants for the modeling tool MagicDraw UML.

Keywords: UML, MDA, Model-Driven Development, MagicDraw.

Introduction

Model-Driven Architecture (MDA) is a new software development trend, which gains popularity in industry. Unified Modeling Language (UML) is considered a key tool for preparing source models for code generation in MDA environments. Recently, the language has undergone major changes moving from UML 1.4 version to UML 2.0. The rational behind most of these changes was to provide a better infrastructure for MDA, [1], [4]. However, recent MDA tools make rather limited use of UML 2 elements focusing mostly on class and activity or state diagrams. In this paper we present how UML 2 models can be concisely developed for domain analysis, requirements modeling, architectural decomposition, detailed design, implementation, and testing. With this framework we show where tools can help with automated transformations, and make more concise use of UML elements for modeling system implementation. Most of the presented ideas come from industrial experience of paper authors working as trainers and consultants for MagicDraw UML modeling tool.

Conceptual Framework for Model-Driven Development

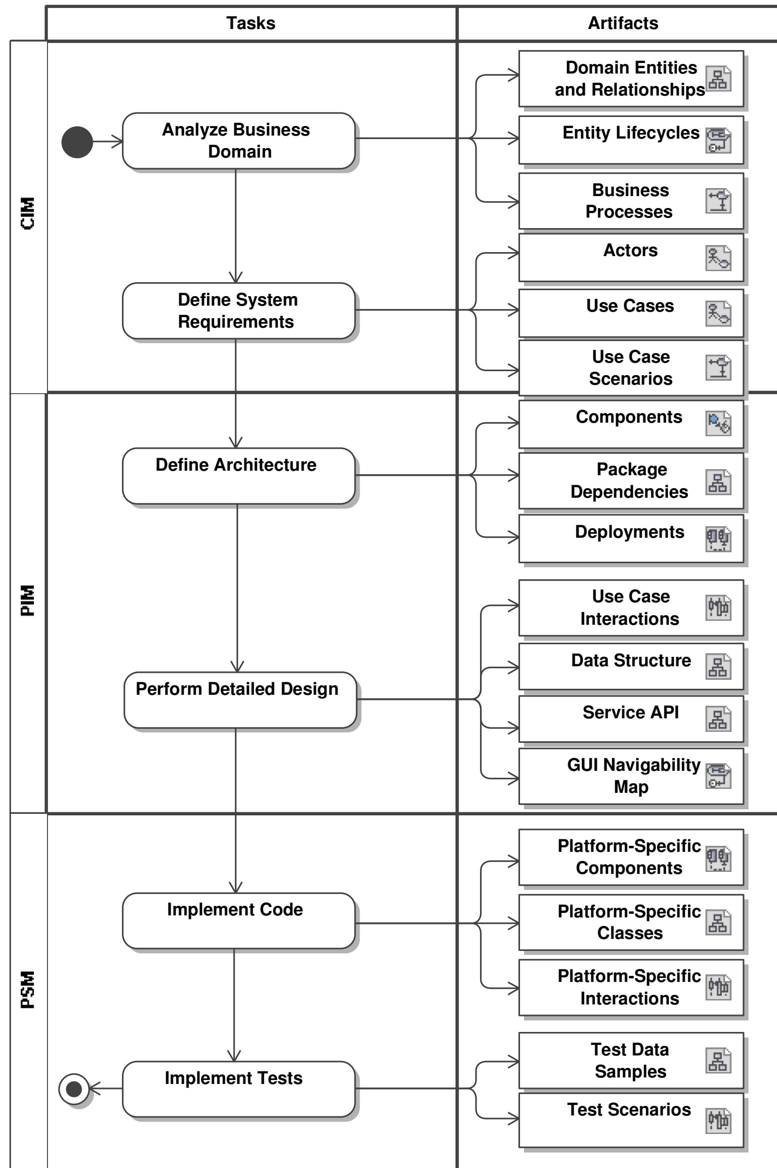

Fig. 1. Conceptual framework for model-driven development

MDA defines three abstraction levels of modeling: Computation-Independent Modeling (CIM), Platform-Independent Modeling (PIM), and Platform-Specific Modeling (PSM). We define the following major modeling tasks that should be completed while developing software system:

1. Analysis of business domain (purely focusing on CIM);
2. Definition of system requirements (mostly focusing on CIM);
3. Definition of high-level system architecture (mostly focusing on PIM);
4. Detailed design (mostly focusing on PIM);
5. Implementing code (mostly focusing on PSM);
6. Implementing tests (mostly focusing on PSM).

