Generation of a Design Class Diagram in Kermeta

Andrés Vignaga

24/10/2006

Generation of a Design Class Diagram in Kermeta
Andrés Vignaga
Published Build date: 24-October-2006

Table of Contents

List of Figures

Introduction

This document describes the development of a transformation implemented in Kermeta. The transformation realizes a design activity proposed in Larman [4] which involves the generation of a Design Class Diagram. Such artifact is a UML class diagram that expresses (part of) the logical structure a system should exhibit in order a particular use case can be executed.

In further chapters the design activity realized by this transformation is described, and the assets which were used or specifically defined for this purpose are presented. In addition, a complete case study is introduced and the result of the application of the transformation to it is shown.

The problem solved by this transformation actually occurs on real projects. The realization reported here may be helpful for other problems, and a reader may gain insight in the approach of transformations based on metamodeling tools.

1.1. Required knowledge

This document describes the development of a transformation which manipulates instances of ECore metamodels, thus minimal knowledge EMOF or ECore is required.

The transformation was implemented in Kermeta, therefore knowledge about the environment and the approach to transformations based on metamodeling tools is necessary.

The transformation implements particular expertise in design activities. It is recommended that the reader should be familiar with the Unified Process artifacts and with Larman's proposal of actitvities.

1.2. Required environment

For using the transformation the following environment is required:

1. Java 1.5
2. Eclipse 3.2
3. Eclipse Modeling Framework (EMF), version 2.2.0
4. Kermeta plugin, version 0.3.0

Please refer to chapter 7 of [5] for instructions on installing and updating a Kermeta environment.

Problem description

2.1. Introduction

In the Unified Process [3], system behavior is captured in the form of use cases. Use cases express the way actors and the system interact in order to fulfill their goals. Larman [4] propose to further express use cases as interactions where the system receives messages from actors, which fire some special operations. These operations are called *system operations*. The design of system operations involves the definition of mechanisms inside the system which realize the expected behavior of each system operation. Such mechanisms are usually expressed as UML interaction diagrams, particularly communication diagrams. Communication diagrams define what objects participate in the mechanisms and what messages they send each other in order to make the mechanisms work. Participants are usually inspired in the concepts that are present in the problem domain and their relations. An abstraction of the problem domain is captured in a UML class diagram called Domain Model.

For the mechanisms depicted in the communication diagrams to actually happen, a complete description of the structure of the participants is required. This description includes the definition of classes with their properties and relations, enabling a configuration of objects which can behave as expressed in the interactions. Such description takes the form of a UML class diagram, and is called a Design Class Diagram.

The purpose of the transformation presented next is the generation of a Design Class Diagram from a number of Communication Diagrams and a Domain Model. Its representation is shown in figure 2.1.

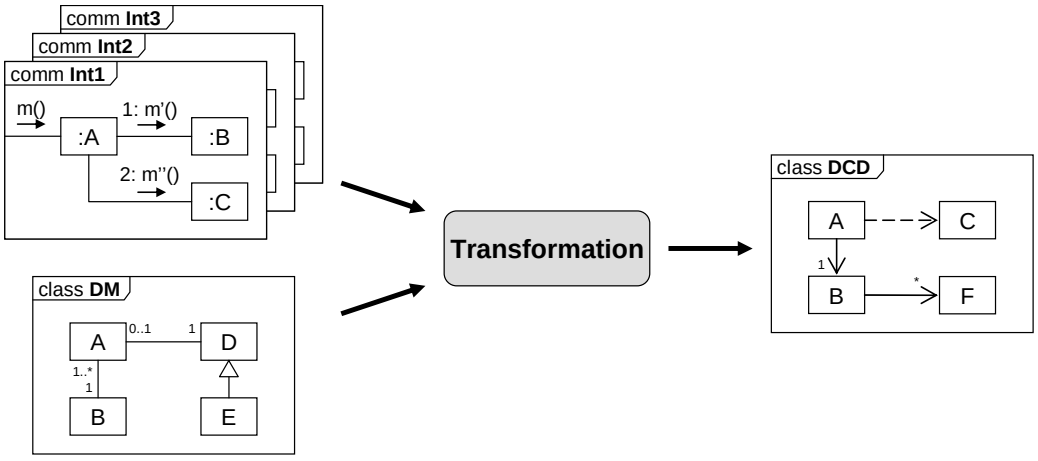

Figure 2.1. Generation of a Design Class Diagram

2.2. Generation of a Design Class Diagram

Larman proposes a high level procedure for generating a Design Class Diagram. This procedure is intended to be manually carried out by a developer. It is presented as a sequence of steps involving the population of an initially empty class diagram with design elements generated from information contained both in the interactions and in the Domain Model.

Classes are created on demand by inspecting the classes of objects participating in the interactions. For example, from the communication diagram Int1 shown in figure 2.1, it can be derived that classes A, B and C need to be created. Attributes for the classes created in the previous step are derived from the Domain Model. In turn, operations are derived from messages occurring in the interactions. For example, again from communication diagram Int1, operation m() is added to class A, operation m'() to class B and operation m''() to class C. Type information, for both attributes and operations, is extracted from the Domain Model and the interactions respectively. Associations between classes are added when visibility by association (i.e. stable) is needed between instances of a set of classes. Navigability of those associations must conform the direction of messages. Finally, dependencies are added when other kinds of visibilities are needed between instances of classes.

The procedure described above is not complete and many details have been omitted. The procedure actually implemented in the transformation is discussed in the next chapter; the fine details can be found in the source code of the transformation.

