Automatic code generation by model transformation from sequence diagram of system's internal behavior

EL BEGGAR Omar, BOUSETTA Brahim*,GADI Taoufiq
LAVETE Laboratory
FSTS, Hassan 1st University
Settat, Morocco

Abstract—**Software engineering process aims to help leading an IT project from requirements specification to code implementation in a specific platform. Such process can be improved by the model driven architecture (MDA) proposed by the OMG. Indeed, during the whole development process a set of models are used to represent different views of system and which can be enriched by additional information and transformed from more abstract into more concrete ones by applying a set of model-to-model transformation. Moreover, the MDA allows also the code generation from platform specific models (PSM) by the means of generators that automatically transform models into source code for the chosen platform.**

Within this context, this presents code generator that allows generating source code from the sequence diagram of system's internal behavior platform independent model by the mean of a set of model transformations. An intermediate structural model representing the Java PSM is generated instead of the source code directly to allow the extension of target platform by enabling the transformation of this model after. To illustrate this feature, we have extended the JAVA platform with the enterprise java beans capabilities by applying an Ecore Meta Facility profile for EJB to produce persistent entities and session beans that provide an interface for different data access operations. Also, the persistent entities can be used to create the different tables of the schema corresponding to those entities, thereby another contribution of this work. Also since the sequence diagram implements the MVC design pattern, we will generate controllers with the detail implementation of their methods that allows to coordinate system's objects to execute the use case.

The main objective of the approach is to concentrate the entire efforts on the system's abstraction and business logic by building different views of systems through PIMs or PSMs and feed the generator with this knowledge to be able to transform those abstract models automatically to implementation code.

Keywords-component; Model Transformation; Code generation; software process; EMF profile; sequence diagram.

I. INTRODUCTION

Throughout the software development a set of models are used to represent different views of the system. Those models are generally enriched by additional information and transformed from more abstract into more concrete ones by applying a set of model-to-model transformation, one of the features of the model driven architecture (MDA) [1] proposed by the Object Management Group (OMG) [2]. Moreover, the MDA allows also the code generation from platform specific models by the mean of generators which automatically transforms models into source code for the chosen platform.

Within this context, this paper aims to provide a code generator that allows generating complete source code for the Java platform from the sequence diagram of system's internal behavior (SDSIB) which is a platform independent model (PIM) by the mean of set of model transformations. An intermediate structural model representing the Java platform specific model (PSM) is generated before getting the source code. The goal of generating such structural representation (model JAVA) of the target code instead of the source code directly is to improve readability as well as efficiency of generated code and to enable the transformation of the code's model after generation allowing thus the extension of the target language with more features. Some authors state that the final application "should not be the main goal of code generation" [3]. For example of these features, we have extended the JAVA platform with the enterprise java beans (EJB3) capabilities by applying an Ecore Meta Facility (EMF) profile [4] for EJB3. The main objective of the approach is to concentrate the entire efforts on the system's abstraction and business logic by building different views of systems through PIMs or PSMs and feed the generator with this knowledge to be able to transform those abstract models automatically to implementation code.

The core idea of this approach is the code generation for the JAVA platform starting from the PIM SDSIB through the PSM of Java model. Firstly, the different model classes corresponding to the objects that are involved in the execution of the uses case and presented in the SDSIB are transformed to a *JavaClasses* in the generated structural model of the Java platform. Supplementary details of these *JavaClasses* such as the different attributes and mapped association are gathered from the Domain Class diagram (CDC). Secondly, different operations from SDSIB that are sent during the interaction between the system's objects and that are expressed in formal syntax with a precise semantic defined by the Extended Post-condition Matrix (EPM) toolset will be transformed to the corresponding java methods based on their semantic according to this toolset. These methods are generated with complete

