

PIM to PSM transformations for an event driven architecture in an educational tool

Geert Monsieur¹, Monique Snoeck¹, Raf Haesen^{1,2}, Wilfried Lemahieu¹

¹KULeuven, Faculty of Economic and Applied Economic Sciences
Management Information Systems Group
Naamestraat 69; 3000 Leuven, Belgium
²Campus Vlekhoe
Koningsstraat 336, 1030 Brussel, Belgium
{Geert.Monsieur, Monique.Snoeck, Raf.Haesen, Wilfried.Lemahieu}@econ.kuleuven.be

Abstract. This paper presents experiences with the use of an MDA approach to generate prototype applications from a conceptual domain model in the context of teaching object-oriented domain modelling. Each conceptual model used to generate the prototype consists of a combination of three views (a class diagram, a proprietary object-event table and a group of finite state machines) and constitutes as such the platform-independent model (PIM). We describe in detail how our event-driven PIM is transformed into an event-driven platform-specific model which is almost directly used to generate the running prototype application (the code). We conclude with a discussion of the lessons we learned, problems we faced, potential solutions and critical aspects for a successful MDA story.

1 Introduction

This paper presents experiences with applying the MDA-approach in an educational context, namely in the context of teaching object-oriented business domain modelling¹. The main goal of the course is to familiarise students with object-oriented analysis techniques and to let them acquire the necessary skills to actually perform domain modelling. In addition, students should be able to envisage the mutual relationship and influence between the IT-system and the organisational work system. For example, the student should learn to evaluate the implications of differences between models both from an IT and an organisational perspective.

The student's prior capability is a crucial element to achieve the educational goal. The course is attended by a set of students with a large variation in preliminary knowledge of object-oriented concepts and programming skills. One can roughly identify three major groups of students. One group of students are very much technology oriented and have good object-oriented programming skills. Their IT-orientation is sometimes a disadvantage as it hampers them in taking a more business oriented view towards specifications: they tend to think in a solution-oriented mode, while the goal of the

¹ The course's page can be found on <http://mermaid.econ.kuleuven.be/content.aspx/>

course is to focus on the characteristics of the problem world. The second group of students have a substantial set of business-oriented skills, are skilled in data modelling and have basic skills in object-oriented programming. This is the group of students that is most at ease with the presented material. However, because of their preliminary database course, these students tend to think in a purely data-oriented way, and have difficulties in conceptualizing the behavioural aspects of an object-oriented model as defined in class operations and finite state machines (FSM). Finally, for a third group of students, this course is their first contact with object-orientation. They have no programming skills and have not yet followed a course on database modelling. Especially this group has difficulties of imagining the concrete system that will result out of an abstract model.

One of the major goals is that the student should be able to form a mental model of a concrete information system built from the conceptual model he/she made. We can describe a mental model as a model that gives a clear concretisation of the working and behavioural aspects of an information system. One of the techniques to concretise models is prototyping, a widely-spread approach for validating user requirements. However, whereas any student with some crafting skills can build a prototype of a house with simple tools such as a cutter, card board and glue, building prototypes of information systems requires a reasonable amount of programming skills and time. Given the fact that the majority of students have no or but limited programming skills, requiring students to build a prototype for every single model they make is impossible. Nevertheless, models only are too abstract and many students find it difficult to understand the consequences implied by a difference between two potential models for a same set of requirements.

Many tools are able to generate fairly easily simple data-oriented prototypes providing standard create, update and delete features per business object type (e.g. using wizard-generated forms in MS Access). However, the goal of the course goes beyond data modelling: students should be able to conceptualise the behavioural aspects and interaction aspects of business objects as well. For generating behavioural aspects, UML advocates the use of finite state machines to model behavioural aspects of a class and collaboration diagrams or interaction diagrams to model the interaction aspects. While the latter diagrams can still be considered as platform independent models, they are at a quite low level of abstraction, since they model software behaviour at the level of the individual message exchanges. Domain modelling is inherently much more abstract. Whereas the life-cycle of a business object can be modelled at a sufficiently high level of abstraction by means of a finite state machine, the design of message exchanges between objects is more oriented towards the solution space than to the problem space. As an example, the life cycle of an order in terms of its creation, confirmation, shipping and payment is at a sufficiently high level of abstraction. The fact that the creation of an order requires the verification of the customer's status, the availability of the product the planning of the shipping will require some interaction between several domain classes. The exact sequence of the processing is however beyond the scope of domain modelling. It is the task of the *designer* to decide the best interaction schema to realise the creation of an order. As a result, one of the goals of our research is to devise a standard pattern for modelling and implementing domain object interaction.