Logical view

In this chapter the transformation is described from a logical point of view. The overall structure is presented first, and then the main elements are discussed.

3.1. Overall logical structure

When developing a transformation two main ingredients are needed, the metamodels that describe the input and output models, and the transformation itself. To that end, the solution is organized in two main packages: Metamodels and Transformations. The relations between those packages is shown in figure 3.1.

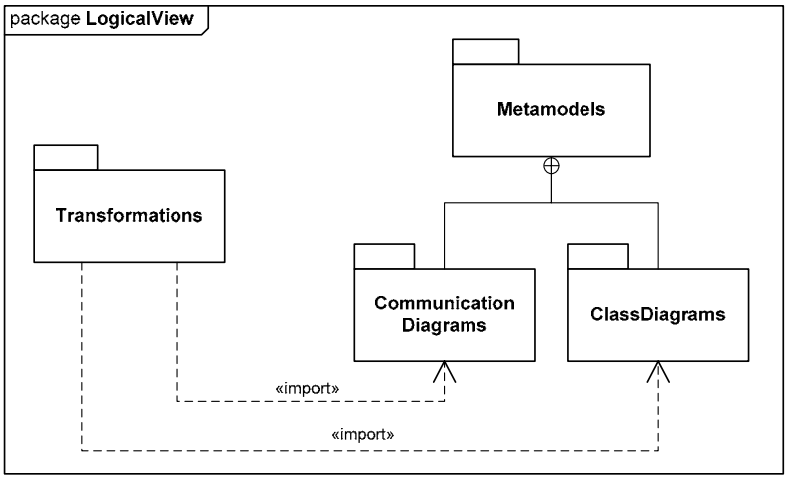

Figure 3.1. Overall logical structure

Metamodels is further organized in CommunicationDiagrams and ClassDiagrams, which contain the metamodel describing interactions, and the metamodel describing class diagrams respectively. Note that the Domain Model and the Design Class Diagram are both class diagrams. The contents of these packages are presented in Section 3.2. In turn, Transformations contains the elements that model the transformation. Since these elements manipulate models, access to their metamodels is needed. The contents of this package are presented in Section 3.3.

3.2. Metamodels

The metamodels presented next were specifically defined for this transformation. Since input and output models are UML models, a part of the UML metamodel could be used. For convenience, a simplified and tailored version is used here instead.

3.2.1. Class diagrams

Figure 3.2 shows the metamodel for class diagrams. A class diagram may contain classes and data types, both with typed attributes. Attribute types may be data types only. Classes may also contain operations. Both the return type and parameter types may be any type (class or data type). A class may be abstract and may have at most one superclass. Associations are binary and may relate classes only. A class may depend on an arbitrary number of suppliers.

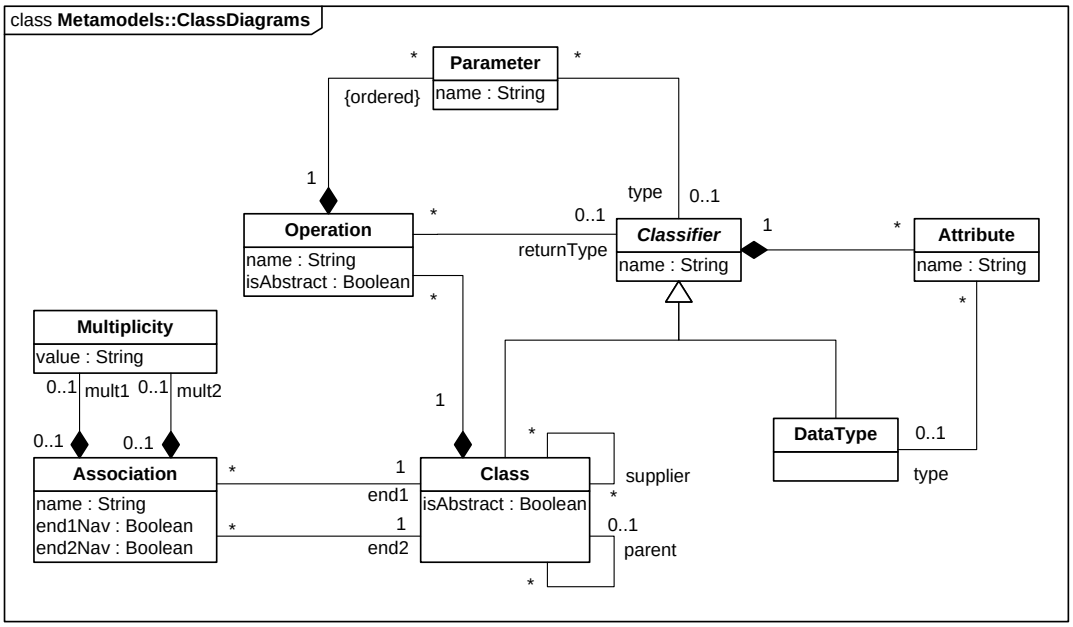

Figure 3.2. Metmodel for class diagrams

A Domain Model is an analysis artifact, therefore design elements such as operations and dependencies are not used. Moreover, associations are bidirectional.

A metaclass ClassDiagram owning a set of classifiers and a set of associations is left implicit.