* Corresponding author. Tel : +212668640231/+212660623211
E-mail address: ibbousetta@gmail.com
Permanent address: N° 21, Rue el okhouane hay raha 20200 Casablanca Morocco

detail such as the full signature and body source code. Thirdly, since the SDSIB implements the MVC design pattern to produce software that is easy to extend and to maintain, we will generate the different controller's methods with their full body's code that will coordinate the system's business objects to execute the use case. Finally, we will extends the JAVA platform with the EJB capabilities by applying the EJB profile [5,6] to produces a persistent Entities session beans that provide an interface to use the different Create/Retrieve/Update/Delete (CRUD) operations. Thus javaClasses will be annotated with different annotations to map tables from the database including the relationships such as OneToOne, OneToMany… Moreover, the generated configuration file for the persistence can be modified to use the JPA API to generate the different tables corresponding to the Entity classes if they are not yet created, thereby another contribution of this work. The Figure 1 below illustrates the approach.

This paper subscribes with our previous works [7,8,9] in a global approach that attempt to automate a software engineering process starting from Computational Independent Model (CIM) - business requirements – to code implementation in a specific platform. However, in [9] we have focused only on automating the generation of the most important increment of the design phase: Sequence Diagram of system's internal behavior (SDSIB) which shows the system's objects interaction needed to accomplish the expected functionality which is obtained automatically by a model transformation using as a source models: (1) the sequence diagram of system's external behavior (SDSEB) that represents actor's actions and their corresponding system's responses; each action or message initiates interactions between system's objects that can be identified over (2) the domain class diagram (DCD) which is an UML class diagram that contains only classes and their attributes without methods; (3) the extended operations contract (EOC) that use the extended post-conditions matrix (EPM) the new toolset proposed to extend the original LARMAN operation contract (LOC) [9] by integrating new elements and proposing an improved formal syntax to determine correctly the operations and their concerned objects source and target present in the expected SDSIB model of the transformation. This paper uses this generated SDSIB as the principle source model to generate the code. Therefore, the messages represented in the SDSIB later will have a precise semantic according to this EPM that will help us to generate full method's body code.

A running example concerns buying items uses case in e-commerce web site is given throughout this article to illustrate our approach. The used SDSIB is the one that were generated in our previous work by applying the EPM toolset. At the end of this work a complete java files are generated for this running example.

The remainder of this paper is structured as follows: In the next section we present the most relevant related work to the topic of the proposal and a motivation of this work is also given, Section III gives an overview of the proposed approach to automate the software engineering process. In this section we introduce also the operation contract of LARMAN and its extended version that was proposed in our previous work as well as the EPM toolset. Section IV concerns SDSIB that represents the source PIM model for this model transformation. An overview of this important design model, it's metamodel and the generated one using the EPM toolset for the running example will be given in this section. The following section is dedicated to present the approach to generate the source code for the java platform by applying model transformation. In this section, we first present the java metamodel, then the different rules that are used to perform the transformation for instantiating this java metamodel. Then, after giving a brief introduction to UML and EMF profiles, we will present the EJB profile that is applied to extend the generated structural mode of the java platform to enrich it with necessary annotation for JPA persistence and generate session beans allowing different CRUD operations; the detail code implementation for these methods is given. In section VI we evaluate the generated code with the proposed approach. Finally we end with a conclusion to discuss what has been done and give some prospects.

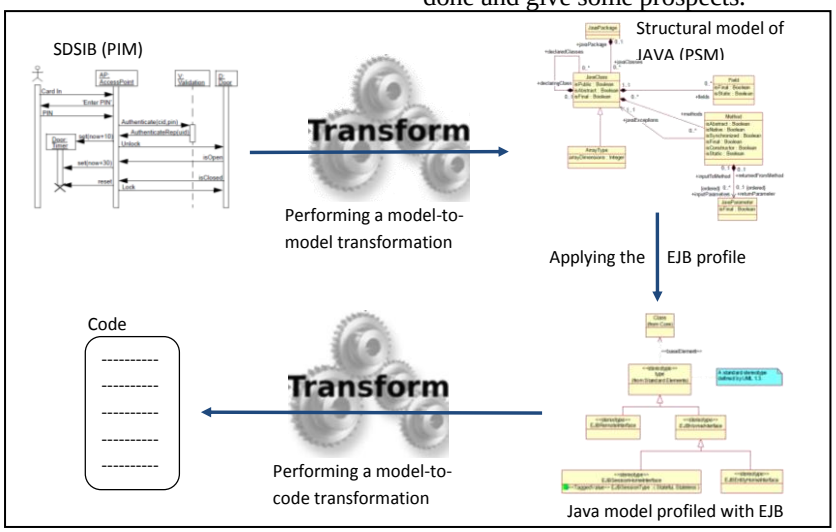

Figure 1: Different transformations performed to generate the code starting from the SDSIB.

