Implementing Model-Based Data Structures using Transient Model Extensions

Michael Thonhauser[1,2], Gernot Schmoelzer[2] and Christian Kreiner[1,2]
[1]Institute of Technical Informatics, Graz University of Technology, Austria
[2]Salomon Automation Gmbh, Friesach bei Graz, Austria
Email: michael.thonhauser@tugraz.at, gernot.schmoelzer@salomon.at, kreiner@tugraz.at

***Abstract*— Software is often constructed using a layered approach to encapsulate various functionality in corresponding layers. Individual requirements of each layer demand layer specific data structures. These data structures typically provide redundant information with respect to the data source.**

Providing a Model Driven Software Development approach for creating these data structures leads to overlapping data models, each containing data structures defined by the data source. Because putting all various requirements of the software layers in a single data model can lead to difficulties, each software layer should only extend the basic data source model with its specifically needed model elements.

The approach presented in this paper applies a mechanism for a dynamic extension of a data model. This extension mechanism is used in the implementational activity of a software process, and allows the changing of a model within a local scope. Using this mechanism, a basic data model can be used by every layer, being extended by additional attributes and classes for satisfying layer specific requirements.

***Index Terms*— Model-driven development, Data modeling, Data Intensive Systems, Software layers**

I. Introduction

In the last years their has been a lot of research on model driven development (MDD) [1]–[3], which has lead to different standards like Unified Modeling Language (UML) [4] and approaches like Model Driven Architecture (MDA) [5]. The aim of an MDD approach is the description of software in an abstract way by making use of a model describing the designed software. This model specifies attributes of the software, which are needed in the corresponding activity of a software engineering process.

There exist different software engineering process models, like the waterfall model, evolutionary development, formal systems development or iterative approaches such as the spiral model [6]. All of these process models define some fundamental activities, like software specification, software design and implementation, software validation and software evolution. Each activity can be supported by models. Some modeling standards like UML also provide different views of a model (e.g. class diagram, use case diagram, sequence diagram), which are best suited for different acitivities in a software engineering process.

While MDD aproaches often require to finish the specification of models before the modeled applications are implemented, agile approaches like Agile Model Driven Development (AMDD) [7] focus only on the view of a model currently needed. In an AMDD approach for developing a data-intensive system a data model is the most important type of model created at the beginning of the software development process. Because data-intensive applications are software systems that focus on data processing, data visualization and data storage (such as enterprise resource planning systems, banking applications or logistic systems), a data model contains the description of persistent data structures. These data structures serve as the basis of a data-intensive system. To ease the development of data-intensive applications a layered approach is typically chosen for such architectures [8] consisting of three layers, which are called presentation layer, business function layer and data source layer.

Data structures used by a software layer are described in a data model. Because every layer has different requirements on its data structures, data models of various layers may differ, but there exists a common partial data model of all layer specific data models. This common partial data model can be seen as the application's minimalistic data model.

Supposing that this minimalistic data model contains all data required for persistence, data models of the different layers can be produced by adding layer specific data structures to this model. While the minimalistic data model needs to be defined in the first activity of a software engineering process, layer specific additions can be defined in later activities (e.g. during implementation of the corresponding layer). Some requirements on layer specific data structures are not known in the design phase of a layer, because they follow from implementational considerations. In order to support these requirements by a data model a dynamical model extension mechanism has been proposed in [9]. This mechanism is called Transient Model Extension (TME) to point out, that extensions made to the model are available only in the scope (e.g. layer, class method) where they are defined.

This paper is based on "Model-Based Data Processing with Transient Model Extensions," by M.Thonhauser, G.Schmoelzer and C.Kreiner, which appeared in the Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, Tucson, AZ, USA, March 2007. © 2007 IEEE.

For demonstration purposes of the TME mechanism a warehouse management system is introduced in Sect. II giving an example of a layered design for a data-intensive system. Sect. III discusses the dependencies of layered software and data models in more detail, while Sect. IV introduces the TME approach. Sect. V describes an implementation scenario of the examplary WMS using the TME approach.

II. Motivating Example

We begin by introducing a simplified warehouse management system (WMS), which is implemented in the business domain of logistics. The example system supports the process of managing transport unit items (TUItem) being stored in a warehouse. Each TUItem is an instance of a specific article and is contained in a transport unit (TU).

Our warehouse management process uses three stages; at the incoming goods stage TUs are received from suppliers. In the second stage received TUs are stored in a high rack. If a customer orders an article a TU containing a corresponding TUItem is looked up in system. According to the strategy (e.g. best-before date, fastest available TUItem), the found TU is delivered to the third stage of our warehouse, the goods issue. In this stage the ordered TUItems are collected together and stored in another TU. While the filled up TU is delivered to the customer, the original TUs are returned to the high rack stage.

Fig. 1 depicts the modules of the described WMS. Note that each stage of the warehouse management process is associated with one module of the system. Each module is constructed using a layered approach.

The chosen approach consists of three layers. A two layered approach would also be feasible, which would require to split up the functionality of the business layer. This approach is often realized using fourth generation programming languages, like SQL, to realize business functions with stored procedures and aggregate functions. The drawback of this approach is the complexity of the realized queries.

Using a three layered approach queries can be transfered to the business logic layer. They can be split up in programming statements, which rely on smaller queries being responsible for retrieving the required data from the data-source. Often this approach is related to usage of an object-relational mapper in the data source layer, which maps the existing objects onto relational database tables.

Because the majority of currently used programming languages for constructing such systems, such as C#, Java or C++, follows the object oriented paradigm, data structures can be described using a class diagram view of the data model.