3.2.2. Communication diagrams

Figure 3.3 shows the metamodel for communication diagrams. A communication diagram may contain objects and multiobjects. Objects may receive and send messages to other participants. Multiobjects are elements representing containers of objects and may receive messages only; the semantics of such messages, which are collection manipulation primitives, is assumed understood. Participants of interactions are typed. In turn, messages are sent from one participant to another, with the exception of the entry point of an interaction which comes “from nowhere”. A message has a sequence number and may return a typed value. A message may have typed arguments as well. The nature of the link between the source and the destination of a message may be specified in its visibility property. To that end, the VisibilityKind enumeration was defined. A message may be annotated with information specifying whether the destination object alone is capable of fulfilling its expected behavior. This is useful for deciding if a class provides a method for a given operation.

A metaclass CommunicationDiagram owning a set of participants and a set of messages is left implicit.

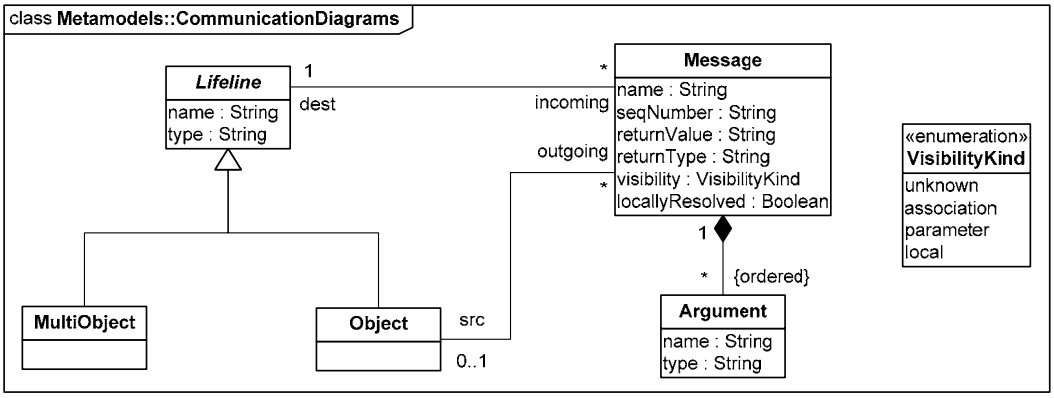

Figure 3.3. Metamodel for communication diagrams

3.3. Transformation

In this section the contents of package Transformations, and thus the transformation is described. The description includes both structure and behavior.

3.3.1. Static structure

The structure of the transformation is shown in figure 3.4. Classes CommunicationDiagram and ClassDiagram, and relations to them, traces the import dependencies shown in figure 3.1.

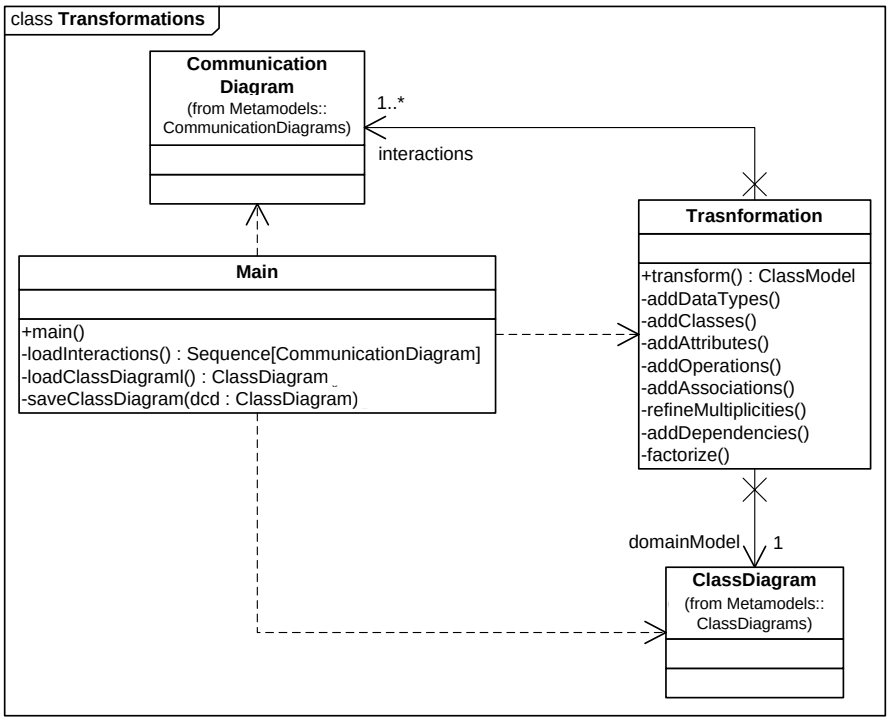

Figure 3.4. Static structure of the transformation

Class Main owns the entry point of the transformation: operation main(). An instance of this class is responsible for loading the input models and starting the transformation process. The transformation itself is performed by operation transform() of class Transformation, called within main(). The result of transform() is the Design Class Diagram, which is finally saved. The behavior of the transformation is discussed next.

3.3.2. Behavior

Figure 3.5 shows an interaction depicting the behavior of the transformation. An instance of class Main, after loading the input models, creates an instance of class Transformation and passes those models to it. The result of message transform() is the Design Class Diagram to be saved as the result of the transformation.