Each of these tasks should create of set of UML-based modeling artifacts. We present these tasks and artifacts in a conceptual framework, which is defined by UML activity diagram presented in Fig. 1. Although the framework defines the sequential logical flow of tasks, we want to emphasize that the nature of modern modeling is iterative – you need to repeat the tasks in iterations and come back for model updates.

MagicLibrary: a Case Study

In this chapter we will present more detailed description of modeling tasks and present major modeling artifacts for a case study system that will serve as illustrations for applying the conceptual modeling framework introduced in previous chapter.

A Case Study Problem Statement

A large organization maintains a library, which contains books, audio and video records. The organization made a decision to implement software system MagicLibrary dedicated for facilitating library usage and management.

MagicLibrary should support three types of users – librarian, reader, and administrator. Both reader and librarian are able to search for library items. Each library item is assigned to one or more categories and contains a list of keywords (optional). Item may be found either by browsing the category tree or searching for items by their property values. If reader finds a desirable item, he makes a reservation for it. If the item is immediately available then the reader is informed that he may contact librarian for loaning it according to the made reservation.

If the item is currently loaned out or assigned to another reservation then the reservation is put to the ordered waiting list. When the waiting reservation becomes available the system notifies the user. Notifications are sent either by e-mail or SMS according to the user preferences. Available reservations are automatically cancelled if reader doesn't come to take the item on loan for a period of time defined in system settings. Librarian registers the loan of the item and sets the due return date.

Librarian also registers the return of the loaned item. If a reader has kept the loaned item after the due date, he is given a penalty.

Reader may review his profile, which contains his reservations, loans, requests, and his personal data. Librarian is responsible for managing inventory data: items and tree of categories. Librarian is also responsible for managing MagicLibrary users and configuring system settings like default loan period, available reservation timeout, max reservations per reader, etc.

Analyze Business Domain

The purpose of business domain analysis is to understand how the business system works before going into development of software systems that automated some of business-defined procedures. We think that the basic views of business domain are identification of business entities and their structural relationships, models of the business processes, and lifecycles of business entities that have important states defining applicability of business actions.

We recommend starting with definition of business entities and their relationships using simple class diagram using classes displaying only name compartment and named associations, see Fig. 2. Additionally, you may define association role cardinalities for better understanding of relationship nature. Such diagram serves as visual dictionary of business terminology. The terms defined in it should be used consistently in all other models. Of course, while modeling, you will need to come back and update this diagram to reflect the discovered changes.

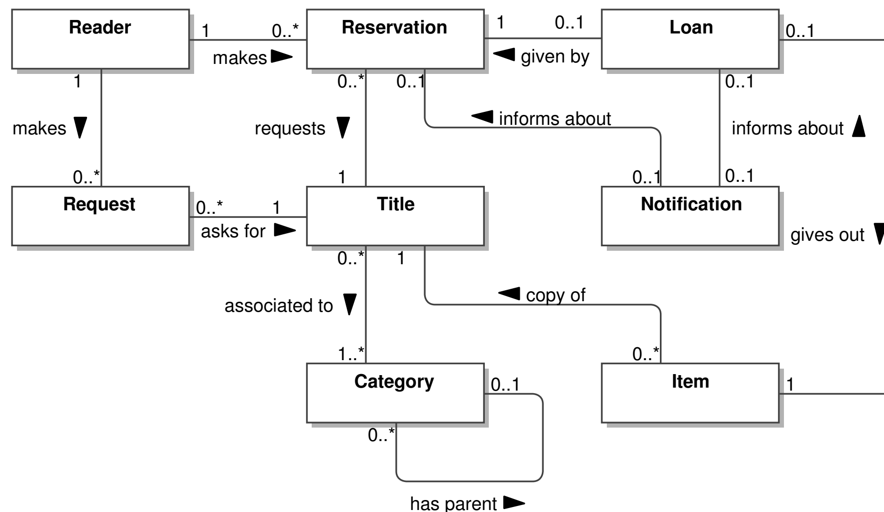

Fig. 2. Library business domain entities and relationships class diagram.