II. RLATED WORK

Various works in the field of code generation domain have been conducted during the last few years, and these works are interesting in the context of the present paper. Although all the works in this field are important for industrial and scientific communities, some projects and research groups/institutions of high relevance will be highlighted.

The first Kind of code generator is based on code generation from Petri-Nets which has a long tradition. However, unlike methods for the analysis and simulation of Petri-Nets, code generation is not yet considered a standard feature. An extensive review of existing work in the area of automatic code generation from Petri-Nets is given in [3]. Most of the approaches in this review focus on code generation from (extended) low-level Petri-Nets, e.g. for the generation of controllers [10, 11]. Even though the review also lists approaches for code generation from high-level Petri- Nets, the work in this area is not based on object-oriented principles, and in consequence not applicable to more complex systems. A frequent use of approaches to automatic code generation from Petri-Nets is the validation of requirements in systems engineering.

In contrast to this view, [12] shed some light on Petri-Net based code generation based on MDA by considering models as a central means not only for the capturing and validation of requirements, but for the whole development process in systems engineering. It introduces an approach based on a class of object-oriented Petri-Nets in order to be applicable also for complex systems and it intend to support the use of models throughout the entire systems development process and not only for validation purposes. It also gives an evaluation of different strategies for automatic code generation from Petri-Nets with the focus on their general applicability as well as the readability, extensibility and efficiency of generated code.

The second kind of code generator is generation of code by model transformation. These approaches treat code as a model, while most MDE approaches generate code through the use of textual template engines, which produce plain text, not amenable to further transformation. By treating generated code as a model, it is possible to extend the target language and add convenient language features such as partial classes and methods, and interface extraction. Some other approaches have generated partial artifacts through the use of partial classes, which are then combined by the regular compiler for the target language. Warmer and Kleppe [13] describe experiences with such an approach. These approaches rely on the target language to support this features. Generation of partial artifacts has also been applied by Huang and Smaragdakis [14] by use of Meta-AspectJ [15]. There have been other approaches that aspect weaving at the model level rather than using this feature in the generated code [16, 17].

In [18] a code generator by model transformation was presented. Also, it have been demonstrated how generator concerns can be better separated and showed how transformation rules can be made more concise and modularized by extending the target language. Several ways of combining type analysis with rewriting have been discussed and introduce the approach of three-phased type analysis and transformation, in which name resolution, constraint checking, and rewriting can all be specified as strictly separate concerns. When additional language abstractions are introduced, they can take advantage of the open extension points provided by the generator. These extension points, allow the extension to easily plug into the type analysis, model transformation and code generation subsystems. A number of language extensions into the generator have been built, most notably the access control and workflow extensions, which are entirely built by plugging into the extension points mentioned. In this approach, the feature of partial classes and methods has been overlaid directly on the output language. This overlay definition can be used across different applications, i.e. other code generators that produce Java code. In contrast, using the more typical approach of strictly separating model transformation and code generation (using templates), as applied in [16, 17], a very low-level, general-purpose model representation would have to be used to achieve the same result.

Stratego/XT [19] provides another development environment for creating standalone transformation systems. It combines Stratego [20], a language for implementing transformations based on the paradigm of programmable rewriting strategies, with XT [21], a collection of reusable components and tools for the development of transformation systems. In general, Stratego/XT is intended for the analysis, manipulation and generation of programs, though its features make it useful for transforming any structured documents. In practice, Stratego/XT has been used to build many types of transformation systems including compilers, interpreters, static analyzers, domain specific optimizers, code generators, source code refactorers, documentation generators, and document transformers.