An additional goal is to develop a modelling and implementation approach that is scalable to real life systems.

A final consideration is the user interface of the generated prototype. Similarly as for domain interaction, the modelling of user-system interaction is beyond the scope of domain modelling. Hence, also for the user interface, we need to resort to a pattern-based approach. In this case however, scalability to real-life system was not part of our goal.

2 Solution

Students create a platform independent model of the business domain. The tool will then map this PIM to a PSM and code, using a set of standard patterns. In this section, we first describe the concepts of our PIM, subsequently followed by a brief discussion of our PSM and we conclude this section with an overview of the rules we defined for the PIM-to-PSM transformation.

2.1 The platform independent model (PIM)

The platform independent model consists of a class diagram, an interaction model and a number of finite state machines. We decided to employ the widely adopted Unified Modelling Language (UML) as the modelling notation for all diagrams. This way, it is easy to interoperate with other tools using the XML Metadata Interchange (XMI) format. Additionally, most MDA frameworks support the use of UML, such as the AndroMDA framework² we used to implement our solution.

The class diagram is a restricted form of UML class diagram: the types of associations are limited to binary associations, with a cardinality of 1-to-many or 1-to-1. Many-to-many associations need to be converted to an intermediate class. In theory, this conversion could be realised as a transformation as well. For didactic and methodological purposes, students are required to transform these associations manually: it obliges them to think about the constraints that apply to this association class (see [12] for a detailed motivation of this approach).

Figure 1 shows a class diagram for a sales domain that will be used throughout the rest of this paper as running example.

² <http://www.andromda.org>

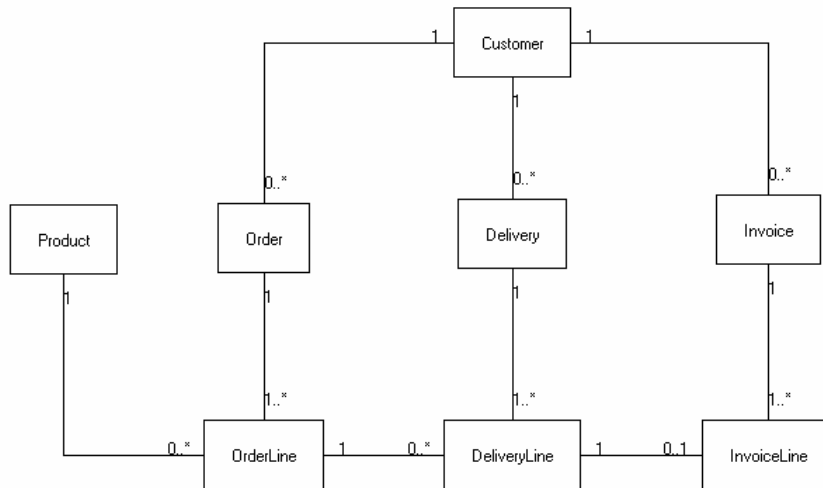


Fig. 1. A class diagram for the sales example

Each class has a number of operations that are classified as creators, modifiers or ending operations. In a typical object-oriented model, classes interact by invoking each other's operations. As explained in the previous section, domain modelling is at a high level of abstraction and oriented to the problem space rather than to the solution space. When an action requires processing in several domain classes, we do not want the modeller to design the details of the collaboration between the classes, as this pertains to the solution space rather than to the problem space. We therefore developed a specific modelling technique which allows identifying atomic actions and indicating which classes are involved in the processing of that action. The atomic actions are called 'business events' to stress their real-world (=business) and atomic (=event) character. The concept of 'business event' is very similar to the concept of event as used in Syntropy [2] and action in Catalysis [3]. In the Object-Event Table (OET) the requirements engineer indicates which domain object type is involved in the processing of a given business event. This processing can be the validation of the action, the creation, the modification or the ending of an instance of that domain object type, respectively indicated with V, C, M or E in the table (see table 1). As an example, creating an order may require the domain object type Customer to verify the customer status (blacklisted or not), and the object type Order to create an instance of that class. Adding an order line to that order to order a product, will require the domain object type Product to verify the availability of the product and update the stock-level of that product, will require the domain object type OrderLine to create an instance of that class and will require the domain object type Order to add the created order line to the order instance.