Business entity lifecycle can be presented with analysis-level state diagram indicating what behaviors are possible on which entity state, what actions trigger transitions between states, and what activities should be executed as side effect of transition, entering a state or exiting from it. In concise modeling, states machine model should be assigned to entity class using classifier's behavior property that is available since UML 2 [1]. A sample state machine, defined in Fig. 3, should be created inside class Item, and assigned as behavior for that class.

Very important modeling artifacts are business processes that can be defined using either pure UML activity diagrams or Business Process Modeling Notation (BPMN) [3]. In such diagrams, the focus is on the sequence of tasks, separation of responsibilities, decision points, triggering business events, communications between processes in different organizations, and exchanged data, mostly documents.

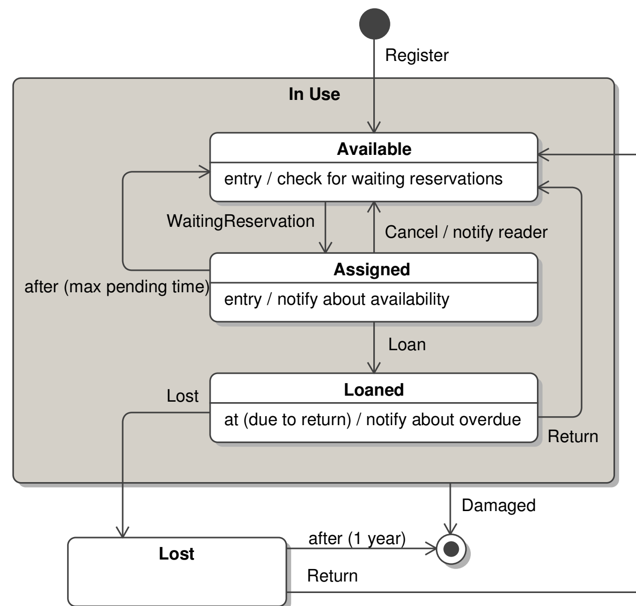

Fig. 3. Analysis-level state diagram presenting lifecycle of *Item* entity.

Define System Requirements

Software development is driven by the expressed requirements. Most of the major approaches to requirements analysis are based on use case method. In the famous 4+1 architectural views model, use cases are defined as central modeling artifact [5]. We suggest the following sequence of use case modeling tasks:

1. Define actors (in a separate diagram) grouping them into primary (main users), secondary (administration, maintenance, support), and system actors.
2. Define main system use cases in a sketch use case diagram.
3. Group the created use cases into packages according to their coherence.
4. Prepare use case package overview diagram, showing which actors uses which use case package. Alternatively, use case model overview can be done using so called dependency matrix, which in tabular forms show the relationships between actors and use cases grouped by packages.
5. Prepare use case package details diagram, showing package use cases, their associations with actors, relationships between use cases including uses cases from different packages (shown onside the package symbol), Fig. 4.
6. Prepare activity diagrams visualizing scenarios of complex use cases, Fig. 5. In model, the activities should be nested within appropriate use cases and assigned as their behaviors. Some actions may call reusable activities.
7. Document use cases according to pre-defined templates, e.g. *Rational Unified Process* or *Process Impact* defined use case templates.