In [23] an approach for simplifying the specification of conceptual schemas (CSs) was presented. It provides a way for modeling the operations that define the system behavior by providing a method that automatically generates a set of basic operations that complement the static aspects of the CS and suffice to perform all typical life-cycle create/update/delete changes on the population of the elements of the CS. The proposed method guarantees that the generated operations are executable, i.e. their executions produce a consistent state with the most typical structural constraints that can be defined in CSs (e.g. multiplicity constraints). In particular, this method takes as input a CS expressed as a UML class diagram (optionally defined using a profile to enrich the specification of associations) and generates an extended version of the CS that includes all necessary operations to start operating the system. If desired, these basic operations can be later used as building blocks for creating more complex ones. While, [23] provide a method for generating CRUD operations to manipulate database data, [22] provides a tool for behavioral modeling. It defined textual and visual notations for UML actions and built supporting editors. Furthermore, it defined also a mapping from UML actions to Java and model compilers were built, which support the generation of complete and compile-ready applications including their behavioral parts.

Regarding to web applications, many modeling languages have been developed, including WebML [24], MIDAS [25], OOWS [26], Netsilon [27], and UWE [28]. UWE generates JSP code via a model representation conforming to a JSP metamodel. Netsilon uses an intermediate language for code generation in order to increase retargetability of the generator. The other approaches use textual, usually template-based code generation. WebML interprets its models rather than generating code from them. Most approaches apply model transformations with the purpose of retargetability, or with the purpose of expressing "as many artifacts as possible using models as this allows for processing these artifacts using model transformations" [29]. Only Netsilon actually models the target source code.

This paper subscribes in the second categories of code generators which bases on code generation by model transformation. We have also generated a structural model as in [18] for the target platform to enable its extensibility. In spite of this variety of approaches for code generation, so far there is no complete tool to carry out an IT project according to a software engineering process from requirements specification up to code implementation. Approaches based on Petri nets are still far to be a standard for code generation and still not yet used in the different phase of modeling, usually used for validation purposes. The motivation of this work is to complete our previous work to completely automate the hybrid software engineering process between UP and XP. In [9] we present the entire approach with the different proposed metamodels and the different transformation to connect them. However, in that work only one analysis increment was generated, the SDSIB which represents the most important increment of the design phase in the proposed process. In [8], we proposed the generation of a second increment, the design class diagram from the state transition diagram. Thus another analysis- PIM to Design-PIM- transformation has been done. This work aims to realize the final stage of the process which the generation of code from the SDSIB which was generated in [9] through a model transformation.

III. Overview of Automating Software Process Development

In our previous work [9], we have presented a software engineering process with different proposed metamodels at different phases of software development. We have also performed a model transformation from a platform independent model (the system's sequence diagram of external behavior) to another detailed platform independent model (the sequence diagram of system's internal behavior) by means of a toolset which provides a rule set based on the post conditions of the LARMAN's operations contract: the extended post-condition matrix toolset (EPM).

A. LARMAN Operations Contracts

A LARMAN Operation Contract (LOC) identifies system state changes for each received incoming message and show how the system objects will interact with each other in order to respond to this message. Indeed, the main objective of this contract is to highlight those interactions and describe consequently the new system state. This contract describes detailed system's behavior in terms of state changes to objects in domain model, after a system operation has been executed [30]. It describes the system state changes, by determining the pre-conditions and post-conditions: While the pre-conditions consist in determining the initial state of the system or otherwise the objects created underway before executing the operation, the post-conditions describe the system objects state after operation completion.

Craig LARMAN has predefined post-conditions as following:

- Objects created or destroyed.
- Associations formed or broken.
- Attributes modified.

These post conditions determine what will happen after the execution of the operation (creation, destruction, associations of objects, attribute changes) without providing the object responsible to do that. However, to draw a complete objects interaction in SDSIB, the source object and the responsible one are required. Thus, using only LOC to trace the SDSIB will be not sufficient. Also the LOC does not indicate the post-conditions regarding display or calculate, print and check messages and their related operations. Therefore, it is insufficient to draw complete interactions between objects and deduce specific operations issued from some incoming messages in the SDSEB like display, check or print.

for completing this operation contracts and allowing thus drawing complete interaction in SDSIB we have extended the LOC with new post conditions by providing the new toolset Extended Post-condition Matrix (EPM).