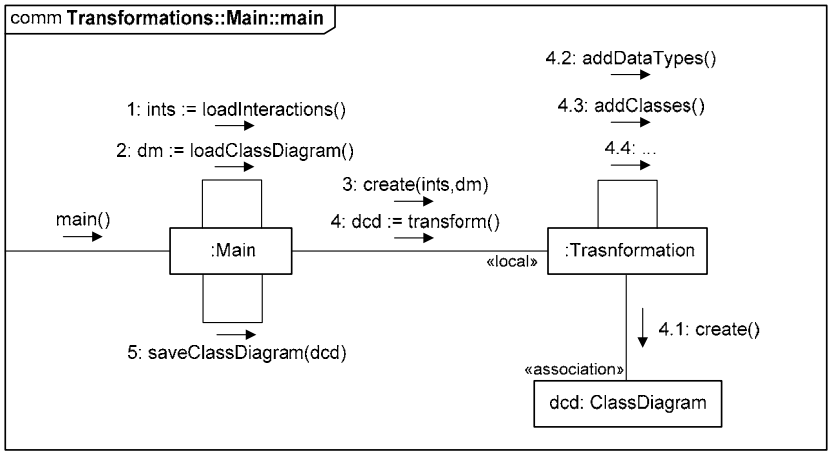

Figure 3.5. Interaction of the transformation

When transform() is invoked, an instance of class Transformation creates an empty class diagram which is populated in the sequence of automessages starting at message 4.2. These messages correspond to the operations of class Transformation declared as private in figure 3.4, and are processed sequentially as shown in figure 3.6.

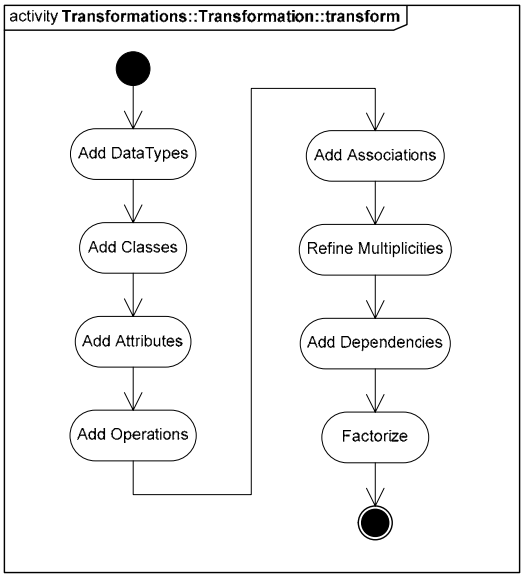

Figure 3.6. Activities of the transformation

Activities in figure 3.6 match the steps of the procedure that realizes the transformation. As an example, figure 3.7 expands the activities involved in the last step; the factorization of the DCD.

> **Note**
>
> Please refer to the source code of the transformation for further details on the activities of figure 3.6.

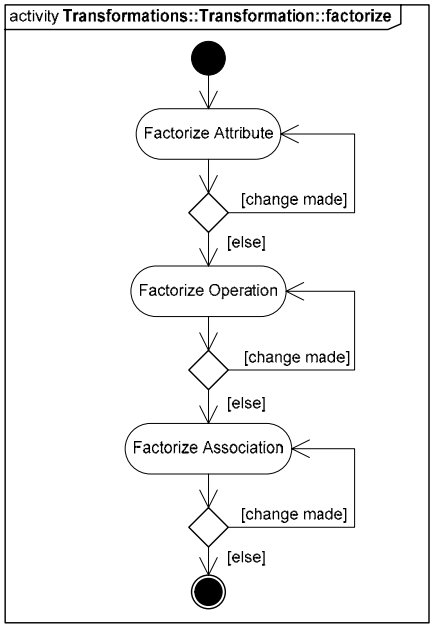

Figure 3.7. Activities of factorization

At every step, the instance of class Transformation reads the input models and the model under construction, creates the required elements, and writes them the Design Class Diagram, as shown in figure 3.8. This can be understood as a combination of *Pipes & Filters* and *Blackboard* architectural patterns.

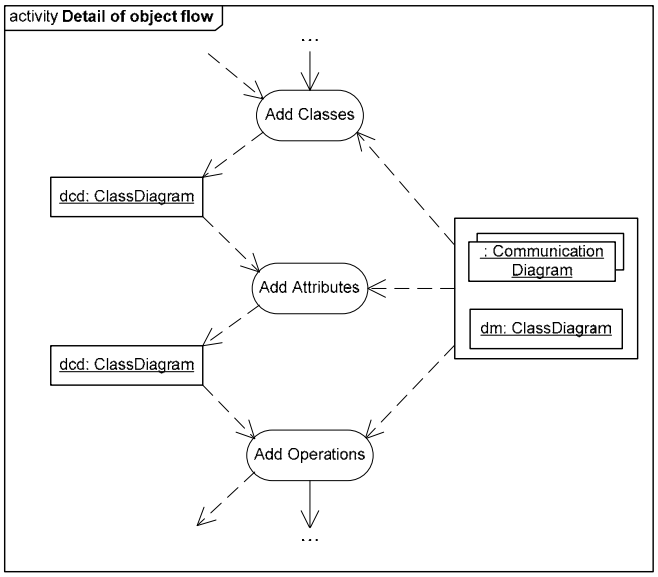

Figure 3.8. Detail of object flow

Implementation view

In this chapter the transformation is described from an implementation point of view.

4.1. Overall implementation structure

The structure of the implementation is analogous to the structure discussed for the logical view. Three main packages Metamodels, Transformations and Models are defined, as shown in figure 4.1. A fourth package Utility was also defined, but is omitted here for clarity.