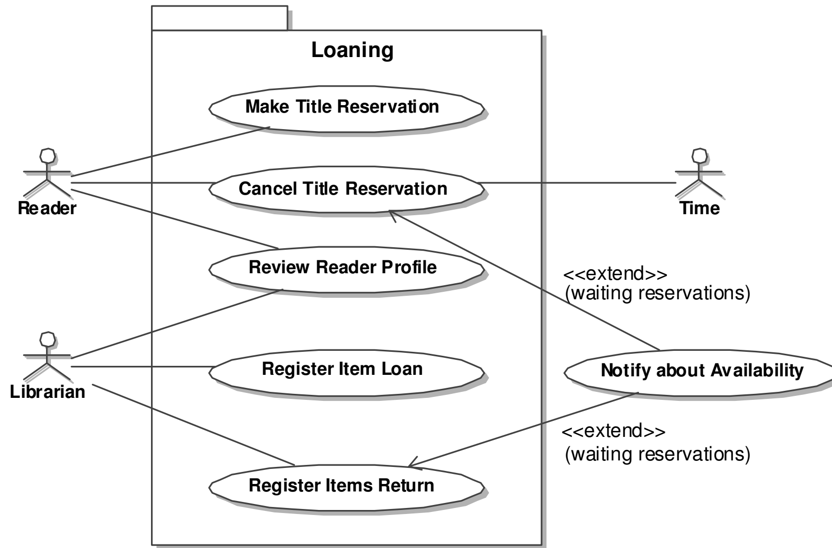

Fig. 4. Use case diagram showing details of use case package *Loaning*.

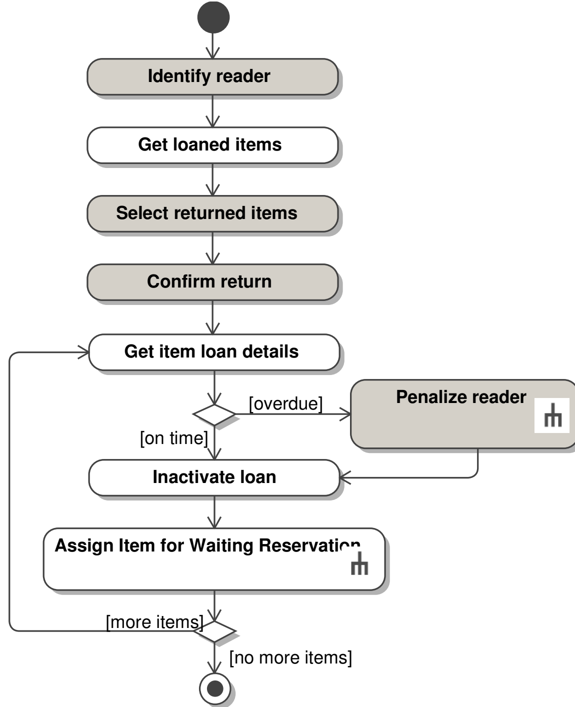

Fig. 5. Activity diagram showing scenarios for use case *Register Items Return*.

However, use cases define just the user-level functional requirements. In practice, functional requirements are usually decomposed into smaller-grained functional requirements; also many types of non-functional requirements are specified [6]. For enabling structural modeling of requirements and their relationships, we suggest to prepare a custom class diagram enhancement for requirements modeling. A similar approach is taken in SysML [2]. This approach also enables requirements traceability using tool-generated dependency matrices.

Define Architecture

Design of the software system starts by specifying its high-level architecture, which is usually modeled in structural component, package dependency, and deployment diagrams. Most of the modern business-oriented software systems are built on layered architecture pattern. For such systems, we recommend to start architectural design with package dependency and robustness analysis diagrams [7]. Package dependency diagram shows how the system is organized into layers, Fig. 6.

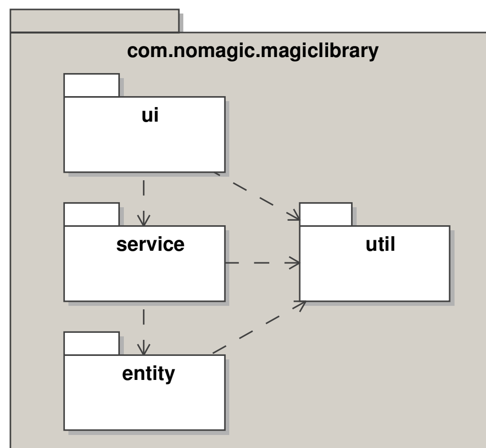

Fig. 6. Package diagram showing dependencies between top-level packages.