III. Related work

A. Layered software

In a layered approach such as proposed in [8] each software layer internally uses data structures, holding

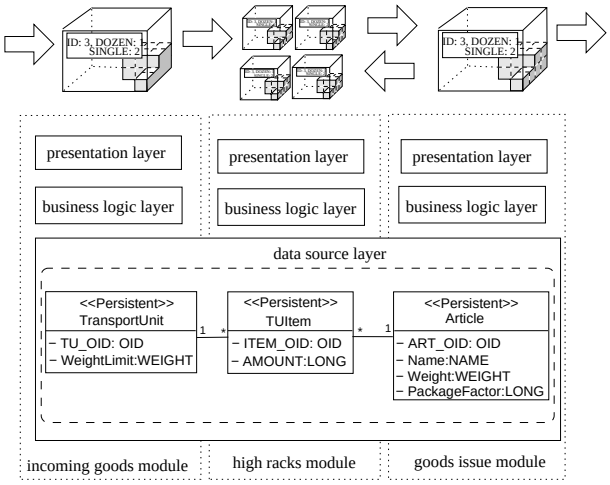

Figure 1. Software layers of warehouse management system

the needed data of this layer. Fig. 2 shows these data structures and depicts the dependencies between them. Every data structure can be seen as a layer specific view on the information generally available to the software. Data structure $S1$ holds the transformed persistent data, which is compliant to the structure of the data source. $S2$ repesents the data structure used by the domain layer and $S3$ is contained in the presentation layer.

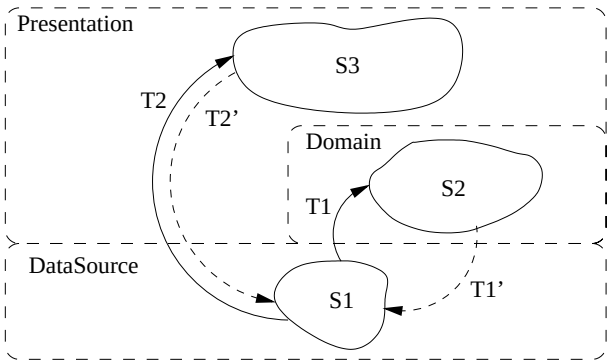

Figure 2. Data structures in software layers

The examplary WMS illustrates different requirements for data structures in various application layers. Requirements for $S1$ are the reduction of redundant information, thus providing support for *data normalization* [7]. This reduction is needed for optimal storage usage, efficient data transfer and to avoid inconistency of stored data. Also the data structure should be easily mappable to the persistent data source.

$S2$ is driven by requirements for *simplifying complexity* of the application logic, adding *transaction handling* and *data consistency checks*. This aims at avoidance of inconsistent data and is done through *adding data redundancy*.

$S3$ contains the data displayed at the presentation layer. Its content meets user requirements and *usability aspects*, which often requires displaying additional information.

As illustraded in Fig. 2 there exist several dependencies

of the data structures in the different layers. Because $S1$ holds all the persistent information available to the application, additional data in $S2$ and $S3$ is normally related to data contained in $S1$. This requirement can be fulfilled by extending $S1$ with a transformation $T1$ or $T2$.

The way this transformation is performed depends on the abstraction level used for application development.

- In a traditional software engineering approach implementation is done by using a relational database and fourth generation programming languages like SQL. This programming language type often includes a database manipulation language (DML) used to define the data structures $S1$, $S2$ and $S3$. In this case the transformation $T1$ and $T2$ of the data structures is done by extending the DML statements.
- A higher level of abstractions is provided by object oriented languages combined with object-relational methods [8]. This solution enables type safety for $S1$, $S2$ and $S3$ consisting of classes. The transformation is done with additional source code for the implementation of $S2$ and $S3$.
- A further abstraction level is reached by using a MDD approach. This leads to a corresponding data model $DM1$ for $S1$, $DM2$ for $S2$ and $DM3$ for $S3$. Following the dependencies of the different data structures, the dependencies of the data models can be seen as $DM1 \subseteq DM2$ and $DM1 \subseteq DM3$. In this case transformation $T1$ or $T2$ is done by extending $DM1$. This extension adds information about additional data needed in the presentation layer leading to $DM3$ and to $DM2$ respectivly in the domain layer.

  Note that this mechanism is called Transient Model Extension (TME), because the extensions to $DM1$ are only visible at the scope where they are applied. There exist also many small extensions for $T1$ and $T2$. Since $DM1$ is used as the basic model for this mechanism, $DM2$ and $DM3$ does not need to be stored in the corresponding layers. Only the transformation rules needed for construction of the appropriate data model need to be saved instead.

B. Model Driven Development

Model Driven Development (MDD) [1] is an approach to implement a software system by describing it with a Platform Independent Model (PIM). A PIM defines associations between data and behavior of the software and it is used as input for generators producing a platform specific model (PSM). To support MDD the Object Management Group (OMG) has released the Model-Driven Architecture (MDA) containing standards, which enable specification and transformation of models. Another approach to MDD are Software Factories, which are proposed by Microsoft and can be seen as a new software development paradigm. Differences of these two approaches are discussed in [10].

Models of software design are often specified using Unified Modeling Language (UML) [11], another standard of the OMG. UML models are based on a metamodel and are situated in the user model layer of the four-level metamodel hierarchy [4]. UML describes several diagrams, which can be used to model different aspects of a software. Structural and behavioral diagrams are differentiated. One example of a structural diagram is the class diagram. It is used to model the structure of classes, such as attributes and methods, as well as associations between different classes in the model.

For data modeling purposes the metamodel of a class diagram can be extended focusing only on class attributes and associations [7].

Agile Data Modeling relies on iterative construction of data models, where each data model satisfies the requirements needed in the current iteration. It is best suitable for applications that rely on relational databases for persistent data storage. Agile Model Driven Development (AMDD) also uses an iterative approach, instead of extensive models being generated in the regular MDD process.

C. Related persistent data frameworks

To realize the mapping of persistent data in dynamic data structures several widely known technologies exist.

Microsoft's **ADO.NET** framework [12] contains several classes, which enable the usage of relational databases or XML files as persistent data sources. Access to the database is provided through an instance of a DataReader or a DataAdapter. While the first one is only used for reading data from a database, the second allows also data manipulation independent of the database type. The data adapter is used by a dataset component, which is an in-memory database.