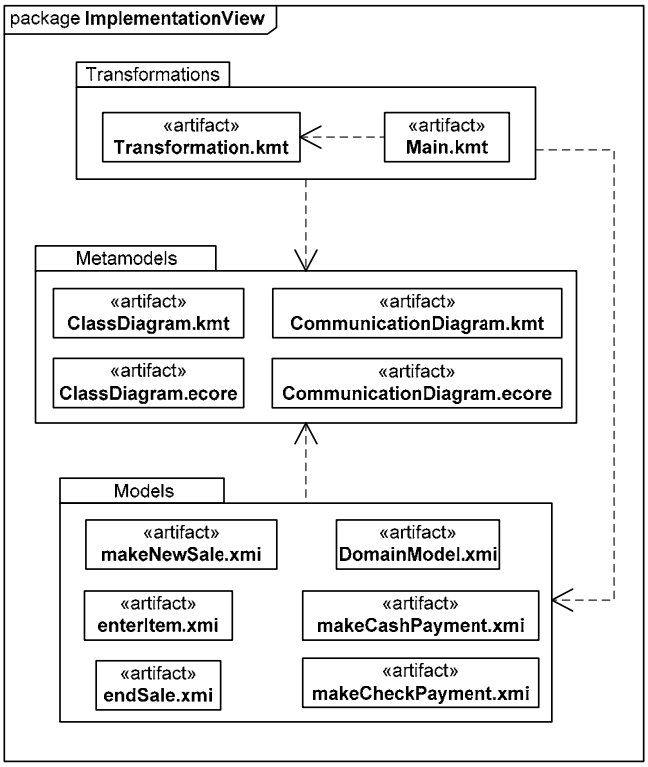

Figure 4.1. Overall implementation structure

Packages Metamodels and Transformations contain the implementation of the design discussed in the previous chapter. In turn package Models contains both input and output models. In figure 4.1, the models shown correspond to those in the case study presented in the next chapter.

4.2. Implementation packages

In this section the contents of the implementation packages just presented are discussed.

4.2.1. Metamodels

The Metamodels package contains an ECore file specifying each metamodel presented in Chapter 3. Please refer to Chapter 2 in [1] for instructions on creation of EMF metamodels for Kermeta programs.

ECore metamodels include structure only, therefore the ECore version of the metamodels exactly match those discussed in sections 3.2.1 and 3.2.2. However, a Kermeta version of those metamodels may be used instead. The main benefit of the latter approach is that metamodels expressed in Kermeta syntax may include behavior as well. For example, the equality of *operations* involves not only checking the names of the operations to be compared, but also matching the types of their parameters, if present. If an ECore version of the metamodel was used, then such operation must be defined in a utility class; however, using a Kermeta version allows for defining the operation in the Operation metaclass, which is more appropriate. In this way, the metamodels can be enriched with behavior when needed. Kermeta versions, including structural features only, were automatically generated from ECore versions using the transformation Ecore2Kermeta included in the Kermeta environment. Behavior was manually included afterwards.

The Kermeta version of the metamodels did not replace the ECore versions. These were still needed for loading and saving (serializing and deserializing) models. The Kermeta versions were used for model manipulation. For further details on adding behavior to metamodels and the Ecore2Kermeta transformation please refer to Chapter 5 of [1].

> **Tip**
>
> When using ECore and Kermeta version of metamodels simultaneously package names in both files must match, and Kermeta files need only to be “required”, otherwise conflict names will arise.

4.2.2. Transformations

Classes in the Transformation package directly map the specification discussed in Section 3.3. Please refer to Chapter 4 of [1] for further details on model manipulation in Kermeta, to [2] for a specification of Kermeta constructs, and to [5] for a description of the Kermeta development environment.

4.2.3. Models

Models are the input of the transformation and thus specific to particular development projects. They must be expressed in XMI format, and must conform to their respective metamodel. For details on creating models from Ecore metamodels please refer to [1], Chapter 3.

> **Tip**
>
> In models, many pieces of information are optional. However, input models which are more complete usually lead to more accurate Design Class Diagrams.

Sample models

In this chapter the functionality of the transformation is demonstrated by its application to a concrete case study. The problem refers to a system for a Point-of-Sale register of a retail store. The complete specification of this case study can be found in [4]. The use case treated is *Process Sale*, which deals with customers buying products at the store. The Domain Model is presented first. The main success scenario for the use case is then illustrated. The interaction for each system operation comes next, and finally the resulting Design Class Diagram is shown.

5.1. Domain model

Figure 5.1 shows the Domain Model for the POS system, focusing on the concepts involved in the processing of a sale.