Robustness analysis diagram defines the major components in different layers – interface boundaries, controll services, data entities, – and their relationships crossing layer boundaries, Fig. 7. It is important to show which actor uses which interface boundaries. The inception for the data entities are the conceptual entities, for the controll services – use case packages, and the interface boundaries need to be invented at this point and may be refined later. Each of the defined components should be placed into appropriate packages. In the subsequent detailed design task, it is necessary to refine each components by modeling their details.

It is common at architectural design level to define at least the major ideas about system deployment. This may usually includes platform-specific information. At this point, it is important to define only the major deployment artifacts, topology of network, and communication protocols, Fig. 8.

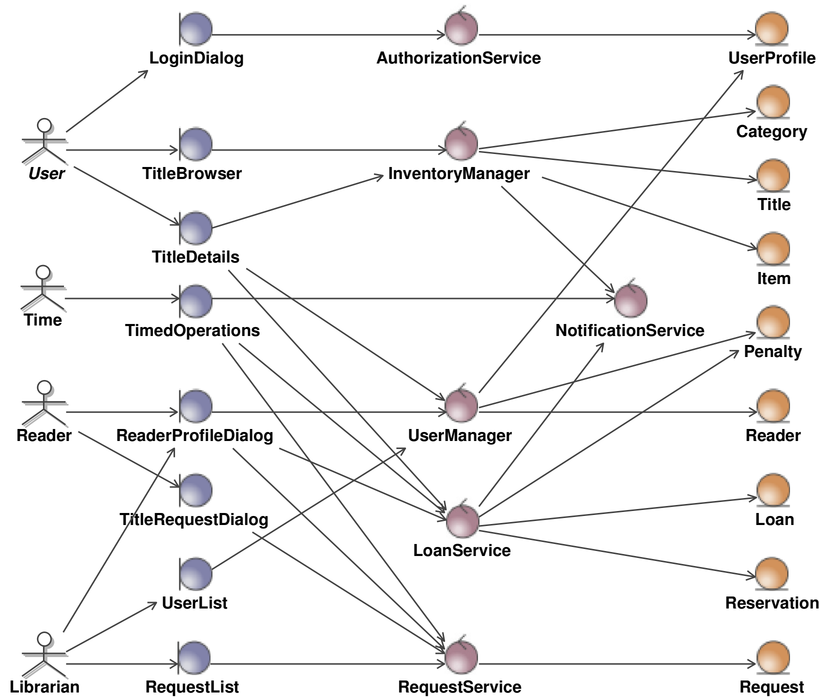

Fig. 7. Robustness analysis of relationships between components in layers.

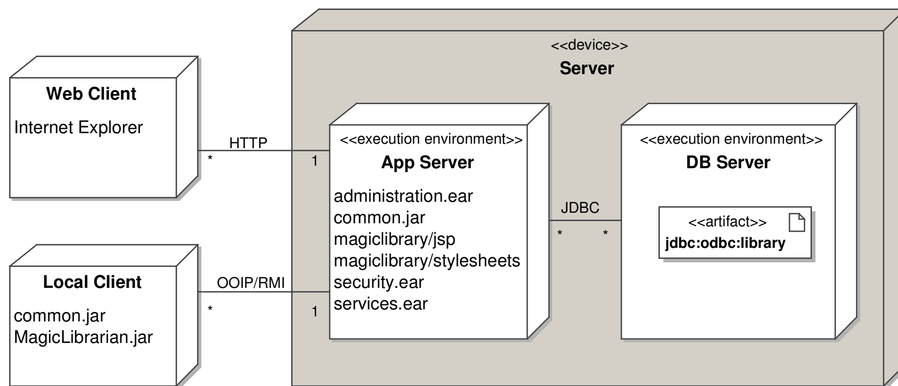

Fig. 8. Deployment Architecture

While UML 2.0 districts the deployment of components, and insists of deploying artifacts, it is necessary to separate logical components and physical artifacts. The relationships between those two sets are modeled by manifestation relationships. We recommend dependency matrices for representing and editing manifestations, Fig. 13.