The dataset is used by other application layers, but it does not utilize a data model. Therefore extending the dataset is done at source code level instead of the model abstraction level.

Hibernate [13] aims at providing an object/relational bridge. It allows the user to work with object oriented concepts like inheritance and composition as well as with database relational concepts such as usage of a DML like SQL. The mapping between the Java classes and the database structure is done by a XML file.

Because Hibernate works at the source code abstraction level, supporting model based extensions is not in the scope of this technology.

The **Eclipse Modeling Framework** (EMF) [14] is an implementation related to the OMG Meta Object Framework (MOF). A EMF model is based on a metamodel called ecore model. This model defines the content of eAttributes and eReferences, which belong to the eClass elements in the model.

EMF models can be built from Java files, XML files or UML models stored in XMI. These models can be used as input for Java source code generators, producing class files with annotated source code. This code can be edited manually to add functionality. EMF supports the

serialization of objects in XMI files, if they are based on an EMF model.

The ecore model defines attributes allowing to control which elements of the model can be serialized. The transient flag defines whether the corresponding element can be serialized. The volatile flag is used to signal that the value of this element depends on the value of other model elements.

Our approach differs in the following points from the EMF modeling concept:

1) The first aspect is the support of simultanous but independent extensions of the persistency data model by different methods. In our understanding the same persistent data model is used by all extending models, while the extensions are only visible within a particular scope. In contrast, extensions to EMF models are globally visible.
2) The second aspect is the used concept of model extension. While EMF supports extension of classes with subclassing an existent class in a model, our approach directly adds attributes and references to existing classes. The advantage hereby is the constant class type of the extended class, so no modification of code expecting a particular class type is required.

D. Partial model techniques

While many model based applications rely on large monolithic models, there exist alternatives in the domain of Domain Specifc Languages (DSL) [15], which rely on the management of multiple partial models. These partial models can be linked together to create one application specific model or can be used for partial source code generation, which can be linked.

Another approach based on metamodels is descriped in [16]. This approach uses core models and fragment models, which conform to the same metamodel. These models are then linked together following a formal definition, which ensures that the resulting model is also conform to the metamodel.

In our approach of transient model extension used models are also expected to be conform to their metamodels. But instead of linking these partial models, our approach allows the specification of model extensions for a specific model in other forms, e.g. in a programming language. Another distinction is the corresponding acitivity in the software process used for creating the extension of models or performing the linkage of partial models. Often partial models are defined during design activities and are often linked before starting with implementational activities in the software development process. This can be done for enabling generators to produce corresponding application artifacts (like source code or database setup scripts) [17]. In contrast the TME approach is applied during implementational activities of a software development process to dynamically manipulate the model, where the manipulations have a local scope. Looking at the different roles in a software process, model linking approaches are applied by application and database designers, while our approach is used for supporting the software developer in implementing the layers.

IV. Transient Model Extension

The structure of a data model can be based on the four-level metamodel hierarchy of the OMG, where level M3 (meta-metamodel) is the same for each data model. Level M2 contains the general metamodel and additional metamodel extensions, allowing the specification of domain specific data models in M1. These domain specific data models maintain the domain specific classes and associations being used for construction of objects which correspond to model level M0. Generally speaking, every model can be seen as an instance of the model in the lower layer. Considering this fact, it is obvious that changing the model in level M1 results in additional data contained in level M0.

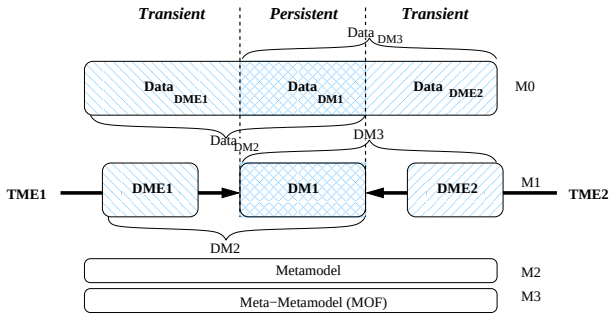

Figure 3. Data model structure

Fig. 3 illustrates the elements needed in the four-level metamodel hierarchy for realization of the data structures defined in Fig. 2. According to this figure $DM1$ is part of every data model defined in level M1. $DM1$ defines the data structure $S1$, which contains $Data_{DM1}$ in M0. In Fig. 2 mappings $T1$ and $T2$ are defined for creating the corresponding data structures $S2$ and $S3$. Mapping $T1$ is represented by $TME1$ and $T2$ is represented by $TME2$ respectivly. Each of these mappings defines a Data model extension (DME) of M1.

According to the assumptions made in Sect. III-A, $DM2$ is equivalent to $S2$ and is defined as a combination of $DM1$ and $DME1$. In the same way the data model $DM3$ results from combining $DM1$ with $DME2$. Every model has corresponding data structures and data in M0.

As stated in Sect. III-A, $DM1$ is used by the data source layer, $DM2$ by the business function layer, and $DM3$ by the presentation layer.

Applying the AMDD approach to a layered software architecture results in the definition of a data model containing the data structures used by the data source layer. Based on the persistent data model the corresponding data models for the domain and the presentation layer are defined. According to Fig. 3 a separate definition of each data model used by the different software layers is not feasible. Common data definitions needed by different

layers are duplicated in the models, leading to maintenance difficulties of the shared information. Also each layer needs to store its own data model, which can lead to increasing storage requirements by the application. To overcome these problems a single data model is defined in the data source layer, containing the persistent data structures.

To get a model for a software layer other than the data source layer, its data model normally can be extended by a mechanism called Transient Model Extension (TME). It is illustrated by arrows $T1$ and $T2$ between the data structures in Fig. 2 and $TME1$ and $TME2$ in Fig.3 respectivly.

Note that the extensions provided to the data model can be applied for different scopes. As stated above, an extension of the model can be defined and used by all classes being part of a layer. Another possibility is illustrated in Fig. 4; the business logic layer contains two methods, which apply a TME to the data model referenced from the data source layer. In this case each TME is only visible to the defining method.