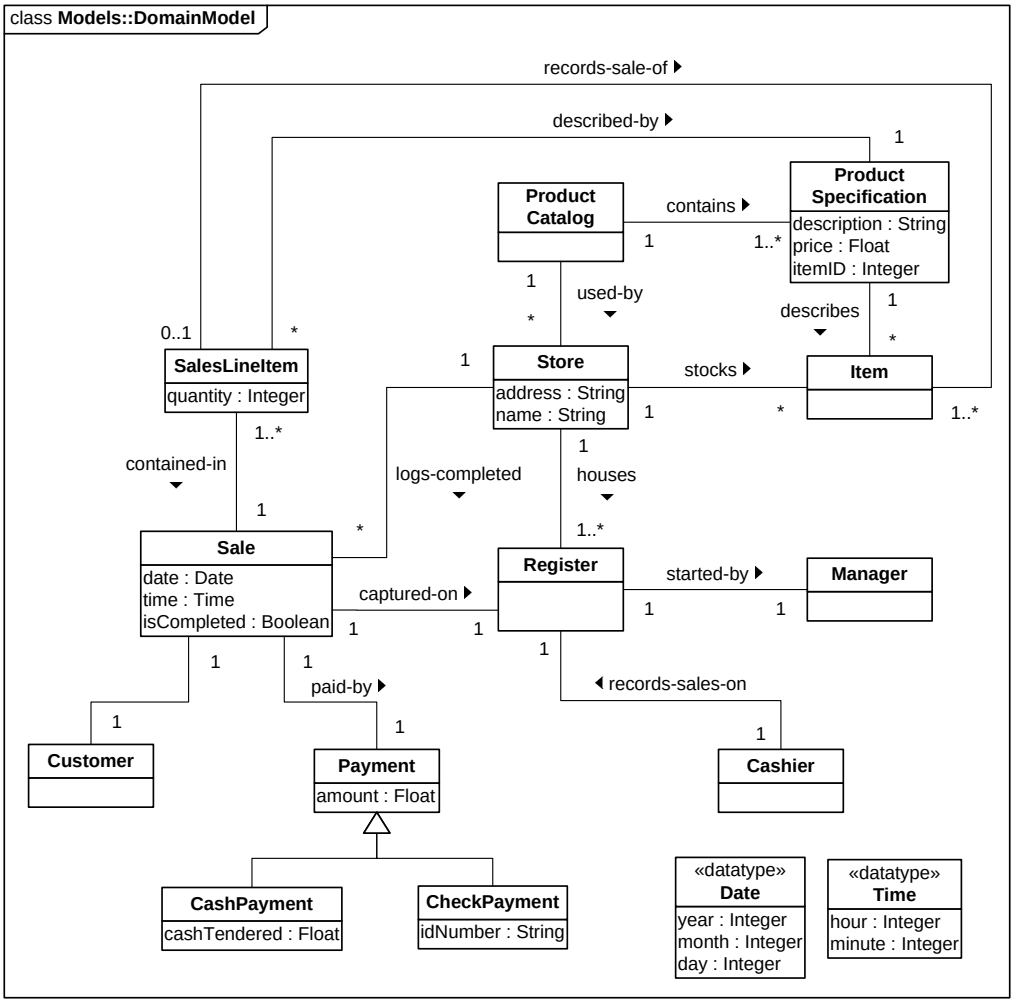

Figure 5.1. POS Domain Model

5.2. Main success scenario

In this section the main success scenario for the Process Sale use case is described by means of a System Sequence Diagram. Such diagram shows the interaction involved in the use case between actors and the system as a black box. It introduces the system operations identified for Process Sale, and specifies the order in which they are applied on the system. A System Sequence Diagram helps visualizing the flow of events which occur in a particular scenario of a use case, where events trigger system operations. The design of system operations is shown in section 5.3.

> **Important**
>
> **System Sequence Diagrams are not used by the transformation in any way. An SSD sets the context for the system operations and is included in this description for clarity reasons only.**

Figure 5.2 shows a sequence diagram for the main success scenario of the Process Sale use case (i.e. a System Sequence Diagram). Participants of the interaction are the System as a black box and a single instance of an actor: a cashier.

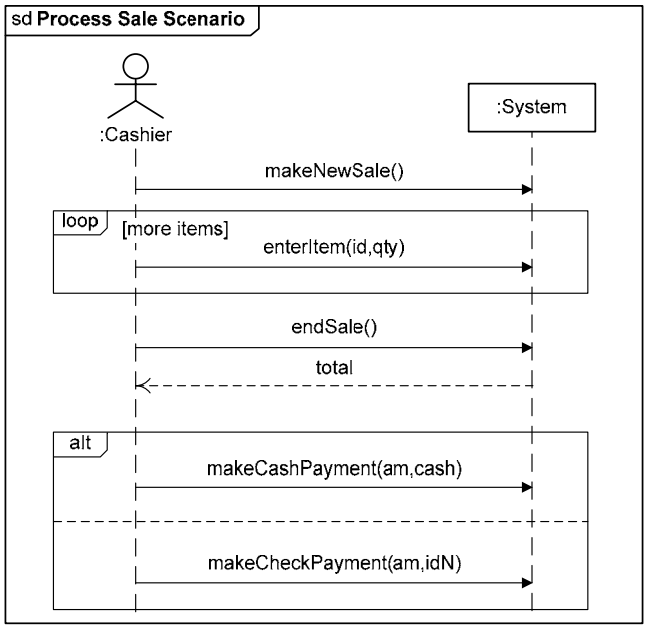

Figure 5.2. Main success scenario for Process Sale

When the use case starts the system receives a makeNewSale message and gets ready for accepting and recording items. An arbitrary number of items are entered via multiple receptions of enterItem message. After the last item is entered the system receives an endSale message and stops accepting items for the current sale. The system may handle either cash and check payments. The payment method is chosen by the customer. The system then receives a makeCashPayment or makeCheckPayment message respectively, and the use case is done.

5.3. Interactions

In this section the interactions for the system operations involved in the Process Sale use case are presented.

When starting a new sale, the register creates a new instance of Sale, which in turn creates an empty collection of sales line items. This is shown in figure 5.2.

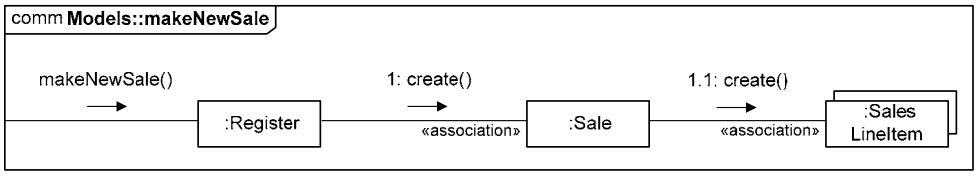

Figure 5.3. Interaction for makeNewSale()

Every time an item is entered, the register looks for its specification and passes it to the sale in process, which creates a new line for it and records it in the line collection. Figure 5.3 shows this interaction.