	Customer	Product	Order	OrderLine	...
<i>cr_order</i>	V		C		
<i>cr_orderLine</i>		V+M	M	C	
....					

Table 1. Partial Object-Event table for the Sales example

Obviously the OET is not a default modelling technique supported by the UML. The implementation of a business event requires the elaboration of a collaboration schema that specifies the required interactions to process the action. If a standard collaboration pattern can be defined, this can be used to automate the transformation of the OET to equivalent UML concepts that are used to generate a platform specific model.

Finally, the platform independent model of the domain contains one or more finite state machines per domain object type. The finite state machines allow the object type to impose sequence constraints on the business events it is involved in. Multiple Finite State Machines allow to model independent aspects as parallel machines. Before transformation, these are algorithmically transformed to a single Finite State Machine that represents the parallel composition of all specified finite state machines for that object type. Figure 2 shows an example of a FSM for the object type Product.

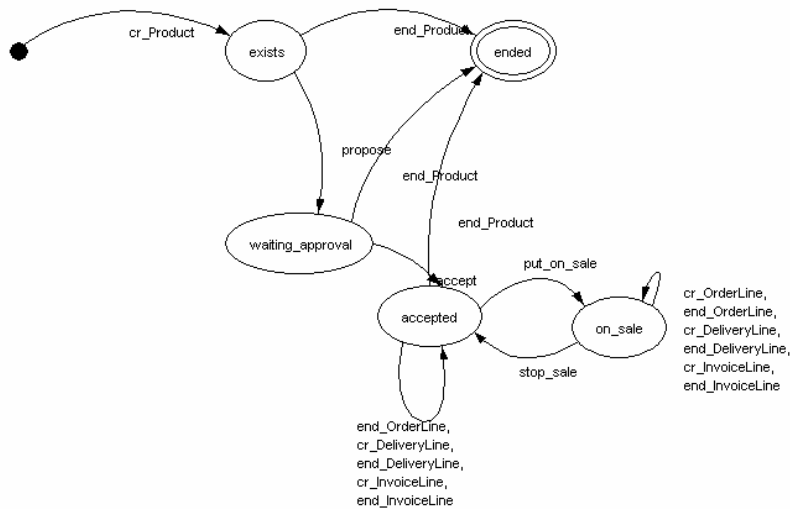


Fig. 2. Finite State Machine for Product

2.2 The platform specific model (PSM)

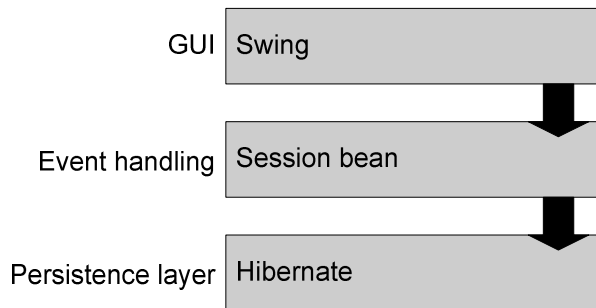


Fig. 3. The three layers of the platform specific model

The target architecture consists of three layers, a graphical user interface (GUI) layer, an event handling layer and a persistence layer (see figure 3). Although any object-oriented programming languages would provide us with the required programming mechanisms, we chose the J2EE framework to implement the solution.

The graphical user interface (based on the Java Swing architecture) has only basic functionality like triggering the creating and ending of objects, and triggering other business events. The GUI layer is built on top of the event handling layer. The task of the latter layer is to handle all events correctly by managing the appropriate interactions with the objects in the persistence layer.

2.3 Transformation rules

We will now discuss our target architecture in detail by focusing on the PIM-to-PSM transformations. In the graphical representations of the transformation rules we use white boxes for PIM concepts, dark gray boxes for the PSM concepts and light gray arrows for the transformation.