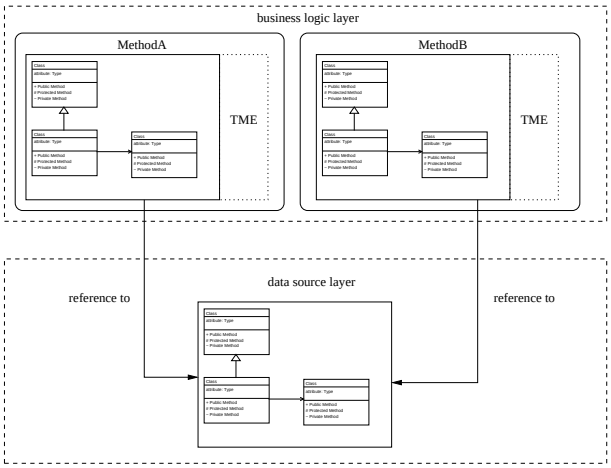

Figure 4. TME with method scope

A. TME types

As extensions of data structures implemented in an object-oriented language need to follow the rules of the language compiler, extensions of the model have to be compliant to its metamodel. According to this fact, an extension of the model element needs to be an instance of an element defined in the metamodel.

The following elements of a class metamodel are used in a TME mechanism for data-models.

1) Attributes: Extending a class with additional attributes is done to model additional information, that belongs to this class. Often this information can be derived from other class attributes or (attributes of) associated classes. The type of an extended attribute has to be defined in the model before the extended attribute is defined.

Fig. 5 demonstrates an example of a TME with an attribute for the datamodel of our WMS defined in Sect. II. For retrieving the weight of a TU the sum of the weight of all TUItems has to be calculated. The weight of a TUItem again depends on the weight of the corresponding article and the amount of items located in the TU. Dynamically adding the weight attribute to the classes *TransportUnit* and *TUItem* allows to divide the calculation descriped above. Also the result of this calculation has the same metainformation as the weight attribute of the article.

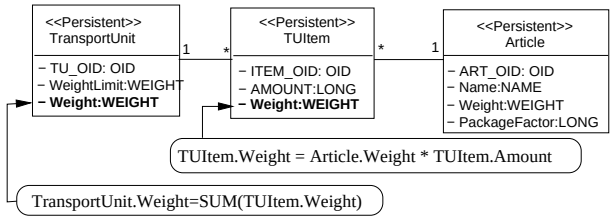

Figure 5. TME for weight attributes

2) Associations: Business functions may require additional associations between classes, that are not associated in the data source. Usually this is done for simplifying associations of related classes in the model. Again this TME type is demonstrated using the datamodel of the WMS. Consider the requirement to display information on all articles contained in a TU. This requirement can be fulfilled by extending the datamodel of the WMS with a 1:n association between the class *TransportUnit* and the class *Article*.

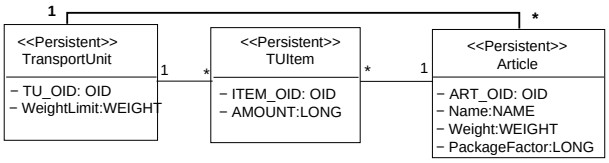

Figure 6. TME for associating articles with TUs

3) Classes: Extension of the datamodel with new classes requires at least an additional TME with a class attribute. It can also require a TME with an association, to connect the new class to an existing class in the model.

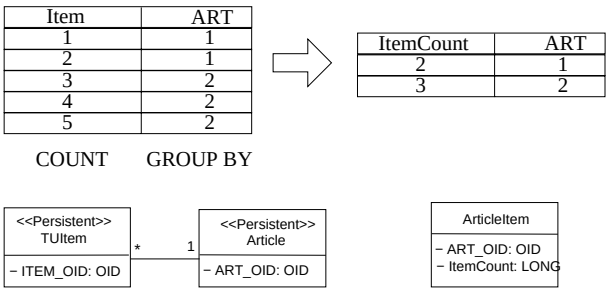

Figure 7. TME for a class

Fig. 7 shows a use case, which does not require a TME with an association. This example is driven by the requirement to count the number of TUItems for each article. Looking at the database tables shown in

the upper section of the figure, it is clear, that this requirement can be solved, by applying a query containing a GROUP BY clause and a COUNT function on the TUItem field. This requirement is solved by a TME with the class *ArticleItem*, which is used for constructing the entity objects containing the result from the database query. Using an additional class satisfies the following constraints which arise from the database query:

COUNT(x) → int

GROUP BY (x) → x

Every entity of the class *ArticleItem* is therefore fully defined within the metamodel, i.e. every attribute contains its correct metainformation. Note that this requirement could also have been solved by adding an additional attribute to the class *Article* instead of specifying a totally new class. The reason for using a class extension are performance considerations, which we will discuss in the next section.

B. Clientside and serverside TME

The initial idea of the TME mechanism has been the client side extension of the datamodel, e.g. to enable additional columns in table widgets being shown in the presentation layer. But in fact most data-intensive systems are build using a distributed architecture. In this context the meaning of software layers is equivalent to the meaning of software tiers. A tier is a layer, which is not deployed on the same machine as the other layers in the system. In a three tier system two configurations are usually deployed. The first configuration, also known as fat client, requires presentation tier and business logic tier being installed on the same machine. The second configuration, known as thin client, requires only the presentation layer being deployed on the client machine, while business logic tier and data source tier are deployed on the server.

Both configurations think of the data source tier being deployed on a server machine. This server contains the datasource, which is often a relational database, or it has a high performance network connection to the machine containing the datasource. In the second situation performance considerations can become important for applying TME to the datamodel.

The example demonstrated in Fig. 7 can also be implemented using a TME with an attribute for the *Article* class. But by applying the TME with an attribute the software engineer requires the infrastructure to transfer first all *TUItem* objects and all *Article* objects from the datasource layer to the layer containing the extended datamodel. Having finished the transfer the sum up of the corresponding TUItem entities for each article can be performed. This variant requires a big amount of data to be transferred, while the example in Fig.7 makes use of the advantages of the RDBMS, applying the corresponding query in a DML and then constructing new entities of the class ArticleItem from the returning set of this query.