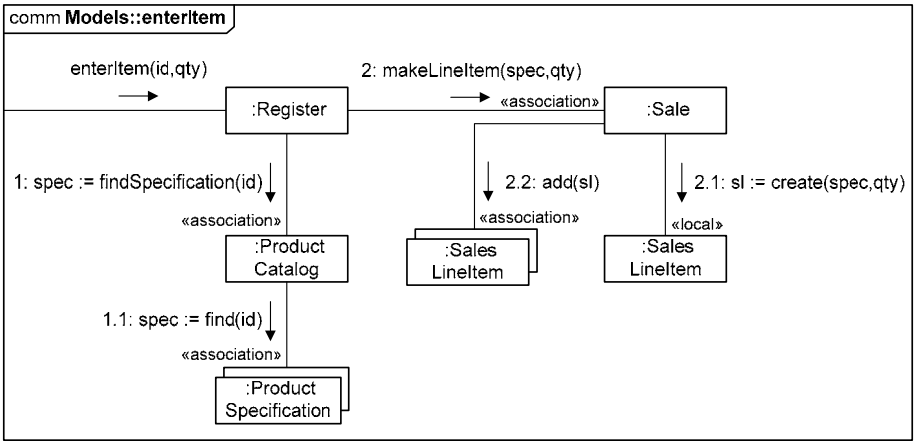

Figure 5.4. Interaction for enterItem()

After the last item is entered the register is asked to end the sale as shown in figure 5.4. The register first notifies the current sale that the purchase is complete and then asks its total for display. The sale iterates over its line items, collecting the subtotal of each, and returns the result.

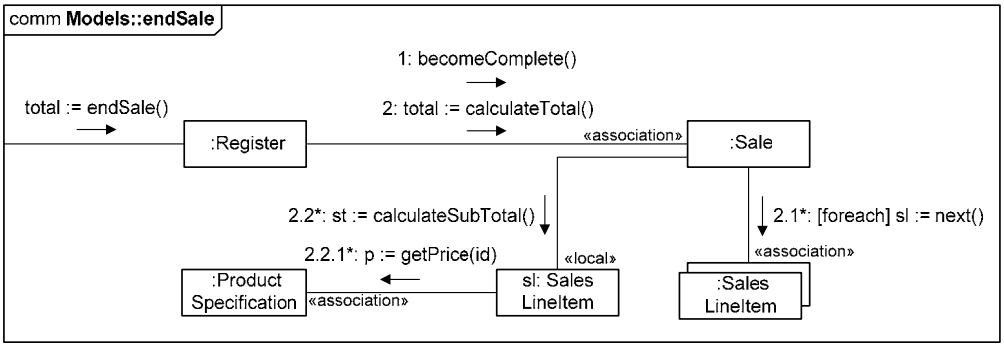

Figure 5.5. Interaction for endSale()

The system supports two different paying methods, where different information is recorded. For finishing the use case, the register is told which paying method applies. The register passes the information to the sale, which creates the appropriate variant of payment. In either case, the register passes the sale to the store which is responsible for recording it. Figures 5.5. and 5.6 show the interactions for both variants of payment handling.

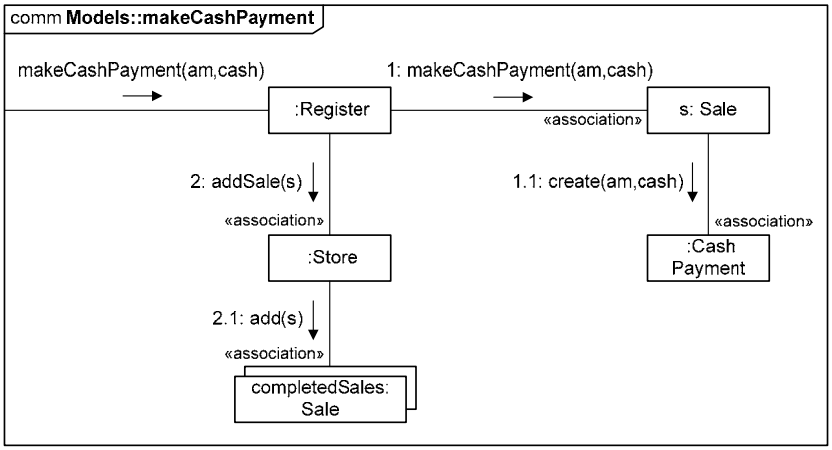

Figure 5.6. Interaction for makeCashPayment()

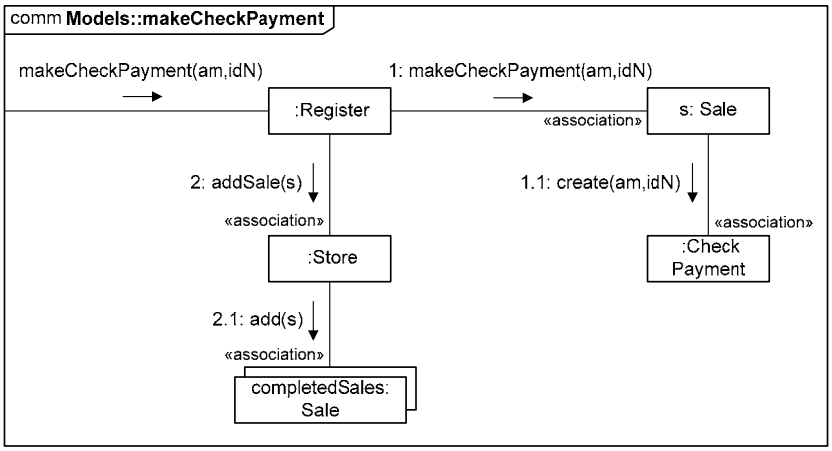

Figure 5.7. Interaction for makeCheckPayment()

5.4. Design class diagram

When the transformation is executed on the models presented above, a Design Class Diagram is generated. The file containing the result is `DesignClasDiagram.xmi` and is saved in the Models package.