Persistence layer

This layer forms the basis of our target architecture. All other layers in our architecture will make use of the services provided by the persistence layer. Generation of this layer is made possible by transforming three PIM elements, namely the class diagram, the columns of the object event table and the finite state machines.

Transforming the class diagram and operations (columns of OET)

For transforming the class diagram and operations we have defined the following transformation rules:

Each object type in the PIM is transformed into (see figure 4):

- an *abstract class* and *implementation class* in our PSM. By making a distinction between an abstract and implementation class it is easier to see the difference between generated PSM elements and PSM elements added

manually after the PIM-to-PSM transformation took place. This leads to a better traceability which can be important for managing the complexity of MDA [6]. Furthermore we use the object-relational mapping (ORM) technique of Hibernate³ to make this type of objects persistent.

- A *factory* which makes it easier to create and collect objects of the same type [4].
- One *method* (in the mapped abstract class) for each event which the object type participates in and one *method* for checking the corresponding preconditions. The (columns of the) Object-event table contain(s) the information needed for this transformation rule.

Pseudo code:

```
method_for_event () and check_preconditions_for_event ()
```

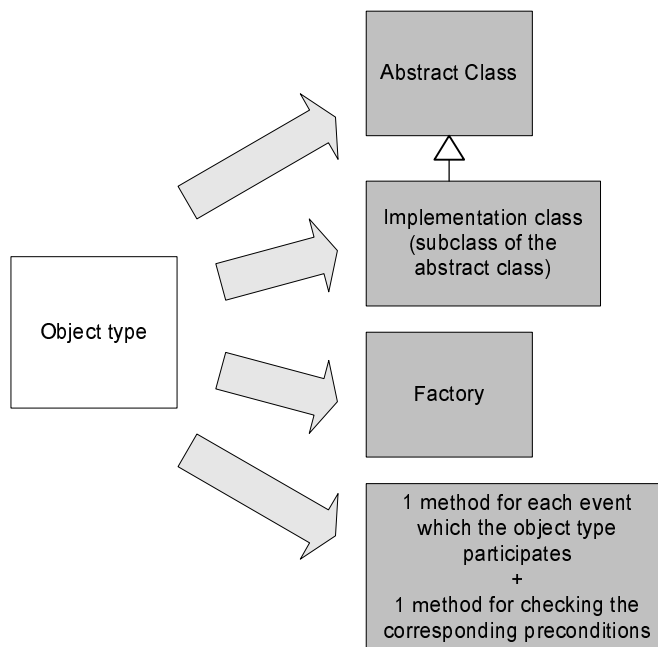


Fig. 4. Transformation rules for an object type

Each attribute of an object type is transformed into an attribute of the mapped class (see figure 5).

Each one-to-many association in our class diagram is transformed into an association attribute in the mapped classes (see figure 6).

³ <http://www.hibernate.org>

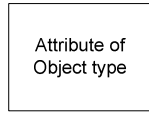


Fig. 5.

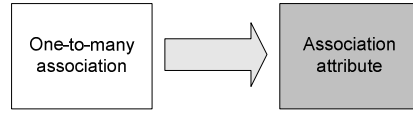


Fig. 6.

Transforming the finite state machines

To transform the finite state machines we created the following transformation rules:

An object type's finite state machine is transformed into (see figure 7):

- *An abstract state class* (associated with the mapped class) with *state subclasses* for each state in the FSM: this is based on the state pattern [4].
- *Methods in the state subclasses* for checking state conditions for each event which the object type participates in.