C. Other design considerations

Beside the descision to use client or serverside TME another design consideration is the *data gathering mechanism* for the extended model elements. Especially in the case of attribute TME it is always required to specify a mechanism for retrieving the value of an attribute.

The first approach is the use of a *hook* function, which is called everytime the value of the extended element is requested. The other alternative implements the *observer* pattern [18], implying changes to the extended model element only when one of the observed model elements changes.

For demonstration purposes consider the example shown in Fig. 5. The example used two TMEs adding a weight attribute to the *TUItem* and *TransportUnit* class. Considering the requirement, that the amount stored in *TUItem* and the weight of the article are fixed, usage of two hook methods each implementing one of the equations shown in Fig. 5 is enough. If this requirement changes, additional observer methods need to be specified for the corresponding variable attributes. These observer methods need to trigger the two hook functions for updating the additional attributes.

Another important aspect of the TME approach is the temporary extension of a data model. Because the same data model of the data source layer is used as basis for the domain and presentation layers, extensions applied within these layers should not affect the (data) models of other layers.

D. Generic Components and TME

Generic components can be seen as software components for which specific properties have been left variable during components implementation [19]. For AMDD it means that component functionality is based on a data model as well as a metamodel and the model is used for configuration.

In case of generic components TME can be used to configure the runtime behavior of the component. Imagine a user interface component like a table, which dynamically displays data of a given class in the associated data model. Extending this class with additional attributes leads to displaying of additional table columns.

V. Implementational considerations

For demonstration purposes the uses cases presented in Sect. IV-A are implemented using the layered WMS introduced in Sect. II. The examples are realized using the framework of our real world application.

Provided that the structure of the persistent data has been modeled, an **Entity Container** (EC) [20] can be used as a model-based object oriented data cache. The architecture of the EC is shown in Fig. 8. The EC provides distinguishable objects called entities, identified by a unique value. It operates on two levels of the four-level metamodel hierarchy of OMG implementing the instance of relation between the two levels.

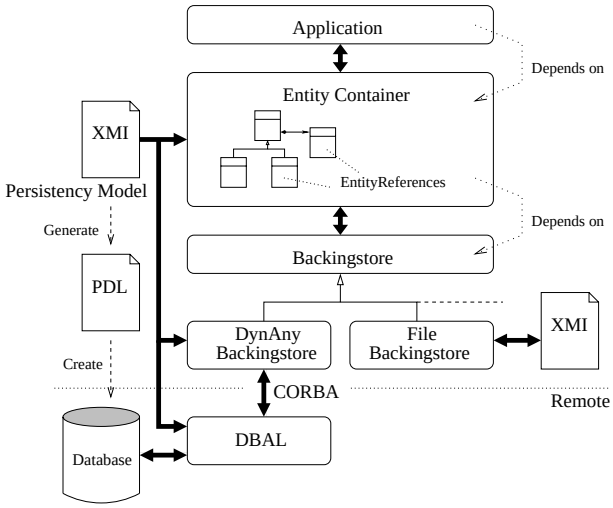

Figure 8. Entity container architecture

The UML data model is stored in a file using the XML Metadata Interchange (XMI) format. This file contains the UML model of the persistent data and is used by the EC and the associated backingstore, such as an object-relational bridge (DBAL). Data entities in the EC are accessed using a dynamic interface.

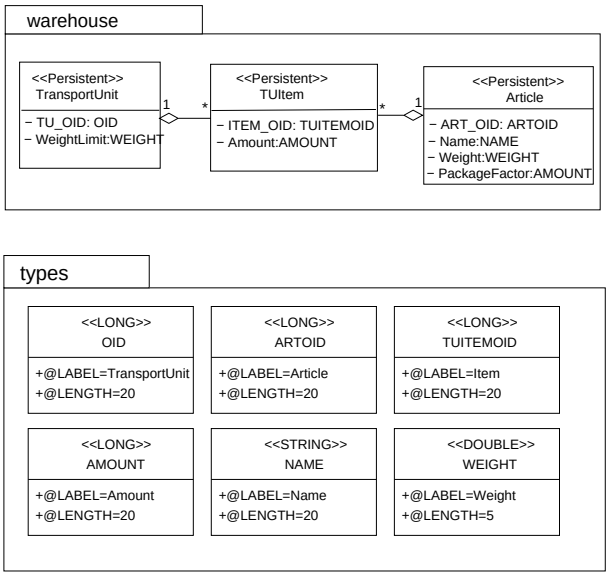

Figure 9. Case study data model

In our MDD approach the database is created from the persistency model, which is also used to configure the EC and its associated backingstore. Fig. 9 shows the data model used for our examples. It consists of two packages, with the warehouse package containing the persistent data classes. Attributes of these classes use the type classes defined in the types package. This data model is applied for creating a database containing the data shown in Table I. There exist two TUs, with one containing two TUItems and the other being empty. The two TUItems are instances of different articles. These data are used for demonstrating

TABLE I.
DATA USED FOR TABLE IN FIG. 10 AND FIG. 11

TU	TUItem	Amount	Article	Article.Weight
1234	211	2	815	3.0
1234	222	1	816	4.0
1235	-	-	-	-

the following use cases:

1) Applying TME for weight attributes (see Fig. 5)
2) Applying TME for a new class (see Fig. 7)

A. TME with the Entity Container

The EC framework is currently implemented for C++ and Java. Because both versions provide the same API, the following examples demonstrated for Java can also be applied in C++. While the C++ implementation is packaged in dynamic libraries, the Java implementation is deployed using the Eclipse plugin mechanism.