Perform Detailed Design

After the structural components are identified, it is time to provide more details for each of them. Different aspects are important for different layers:

- Data entity layer should focus on specification of data class attributes and details of associations, such as navigability and role names, Fig. 10.
- Service layer should focus on specification of API, which is defined in public operations of services. We recommend specifying service operations while designing interactions for use case scenarios. With this approach, it is easier to see the context in which operations is used, which allows to define appropriate responsibilities and their contracts (operation parameters, returns, visibility and other properties). We consider this as modeling substitute for test-driven programming approach since we define structural class features – operations, while designing behavioral scenarios, Fig. 9. Later service API can be visualized using class diagrams showing only operations and dependencies between services.
- User interfaces should focus on GUI navigability maps and definition of inner structure of GUI elements. The former is best modeled with state diagram, where each state defines separate GUI component and the transition triggers define GUI events, Fig. 11. The later is usually done using graphical prototypes that are not based on UML. In UML 2 introduced composite structure diagram is a potential candidate for this task but the tool support for modeling this in user-friendly way is still not available.

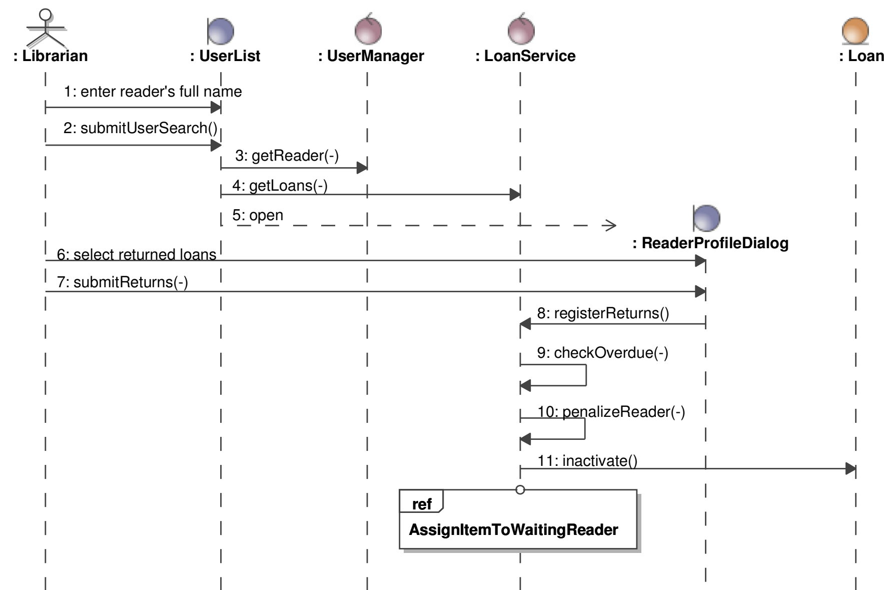

Fig. 9. Sequence diagram showing interaction of identified components for implementing Register Item Return "happy day" scenario

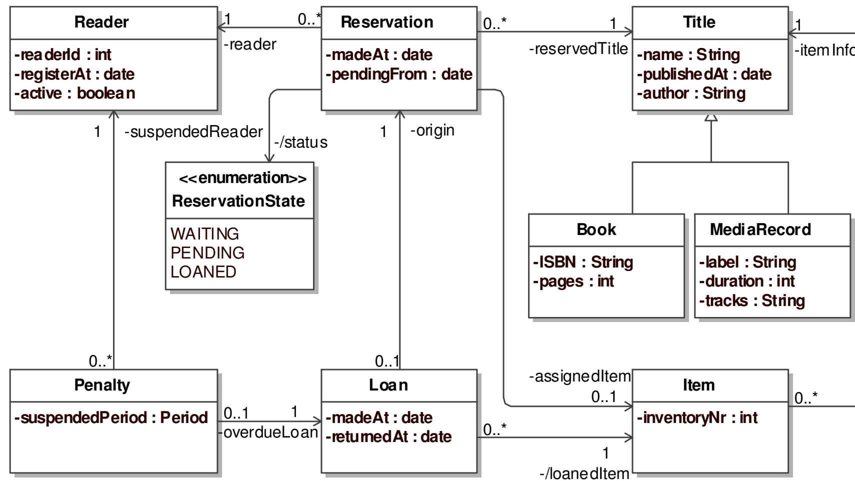

Fig. 10. Class diagram emphasizing data entity attributes and relathioships.