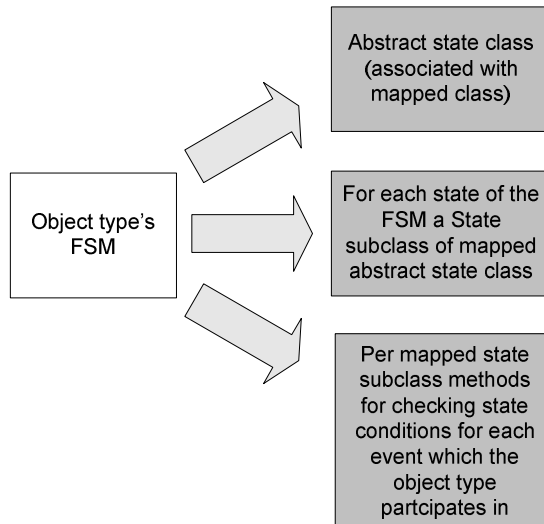


Fig. 7. Transforming the Finite State Machines

Event handling layer

The event layer can be seen as the heart of our *event-driven* target architecture. Good organised mechanisms to handle events play a central role in an event-based PSM. For this purpose we currently use one session bean which bundles all needed event handlers, although it is possible to use another way of transforming the event-driven concepts, e.g. by using one PSM class per event. This would of course lead to a different transformation and a different PSM.

Transforming the events (rows of OET)

The event layer described in this paper consists of only one session bean which *handles* the events. In general the following transformation rule is valid for our PSM: For each event (found as rows of the OET) there is an operation (an event handler) in a so called EventHandlerBean (see figure 8). The implementations of these event handlers are also generated. The latter generation follows a *standard collaboration pattern*. We will now discuss further details about this pattern.

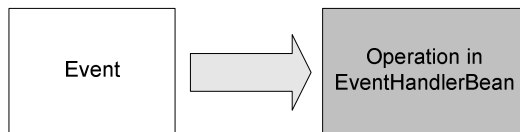


Fig. 8. Transforming the events

Standard collaboration pattern for generation of handle_event()

We summarize the pattern in four steps. Steps 1 and 2 are represented in figure 9. Steps 3 and 4 are visualized in figure 10.

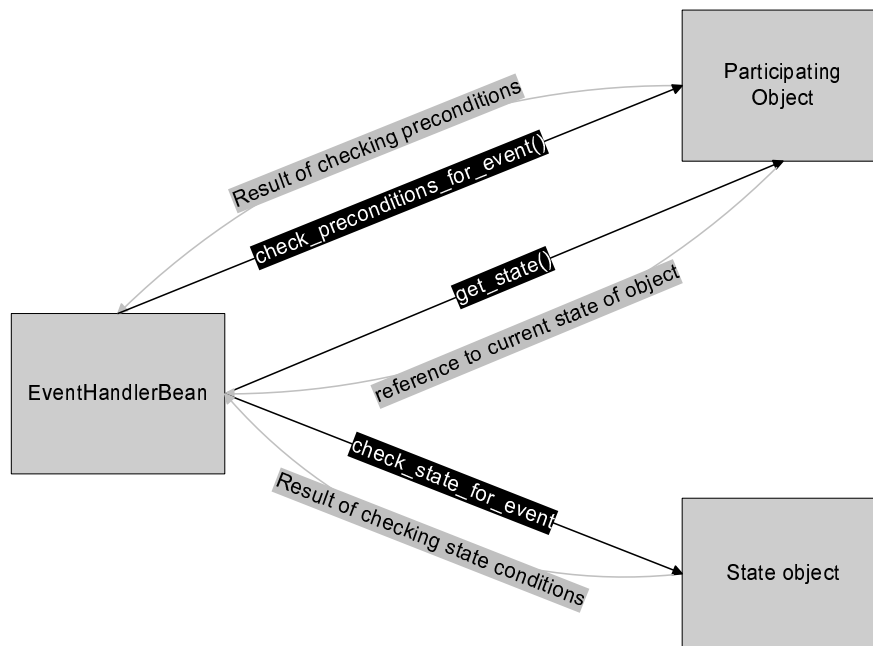


Fig. 9. Standard collaboration pattern for generation of handle_event() (step 1 and 2)

Step 1 The event handler ‘asks’ every participating object (an object which is involved in the processing of a business event) whether all preconditions set by the object are met. These preconditions can be defined by the analyst (as V’s in the OET), but also include conditions that can be derived from other parts of the model. For example, associations between classes will lead to preconditions to maintain referential integrity.

Pseudo code:

```
participant_1.check_preconditions_for_event ()
participant_2.check_preconditions_for_event ()
...
participant_n.check_preconditions_for_event ()
```

Step 2 Similarly to the previous step the event handler retrieves from every participating object its current state (or reference to the corresponding state object) and checks whether that state allows further processing of the event.