Eclipse is an open source Integrated Development Environment (IDE) written in Java, which has been initiated by IBM. The Eclipse IDE is based on the Open Service Gateway Infrastructure (OSGI) [21] framework and makes use of different design patterns [22]. It offers support for development of modularized applications consisting of several plugins. Each plugin contains classes and development fragments belonging together, and is used for realising a well-defined use case. A plugin manifest defines the plugin configuration data and contains extension point (EP) definitions, enabling the dynamical extension of plugin behaviour at runtime.

Listing 1. TME for weight attribute using sourcecode

```
String TU = "::warehouse::TransportUnit";
String TUItem = "::warehouse::TUItem";
String typeWeight="::types::Weight";

public EntityModel getModel() {
        EntityModel persistModel = datamodel.Activator.getDefault().getModel();
        try {
      IAttribute attr = persistModel.TME_addNewAttribute(TUItem,
                                                         "Weight",
                                                         typeWeight);
      attr.setHook(new TUItemWeightHook());
      attr = persistModel.TME_addNewAttribute(TU,"Weight",typeWeight);
      attr.setHook(new TUWeightHook());
        }
        catch (Exception ex) {
          System.out.println(ex.getMessage());
        }
    return persistModel;
}
```

TME use cases are typically realized as statements in the code of the programming language (see Listing 1 for an example realizing usecase (1) defined in the previous section).

Listing 2. EntityAttributeHook for TransportUnit weight attribute

```
public class TUWeightHook extends EntityAttributeHook {

  public IValue getAttribute(IEntityReference entity, String name) throws
                                                    EntityAttributeHookException
  {
    double weight = 0.0;
    try {
      for (IEntityReference ref :entity.getConnections("TUItem")) {
        weight += ((DoubleValue)ref.getAttribute("Weight")).getValue();
      }
    }
    catch (Exception ex) {
      System.out.println("Error calculating weight for '" + name +"';"+
                                                          ex.getMessage());
    }
    return new DoubleBuilder(weight).newValue();
  }
}
```

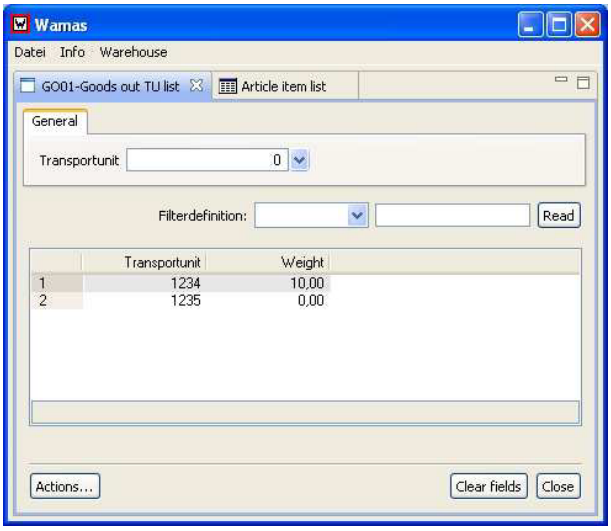

Figure 10. Dialog displaying extended *TransportUnit*

Listing 3. Table configuration for Fig. 10

```
// retrieve table from dialog layout
table_ = (TableView) swtLayout.getControl("ViewPanel.TableView");
//retrieve extended model
EntityModel model = userinterface.Activator.getDefault().getModel();
//construct Entitycontainer
IEntityContainer ec = ContainerFactory.createContainer();
// construct table configuration
TableViewConfiguration config = new TableViewConfiguration(ec,
                                                           model,
                          "::warehouse::TranportUnit#ENTRY");
// configure table
table_.setTableViewConfiguration(config);
```

Listing 2 presents a class, which is derived from the abstract class *EntityAttributeHook*. The hook method **getAttribute()** receives the extended entity of type *TransportUnit* and sums up the weight attributes of its connected *TUItem*s. Note that an object of this class is set as the hook object for the extended attribute weight in Listing 1. The second hook class *TUItemWeightHook* is also derived from *EntityAttribueHook* and is providing the same method getAttribute(), which calculates the weight of a *TUItem* by multiplying the weight of the associated *Article* with the amount attribute of *TUItem*.

A screenshot of a dialog displaying a table, which is configured with the code presented in Listing 3 is shown in Fig. 10. The dialog consists of a filter panel, containing a combo box with all *TransportUnit*.TU attribute values, and the table mentioned above, which displays the extended class *TransportUnit*.

The value of the weight column is calculated using the hook function defined in Listing 2 and a corresponding hook function for the weight attribute in TUItem. The data displayed in the table widget are based on the data defined in Table I.

Listing 4 illustrates the code for the data source layer. This layer holds the data model of the application. It also provides access to the data source.

According to Fig.8 each EC uses a data model and a backingstore, which holds the connection to the data source. The parameter for initialization of the EC in the upper layers of the application are retrieved from the data source layer.

Listing 4. Data source layer

```
..
        // Datastructure holding the data model
    private IModel dataModel_;

    // Backingstore connecting to the database
    private IBackingStore bs_;

        /**
        *  Public constructor initializing data layer
    **/
    public DataLayer() {
        // initialize the data model
        dataModel_ = Model.create4TME("resource/WMSModel.xml.zip");
        // open connection to database for backingstore
        IPersistency persist = new RemoteDbalPersistency();
                // initialize backingstore
        bs_ = new JavaDynAnyBackingStore(dataModel_,persist);
    }

    //Method for retrieving the model of the data source layer
    public IModel getDataModel() {
        return dataModel_;
    }

    // Method for getting access to the data source
    public IBackingStore getBS() {
        return bs_;
    }
..
```

Usecase (2) from the previous section is realized by a class *ArticleItemCounter* in the business logic layer. This class contains the method **countItems()**, which is presented in Listing 5.