B. Extended Operations Contracts

Generally, when using the LOC to describe system's state after the execution of an operation, designers have to apply manually General responsibilities Assignment Software Patterns (GRASP) to assign responsibilities to the object in charge to achieve the post-conditions. To remedy these shortcomings, we proposed a solution that extends the post-conditions by presenting a new post-condition syntax which includes GRASP and generating finally not only the operations, but also deduces their objects source and target as well as the post-conditions regarding display, print and check messages were added by proposing a new solution the Extended Post-conditions Matrix (EPM) allowing to generate automatically the operations with their source and target objects.

In the new EOC, the extended post-conditions are used to determine automatically in common sense the interactions between objects based on new formal syntaxes. Thus, EOC allows designers to avoid design mistakes, by providing exact source and target objects participant in the interaction and thus improves the quality of modeling. Regarding display, print and check messages, new extended post-conditions are created.

C. The toolset: Extended Post-condition Matrix (EPM)

In order to generate automatically resulting interactions, we have proposed the EPM matrix that extends the original post-conditions by adding new details and defining new post-condition syntax. For example, the creation post-condition as it is defined by LARMAN: "Subject class was created" does not determine the responsible object which will be charged to send the operation "create" to the subject class in order to instantiate it. However, the alternative syntax in EPM: "Subject class was created by Responsible class" specify the element responsible for the creation and integrate the GRASP known as (Creator) which allow for deducing that the operation "create" has the responsible object as source and the subject class as target. Otherwise, the new syntaxes used in EPM incorporate the GRASP concepts that represent the guidelines for assigning responsibility to classes and objects in object-oriented software system to determine correctly the interactions.

To use the EPM toolset in order to determine completely all system interactions, the designer is asked to respect the formal syntax of the post-conditions. Thus, they can generate the operations signatures and the interaction responsible objects. Each post-condition has its own formal syntax and its inputs could be optional or required. The post-condition has as possible inputs: subject class, responsible class, associate class, multiplicity, attribute and parameter.

The Table 1 represents the proposed EPM which takes all the post-conditions already defined by LARMAN and offers a new formal syntax for each one by integrating GRASP and matching them to operations signatures. The EPM propose new extended post-conditions "Display" "Check" and "Print" which are not expressed before by LARMAN as post-conditions.

Concerning the new post-conditions added to the EPM, first the post-condition display may simply be an on-screen display or represents a calculation. For this post-condition, we must indicate the subject class attribute that will be displayed and the calculating attributes to accomplish the post-condition if it's a calculation. The Boolean attribute "Onscreen" in the meta-model allow distinguish between those kinds of messages.

Finally, the post-condition check allows verifications of the subject class attributes called "Checked" attributes with the post-condition parameters entered through the GUI. However, the post-condition print allows for identifying print operations concerning the printable classes like tickets, commercials orders and it has also parameters like order date, order number etc.

With regard to the Associations and disassociations post-conditions, in addition to objects responsible and subject, we have to indicate the associate object. Sometimes you have to retrieve the associate object in order to perform association or break it. In this case, we must enter the search key which is simply the parameter of the post-condition as well as the responsible object where we can find the associate object.