Pseudo code:

```
participant_1.getState().check_state_for_event ()
participant_2.getState().check_state_for_event ()
...
participant_n.getState().check_state_for_event ()
```

Step 3 If all results of the tasks in step 1 and 2 are positive (this means ‘no exceptions are thrown’), the event handler invokes the methods in the participating objects which correspond with the triggered event (i.e. the C’s, M’s and E’s in the OET).

Pseudo code:

```
participant_1.method_for_event ()
participant_2.method_for_event ()
...
participant_n.method_for_event ()
```

Step 4 Next, (if all results of the tasks in step 1 and 2 are positive) the event handler executes in all state objects retrieved in step 2 the method for modifying the state (according to the triggered event).

Pseudo code:

```
participant_1.getState().change_state_for_event ()
participant_2.getState().change_state_for_event ()
...
participant_n.getState().change_state_for_event ()
```

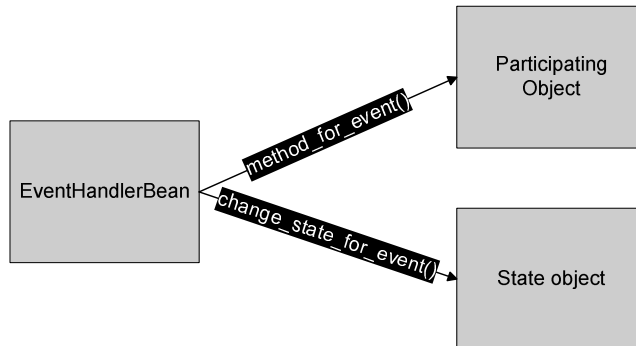


Fig. 10. Standard collaboration pattern for generation of `handle_event()` (step 3 and 4)

In case of negative results in step 1 and/or 2 an exception is thrown. In the prototype application the end-user will notice this exception as a message box which tells what went wrong (violated preconditions or state conditions) and why the processing of the event is not allowed.

GUI layer

The design of the GUI layer is based on a pattern for event-based user interface proposed in [10]. Although the GUI layer is like other parts of our PSM strongly based on the event-based aspects of our PIM (e.g.: one window generated per defined event, etc.), we consider the discussion of these transformation rules as beyond the scope of this paper.

2.4 Transformation rules applied for the sales example

In this section we show (some part of) the results from applying the transformation rules to the sales example described earlier in this paper in the discussion of our PIM concepts.

Generated persistence layer

In figure 11 the generated persistence layer is partially shown (to maintain the overview only two transformed object types are shown, and some methods are hidden).

The transformed class diagram and operations (columns of the OET) are visualized as white boxes. One can see that object type `OrderLine` is transformed into an abstract class `OrderLine`, an implementation class `OrderLineImpl` and a factory class `OrderLineFactory`. The attributes and methods to check preconditions and process events are also generated for the `OrderLine` class. The required methods for the `Product` class are not shown to avoid overloading the figure with too much information.

The gray boxes in figure 11 represent the transformed finite state machines. The different states for `Product` are transformed in different subclasses (`ProductExistsState`, `ProductWaiting_ApprovalState`, ...) of an abstract state class

(ProductState). Only a few generated methods for checking state conditions and changing states are shown in this figure.