Listing 5. Counting items of an article

```
public static ArticleItemCounterResult countItems() {

String article = "::warehouse::Article";

// retrieving model from data source layer
EntityModel model = DataSourceLayer.getModel();

// extending model with class and attributes
try {
  IPersistent articleItem = model.TME_addNewClass("::warehouse", "ArticleItem");
  IAttribute oid = model.findPersistent(article).lookupAttribute("Article");
  articleItem.addAttribute(oid);
  IPrimitive amount = model.findPrimitiveType("::types::AMOUNT");
  IAttribute itemCount = model.newAttribute("ItemCount", false, amount);
  articleItem.addAttribute(itemCount);
}
catch (WXException ex) {
  logger_.error("Error_applying_TME_"+ ex.getMessage());
}

// constructing EntityContainer with extended model
IEntityContainer ec = ContainerFactory.getInstance().createContainer(model);

// counting the items for each article
try {
  for (IEntityReference articleRef :ec.getAll(article)) {
    long itemCount = 0;
    for (IEntityReference itemRef:articleRef.getConnections("Item")) {
      itemCount += ((LongValue)itemRef.getAttribute("Amount")).getValue();
    }
    LongBuilder itemCountValue = new LongBuilder(itemCount);
    ITemplate templ = new Template();
    templ.setAttribute("Article", articleRef.getAttribute("Article"));
    templ.setAttribute("ItemCount", itemCountValue.newValue());
    ec.createByTemplate("::warehouse::ArticleItem", templ);
        }
        return new ArticleItemCounterResult(ec,model);
  }
  catch (Exception ex) {
    logger_.error("Error_calculating_itemcount:_"+ ex.getMessage());
  }
  return null;
}
```

This method retrieves the data model from the data source layer and applies a TME with the class *ArticleItem*, containing the attributes Article and ItemCount. After applying TME, an EC is constructed for the extended model and is loaded with entitities of the class *Article* and associated *TUItem* entitites contained in the database. For every *Article* in the EC the value of the amount attribute of its associated *TUItem* entitities is summed up, and a new entity of type *ArticleItem* is constructed for the current Article attribute and the received sum of items.

Having counted all items, the EC is returned to the calling class. Listing 6 contains the configuration for the

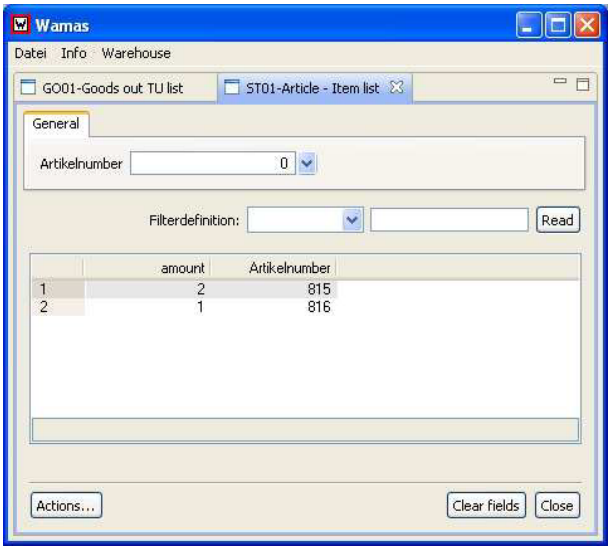

Figure 11. Dialog displaying the number of items for each article

table widget presented in Fig. 11. The configuration is based on the result of the method defined in Listing 5.

Listing 6. Table configuration for Fig. 11

```
// retrieve table from dialog layout
table_ = (TableView) swtLayout.getControl("ViewPanel.TableView");

//call business logic
ArticleItemCounterResult result = ArticleItemCounter.countItems();
TableViewConfiguration config = new TableViewConfiguration(
        result.getEC(), result.getModel(),
"::warehouse::ArticleItem#ENTRY");

// provide configuration
table_.setTableViewConfiguration(config);
```

B. Generic programming using Eclipse plugins

In case of programming with Java and using the Eclipse plugin mechanism, the declaration of TME use cases can be done by contributing to a specific EP, which is defined in the plugin containing the data model.

Listing 7. EP definition for TME with attribute

```
<?xml version='1.0' encoding='UTF-8'?>
<schema targetNamespace="DataModel">
  <annotation>
    <appInfo>
      <meta.schema plugin="DataModel" id="TMEAttribute" name="TME"/>
    </appInfo>
    <documentation>Transient model extension with attribute</documentation>
  </annotation>

  <element name="extension">
    <complexType>
      <sequence minOccurs="1" maxOccurs="unbounded">
        <element ref="TME_Attribute" minOccurs="1" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="point" type="string" use="required">[..]</attribute>
      <attribute name="id" type="string">[..]</attribute>
      <attribute name="name" type="string">[..]</attribute>
    </complexType>
  </element>

  <element name="TME_Attribute">
    <complexType>
      <attribute name="class" type="string" use="required">[..]</attribute>
      <attribute name="attribute" type="string" use="required">[..]</attribute>
      <attribute name="type" type="string" use="required">[..]</attribute>
      <attribute name="AttributeHook" type="string">
        <annotation>
          [..]
          <appInfo>
            <meta.attribute kind="java"
                    basedOn="wx.datamodel.EntityAttributeHook"/>
          </appInfo>
        </annotation>
      </attribute>
    </complexType>
  </element>
</schema>
```

An exemplary definition is shown in Listing 7. This definition is contained in the data model plugin and is used by the extension defined in Listing 8, which is located in the corresponding layer plugin applying the TME mechanism.