Table 1: The EPM toolset

Name	Description and generating operations
Creation	Creation of a subject class instance which defines two operations "create" results from controller to the responsible class of creation and Constructor which has the same name as its subject class invoked by the responsible class to the subject class
Modification	Modify the subject class attribute by taking as value the post-condition parameter. The type of parameter will be deducted from the corresponding domain attribute.
Destruction	Destruction of a subject class instance from which results the destruct and Destructor in subject class which has the same name as its subject class but it is prefixed by "$" .
Association Formed with parameter	Association formed between the subject class and the associate class that must be found first by a parameter with the operation "find" and an operation "set" will result from the controller to the subject class to perform the association
Association Formed without parameter	Association formed between the subject and the associate class without an incoming parameter which is expressed by a set or add operation depending on the max multiplicity of the association end between the associate object and the subject class.
Disassociation with parameter	Association braked between the subject and the associate class. two operation are produced : "find" to retrieve the associate object by the post-condition parameter; and "remove" break association
Disassociation without parameter	Association braked between the subject and the disassociate class without an incoming post-condition parameter
Display	Display event allows displaying subject class attributes by calculating or simply displaying them on screen.
Check	Post-condition Check allows checks or verifications between its parameters and attributes of the domain classes
Print	Post-condition print allows identifying print operations concerning subjects classes which are printable like tickets, commercials orders

IV. SDSIB OVERVIEW:

While The sequence diagram of system's external behavior (SDSEB) is a UML sequence diagram that shows only interactions between actors and the whole system as unique entity which is represented by one lifeline without focusing on system objects interactions, the SDSIB shows ordered interactions between objects with their lifeline and the exchange of messages between them. In addition to these elements these sequence diagrams generally represent the object activated by a rectangular lifeline. When an object is not active, just existing, it has a dashed lifeline. Along the time axis, timing notes or marks can be added. These timing marks can be used to give constraints, like specifying the maximum time a message exchange may take.

To obtain software that is easy to change and to maintain it is recommended not to allow the actors to interact directly with the business objects to avoid creating a strong coupling. This can be resolved using the MVC design pattern.

It is to remember that the SDSIB used in this paper and which is considered as the principle source model of the transformation to generate the code is traced basin on the EPM formal syntax, therefore each message has a precise semantic the will lead us to generate the source code.

A. Meta-model of SDSIB

Figure 2 below shows the used source meta-model of SDSIB that presents the different messages sent by the actors implied in the uses case. For each incoming message a set of operations that represents the exchanged messages or interactions between objects will result. The operation may have parameters and return a value and it is concerned by two objects: source object that represents the object that will invoke the operation and target object that will contain and execute the operation. As it was mentioned above, we specify three types of objects to implement the MVC design pattern: View, Controller and Model. Each operation belongs to an interaction operand that can be simple or a combined fragment (loop, ALT…).

We can see that the used meta-model is slightly different from the standard UML sequence diagram [32, 33, 34] without opposing with it or with the Meta Object Facility (MOF) [35]. Indeed, the SDSIB is a particular sequence diagram that contains whole interactions to respond to the message incoming from actors to the system. On the other side, the SDSIB meta-model presents several ordered operations. An operation has parameters and a return value. Each operation has also two extremities: source class and target class which can be any kind of classes (view, controller or model). Each operation concerns an incoming message trigged by an actor which can be principal or secondary.

B. SDSIB of the running example

Figure 3 shows resulting model of the SDSIB according to the running example "buy item" after the execution of the model transformation that was based on the EPM toolset. We can see that the whole operations regarding the post-condition declared in EOC are generated with all details necessary to draw the SDSIB. For example, the operation "findItem (code: int): Item" according the post-condition association number 3 in EOC, has as source class: the object controller called: "Handler_Buyitem", as target class: Catalog. The order of the operation is 3 related to its post-condition and it has code as parameter. Finally, the operation returns an object Item.