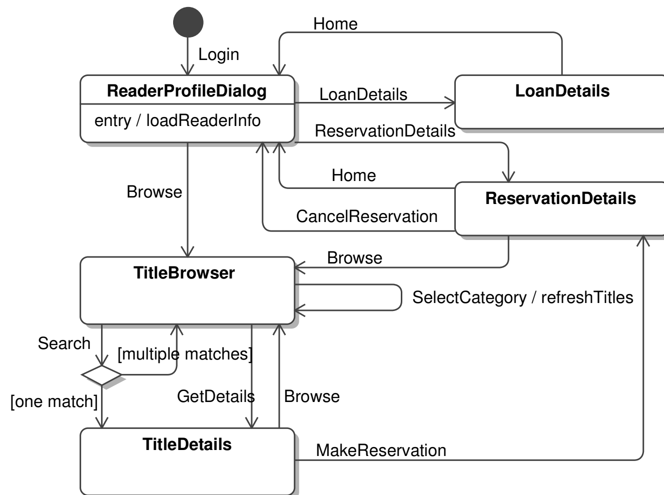

Fig. 11. User interface navigation schema for actor *Reader*.

Implement Code

In model-driven approach, we use the created model artifacts for generating code. For enabling code generation, we need to enrich PIM level models with platform-specific information. It is recommended to do it via semi or fully automated model-to-model transformations. Code generation from PSM models is implemented either by plug-able code generation cartridges or model-to-code transformations. As a sample, we

present a fragment for PSM model for relational database design, Fig. 12 and DDL script generated from this PSM model, Tab. 1.