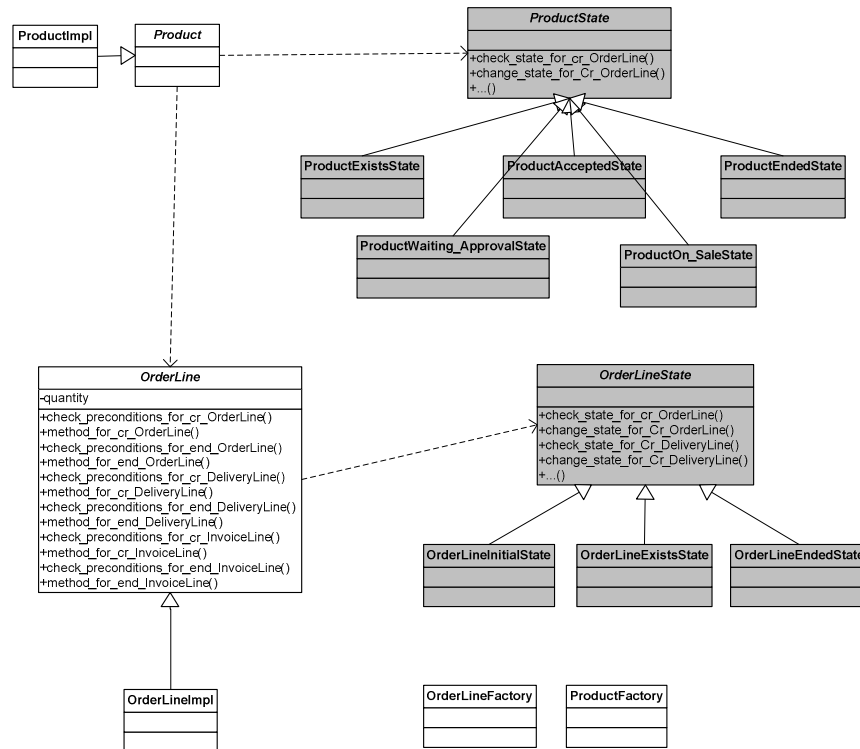


Fig. 11. Partial generated persistence layer

Generated event layer

At first sight the generated event layer looks very straight forward (see figure 12). However the real challenge is the implementation of the different event handlers defined in the session bean. We give an example in which we apply the collaboration pattern for the event cr_OrderLine.

EventHandlerSessionBean
+handle_cr_Customer() +handle_end_Customer() +handle_cr_Product() +handle_end_Product() +handle_propose() +handle_accept() +...() +handle_cr_OrderLine() +...()

Fig. 12. Generated event layer

Standard collaboration diagram applied for event cr_OrderLine()

Involved objects:

orderLine, order, product

Pseudo code for handle_cr_OrderLine()⁴

```
// checking preconditions (step 1)
orderLine.check_preconditions_for_cr_OrderLine ()
order.check_preconditions_for_cr_OrderLine ()
product.check_preconditions_for_cr_OrderLine ()
// checking state conditions (step 2)
orderLine.getState ().check_state_for_cr_OrderLine ()
order.getState ().check_state_for_cr_OrderLine ()
product.getState ().check_state_for_cr_OrderLine ()
// event processing (step 3)
orderLine.method_for_cr_OrderLine ()
order.method_for_cr_OrderLine ()
product.method_for_cr_OrderLine ()
// state modifications (step 4)
orderLine.getState ().change_state_for_cr_OrderLine ()
order.getState ().change_state_for_cr_OrderLine ()
product.getState ().change_state_for_cr_OrderLine ()
```

3 Conclusions

3.1 Summary

This article has presented an approach to derive prototype applications from conceptual domain models using the MDA ideas. Each conceptual model consists of a

⁴ Actually a call to OrderLineFactory is required before the execution of step 1 is possible.

combination of three views and constitutes as such the platform-independent model. The first view presents the business object types and their interrelations by means of a class diagram. The dynamic view consists of a proprietary object-event table that identifies business event types and indicates how a business event type affects object types. If an object type is affected by an event type it will define a method implementing the effect of this event type on objects of this type. Finally a set of finite state machines constrain the invocation of the business events.

Using this platform-independent model it is possible to generate three-tier prototype applications. The persistence layer consists of the business object types enriched with all required method types derived from the OET. On top of that an event handling layer is constructed using the OET and the finite state machines. For each event type the user-defined preconditions and state preconditions of the participating object types are checked. If all preconditions are met, the appropriate methods and state transitions are executed. The GUI layer provides access to the events that can be triggered.

3.2 Lessons learned: what worked fine, remaining problems, potential solutions and future work

After introducing our tool in an academic course, we noticed that for students, it is not always clear what's wrong: their model or the code-generator. This is an important lesson: the more we are going to work according to the principles of the model-driven architecture, the more important it will be to ensure the quality of our implementation, and in casu of the model transformations. If the final application doesn't work as expected (either because of bugs or because it doesn't behave as expected) there are two possible sources for the error: either the model (PIM) is wrong, or the transformation rules are not like they should be. To support the process of making a *high-quality* PIM - which can be relatively easily transformed into a PSM and a working prototype application - we enforce the students making *consistent* models. This enforcement is realized by means of advanced (intra-model and horizontal) consistency management techniques like consistency by construction, monitoring, analysis, etc. [5][14].