Listing 8. TME for weight attribute using EP from Listing 7

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension
         point="DataModel.TMEAttribute">
      <TME_Attribute
            AttributeHook="TUWeightHook"
            attribute="Weight"
            class="TransportUnit"
            type="::types::Weight"/>
   </extension>

</plugin>
```

The extensions of the data model plugin can be processed each time, the data model is requested. Before returning a new instance of the data model the plugin processes the provided EP contributions, which leads to the data model with TME applied.

Note that both TME attribute use cases, the sourcecode mechanism(Listing 1) as well as the EP mechanism (Listing 8), make use of the same hook class defined in Listing 2.

VI. Conclusion

This paper introduced a new method to ease the model driven construction of layered data-intensive software. Applying the concepts of data modeling using traditional approaches results in one data model for each software layer. This leads to redundant class definition in different models with respect to the data source. Changing the data model of the data source becomes difficult, because all corresponding classes in the other data models need to be changed as well. Furthermore each model is driven by various requirements, which leads to a different number of attributes in the equivalent class depending on data model.

To overcome this challenge, data models of different software layers can be received applying additional extensions on the data model from the underlying software layer. The mechanism for applying these model changes on the fly is called Transient Model Extension (TME). This paper presented TME use cases including the extension of classes, class attributes and associations. These transient extensions are driven by layer specific requirements and are applied with different scopes (e.g. layer specific TME, method specific TME).

To demonstrate the advantages of this method, we presented an example using the framework of our real world application, considering various requirements of the business logic layer and the presentation layer. We also provided a comparision of our approach to related technologies dealing with persistent data structures.

We have found this approach to be useful in constructing large data-intensive systems in the business domain of logistic, because it supports the modularization of different methods in specified data retrieving classes. Because changes to the model are transparent to the

persistent data structures only small changes are needed in the configuration of data displaying widgets. The TME method is also an important part of our approach for implementing a model based software product line for data-intensive systems [23].

ACKNOWLEDGMENT

We would like to thank our company, Salomon Automation GmbH, for their grant to support our research. We also thank Zsolt Kovacs for the interesting discussions, and Carlo Jenetten and Robert Lechner for their support in coding the mentioned tools.

REFERENCES

[1] S. J. Mellor, A. N. Clark, and T. Futagami, "Model–driven development — guest editor's introduction," *Software, IEEE*, vol. 20, no. 5, pp. 14–18, 2003.
[2] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, 2003.
[3] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, p. 25, 2006.
[4] OMG, "UML Infrastructure, Version 2.0 ," Object Management Group, Tech. Rep. 2002-09-01, 2002.
[5] D. S. Frankel, *Model Driven Architecture, Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.
[6] I. Sommerville, *Software engineering (6th ed.)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2001.
[7] S. W. Ambler, *Agile Database Techniques*. Wiley Publishing, Inc., 2003.
[8] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison Wesly, 2003.
[9] M. Thonhauser, G. Schmoelzer, and C. Kreiner, "Model-based Data Processing with Transient Model Extensions," in *ECBS*, 2007, pp. 299–306.
[10] A. Demir, "Comparison of model-driven architecture and software factories in the context of model-driven development," in *MBD-MOMPES '06: Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD-MOMPES'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 75–83.
[11] OMG, "UML Superstructure, Version 2.0," Object Management Group, Tech. Rep. 2002-09-02, 2002.
[12] D. Sceppa, *Microsoft ADO.NET 2.0 Core Reference*. Microsoft Press, 2006.
[13] *"Hibernate Reference Documentation"*, "3.2.0ga" ed., "Hibernate", "http://www.hibernate.org/5.html", 2006.
[14] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose, *"Eclipse Modeling Framework"*. Addison-Wesley, "August" 2003.
[15] J. B. Warmer and A. G. Kleppe, "Building a flexible software factory using partial domain specific models," in *Sixth OOPSLA Workshop on Domain-Specific Modeling (DSM'06), Portland, Oregon, USA*. Jyvaskyla: University of Jyvaskyla, October 2006, pp. 15–22.
[16] M. Barbero, F. Jouault, J. Gray, and J. Bézivin, "A practical approach to model extension." in *ECMDA-FA*, ser. Lecture Notes in Computer Science, D. H. Akehurst, R. Vogel, and R. F. Paige, Eds., vol. 4530. Springer, 2007, pp. 32–42.
[17] S. Mitterdorfer, E. Teiniker, C. Kreiner, Z. Kovács, and R. Weiss, "XMI based Model Linking," in *CAINE 2002, International Conference on Computer Applications in Industry and Engineering, Las Vegas, Nevada USA, Nov. 11-13*, E. Nygard, Ed. Cary, NC: ISCA, 2003, pp. 322–325.
[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Reading, MA: Addison Wesley, 1995.
[19] L. Baum and M. Becker, "Generic Components to Foster Reuse," in *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems*, 2000, pp. 266 – 277.
[20] G. Schmoelzer, S. Mitterdorfer, C. Kreiner, J. Faschingbauer, Z. Kovács, E. Teiniker, and R. Weiss, "The Entity Container — an Object–Oriented and Model–Driven Persistency Cache," in *HICSS'05, Big Island, Hawai'i, USA, Jan. 3-6*. IEEE, 2005, p. 277b.
[21] T. O. Alliance, *OSGi Service Platform Core Specification*, version 4.1, release 4 ed., April 2007.
[22] E. Gamma and K. Beck, *Contributing to Eclipse. Principles, Patterns, and Plugins.: Principles, Patterns and Plugins*. Addison-Wesley Professional, 2003.
[23] G. Schmoelzer, C. Kreiner, and M. Thonhauser, "Platform design for software product lines of data-intensive systems," in *Proceedings of the 33th EUROMICRO Conference*, August 2007.

Michael Thonhauser is currently a Ph.D. student at Graz, University of Technology, Austria and is also working at the research and development department of the company Salomon Automation GmbH. He received his MS degree in informatics and eletrical engineering from Graz, University of Technology in 2005. His research interests include software engineering methods and distributed systems.

Gernot Schmölzer is currently working at the research department for Salomon Automation GmbH, a large logistics software vendor in Europe. He received his Ph.D. in informatics and electrical engineering from Graz University of Technology in 2007. His research interests include model-driven development, software product line engineering and software engineering practices.

Christian Kreiner graduated and received Ph.D. degree in Electrical Engineering from Graz University of Technology in 1991 and 1999 respectively. His research interests include software technology, software engineering and quality management. Christian Kreiner is currently head of the R&D department at Salomon Automation, Austria, focusing on software development for AS/RS (automatic storage/retrieval systems) and researcher at the Institute for Technical Informatics of Graz University of Technology.