Figure 5.7 shows a graphical representation of the result of the transformation, which is in fact similar to that presented in the original case study in [4].

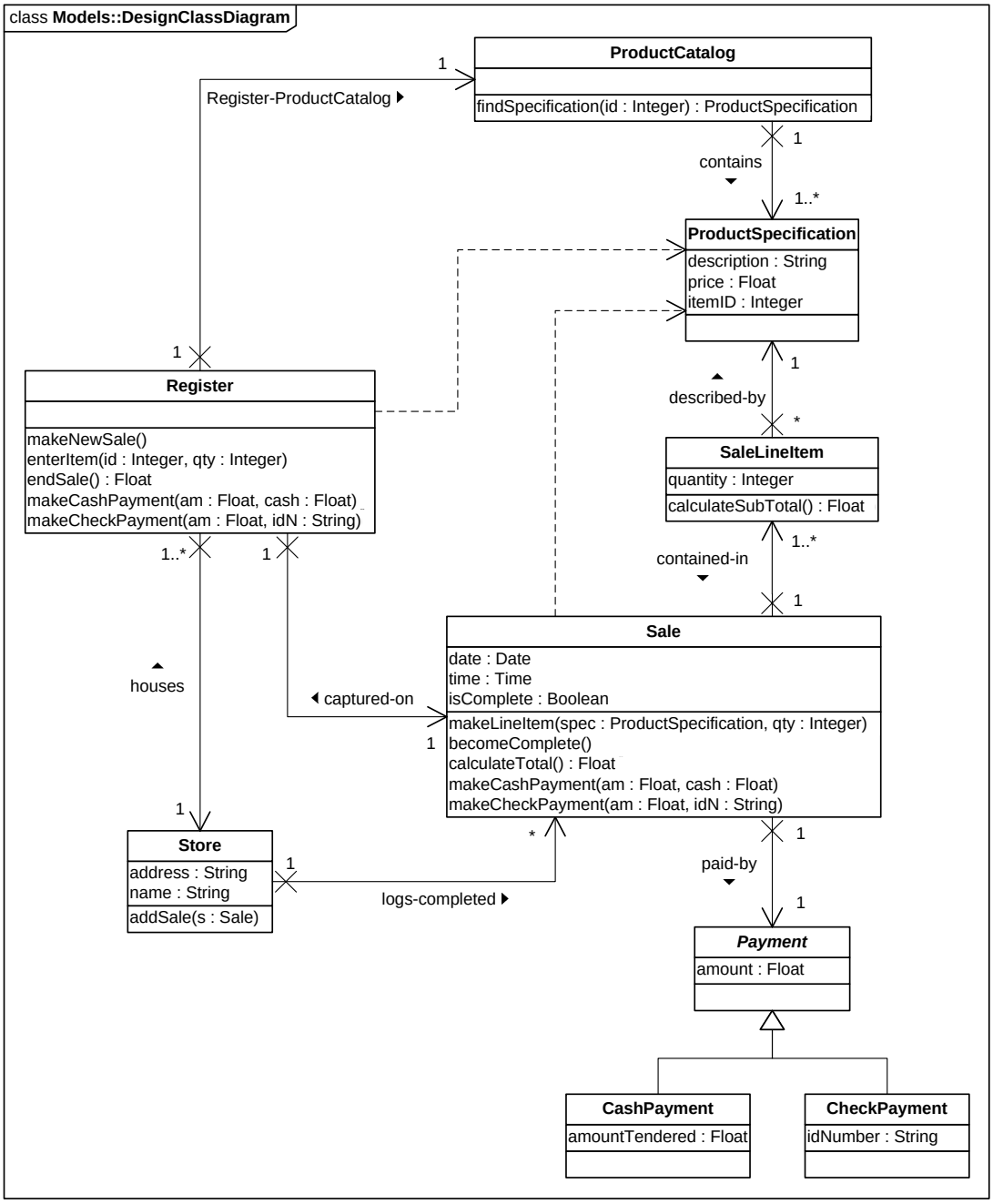

Figure 5.8. Design Class Diagram resulting from transformation

References

[1] Z. Drey, D. Vojtisek. How to create an EMF model and use it in Kermeta. Internet: `http://www.kermeta.org/docs/KermetaEMFTutorial.pdf`, 2006.

[2] F. Fleurey, Z. Drey, D. Vojtisek, C. Faucher. Kermeta Language, Reference Manual. Internet: `http://www.kermeta.org/docs/KerMeta-Manual.pdf`, 2006.

[3] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley Professional, 1999.

[4] C. Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3rd Edition, Prentice Hall, 2006.

[5] D. Vojtisek. *How the workbench is able to help you in your metamodelling tasks.* Internet: `http://www.kermeta.org/docs/KerMeta-UI-UserGuide.pdf`, 2006.