We also have experienced that managing our transformation rules becomes a crucial issue for a successful MDA story. This should be improved in our future implementations by making better use of standards like MOF and QVT [9].

At the moment we can't generate all information modelled in our PIM. For example mandatory relationships are not enforced in the generated application, and transforming inheritance relationships is still not possible. Transformation rules for inheritance still need to be defined. The use of inheritance in the OET in combination with certain constraints can yield complex models that are very difficult to translate to PSMs in an automated way. Discovering transformation rules that are in general robust to arbitrary combinations of PIM concepts is quite a challenge for the future.

The concept of business events plays an important role in our PIM and should make it possible to define transformation rules to generate other PSM than the one described in this paper. We intend to generate transformations for service-oriented and component-based platforms, because it's already proven that events can also add

value (events as contract, coordination 'tool', etc.) in these sorts of platforms [7][13]. Notice that although the event handler shows a striking similarity with a number of event based architectures [1][8][11], there are substantial differences. For example, the fact that an event is verified prior to the notification to participating domain objects makes it very different from a classical fire-and-forget architecture. For a detailed comparison, the reader is referred to [13].

References

- [1] Barrett, Daniel J., Clarke Lori A., Tarr, Peri L., Wise, Alexander E. (1996), *A Framework for Event-Based Software Integration*, ACM Transactions on Software Engineering and Methodology, 5(4), 1996, pp. 378-421
- [2] Cook S. and Daniels J. (1994) *Designing object systems: object-oriented modeling with Syntropy*, Prentice Hall
- [3] D'Souza D. and Wills A.C (1999): *Objects, Components, and Frameworks with UML, the Catalysis approach*, Addison Wesley, Reading MA
- [4] Gamma E., Helm R., Johnson R. and Vlissides J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [5] Haesen R. and Snoeck M.: *Implementing Consistency Management Techniques for Conceptual Modeling*, accepted for UML2004: 7th conference in the UML series, Lisbon, Portugal, October 10-15, 2004
- [6] Kleppe A., Warmer J., and Bast W.: *MDA Explained: The Model Driven Architecture Practice and Promise*. Addison Wesley, 2003.
- [7] Lemahieu W., Snoeck M., Goethals F., De Backer M., Haesen R., Vandenbulcke J. and Dedene G.: *Coordinating COTS Applications via a Business Event Layer*. IEEE Software, 22(4):28-35, 2005
- [8] Meier R., Cahill V. (2002) *Taxonomy of Distributed Event-Based Programming Systems*. Technical Report, TCD-CS-2002, Dept. of Computer Science, Trinity College Dublin, Ireland
- [9] MOF and QVT, OMG, <http://www.omg.org/mda/specs.htm>
- [10] Robinson K. and Berrisford G.: *Object-oriented SSADM*. Wiley Chichester, 1994
- [11] Shaw M., Garlan D. (1996), *Software architecture: perspectives on an emerging discipline*, Prentice-Hall, N.J., 242 pp.
- [12] Snoeck M. and Dedene G.: *Existence dependency: the key to semantic integrity between structural and behavioural aspects of object types*. IEEE Trans. Software Eng., 24(4):233-251, 1998.
- [13] Snoeck M., Lemahieu W., Goethals F., Dedene G. and Vandenbulcke J. (2004), *Events as Atomic Contracts for Application Integration*. Data and Knowledge Engineering 51, 81-107
- [14] Snoeck M., Michiels C. and Dedene G.: *Consistency by construction: the case of MERODE*, in Jeusfeld, M. A., Pastor, O., (Eds.) *Conceptual Modeling for Novel Application Domains*, ER 2003 Workshops ECOMO, IWCMQ, AOIS, and XSDM, Chicago, IL, USA, October 13, 2003, Proceedings, 2003 XVI, 410 p., Lecture Notes in Computer Science, Volume 2814, pp.105-117