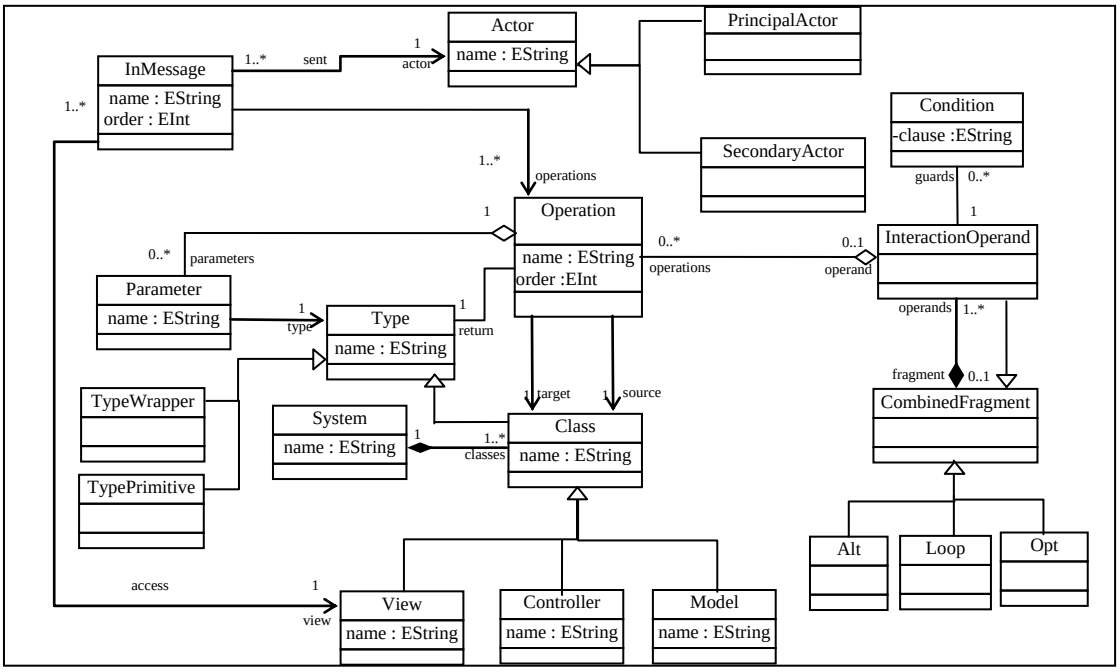

Figure 2: Meta-model of the sequence diagram of system's internal behavior

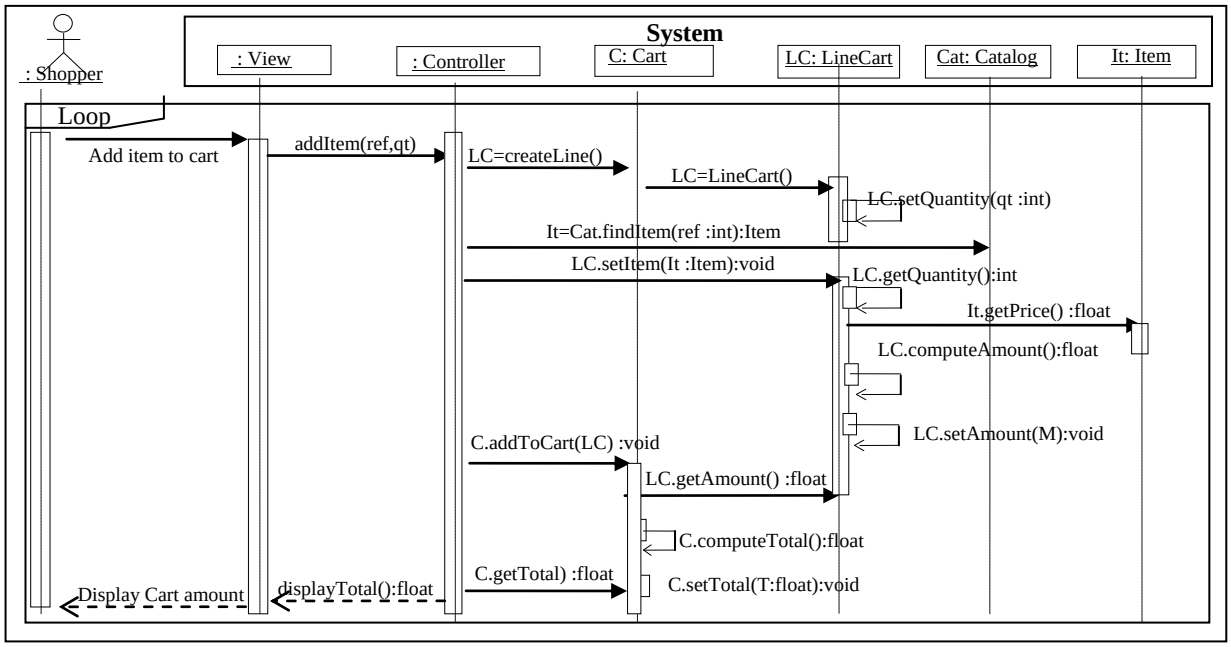

Figure 3-a: SDSIB of the running example.