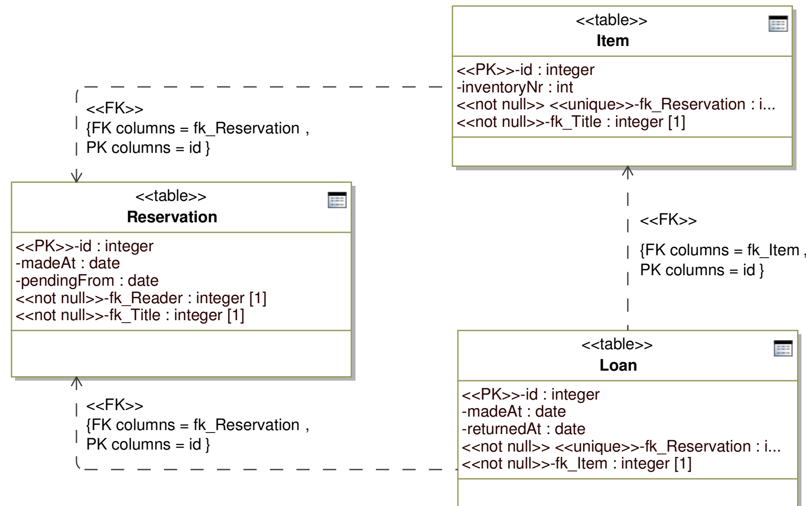

Fig. 12. A fragment of PSM for relational database structure

Tab. 1. A fragment of DDL script generated from PSM for database design.

```
CREATE SCHEMA MagicLibrary;
...
CREATE TABLE Reservation (
   id integer PRIMARY KEY,
   madeAt date,
   pendingFrom date,
   fk_Reader integer NOT NULL,
   fk_Title integer NOT NULL,
   FOREIGN KEY(fk_Reader) REFERENCES Reader (id),
   FOREIGN KEY(fk_Title) REFERENCES Title (id)
);
CREATE TABLE Item (
   id integer PRIMARY KEY,
   inventoryNr int,
   fk_Reservation integer NOT NULL UNIQUE,
   fk_Title integer NOT NULL,
   FOREIGN KEY(fk_Reservation) REFERENCES Reservation (id),
   FOREIGN KEY(fk_Title) REFERENCES Title (id)
);
CREATE TABLE Loan (
   id integer PRIMARY KEY,
   madeAt date,
   returnedAt date,
   fk_Reservation integer NOT NULL UNIQUE,
   fk_Item integer NOT NULL,
   FOREIGN KEY(fk_Reservation) REFERENCES Reservation (id),
   FOREIGN KEY(fk_Item) REFERENCES Item (id)
);
...
```

Implement Tests

For implementation of tests, a modeler may use object diagrams for visualizing data structures that should be used in testing. The sequence diagram, which is not a very good tool for modeling algorithms, is perfectly suited for defining test scenarios. The assertion fragment introduced in UML 2 is useful is very useful construct for that purpose. Both these diagrams are suitable artifacts for code generation, [8], [9].

Enabling Toolkit for Model-Driven Development

For enabling efficient model-driven development, the UML modeling environment should provide multiple features. Below we list features that are already supported in modeling tools like MagicDraw UML:

- Concise integration of model elements through UML-define properties;
- Model validation;
- Plug-able patterns;
- Plug-able model transformations;
- Plug-able code generation cartridges;
- Modeling project decomposition in several modules;
- Interactive modeling teamwork;
- Tracking model element relationships;
- Dependency matrices representing model element relationships;
- Calculating model metrics;
- Model documentation reports.

The following features are also highly demanded, but are not yet implemented (or implemented very poorly) in the state-of-the-art modeling tools:

- Model refactoring;
- Merging changes between different model versions;
- Modeling unit tests for supporting test-driven modeling approach.

With the mentioned tools at their hands, the modelers can apply a model-driven development in efficient ways allowing increase of development speed and quality.

Summary

We have introduced the conceptual UML-intensive framework for model-driven development and provided case study-based illustrations of essentials modeling artifacts that should be produced as results of tasks presented in the framework. In the context of these artifacts we discussed the ideas of how to increase modeling efficiency by desirable automation features. Finally, we have briefly introduced the modeling environment features that need to be supported for enabling efficient and agile software development while applying the proposed framework. We hope that the presented information is valuable starting point for more detailed industrial and academic research on the topic of UML-intensive model-driven development.

References

1. Object Management Group. UML 2.1.1 Unified Modeling Language: Superstructure, Specification, 2007
2. Object Management Group. Systems Modeling Language (SysML), Specification, 2006
3. Object Management Group. Business Process Modeling Notation (BPMN), Specification, 2006
4. Thomas O.Meservy, Kurt D.Fenstermacher: Transforming Software Development: An MDA Road Map. Computer Volume 38, Issue 9, Sept. 2005 Page(s): 52 – 58
5. P.B. Kruchten, The 4+1 View Model of architecture. Software, IEEE Volume 12, Issue 6, Nov 1995 Page(s): 42–50
6. Karl E. Wiegers, Software Requirements, Second Edition, Microsoft Press, Redmond,Washington, 2003.
7. van Leeuwen, J. (ed.): UML Software Architecture and Design Description. Christian F.J. Lange and Michel R.V. Chaudron. In: Software, IEEE Volume 23, Issue 2, March-April 2006 Page(s): 40 – 46
8. Thongmak, M.; Muenchaisri, P.: Design of Rules for Transforming UML Sequence Diagrams into Java code. Software Engineering Conference, 2002. Ninth Asia-Pacific Volume , Issue , 2002 Page(s): 485 – 494
9. Nebut, C.; Fleurey, F.; Le Traon, Y.; Jezequel, J.-M.: Automatic Test Generation: A Use Case Driven Approach. In: Software Engineering, IEEE Transactions on Volume 32, Issue 3, March 2006 Page(s): 140–155