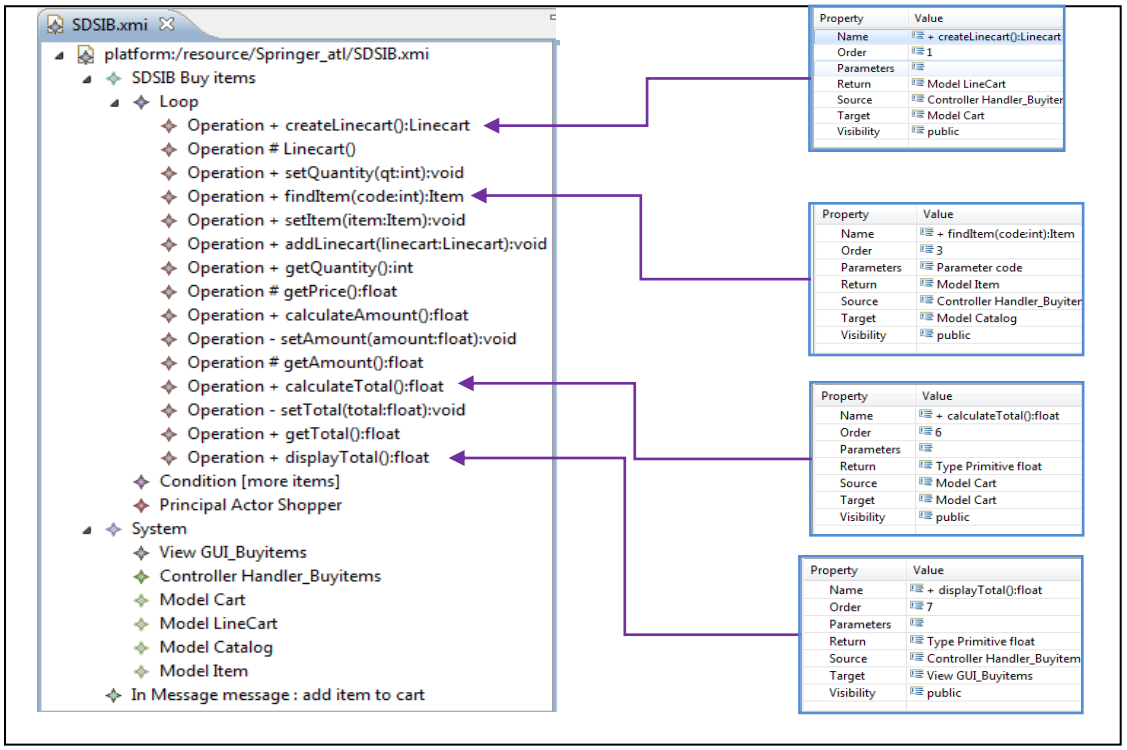

Figure 3-b: SDSIB's ecore diagram of the running example.

V. CODE GENERATION BY MODEL TRANSFORMATION

Since UML has become a standard for object oriented system modeling and used during the whole system development process to represent its different views of abstractions, it is interesting to investigate how we can generate code from these models. However, UML remains largely undefined from a semantic point of view because it is not made for a domain specific language and thus the code generation is too difficult. To overcome these problems, many works related to this topic have been done to customize UML with new formal basis to adjust it to a domain specific language (DSL) thus allowing the code generation.

In this paper we present a model transformation approach to generate code from the sequence diagram of system's internal behavior (SDSIB) which is one of the most important platform independent models of the design phase in software development process. It shows how the systems objects collaborate and interact to execute the uses case. However, code generation from the sequence diagram as it is specified by the OMG seems impossible. For that, we will use the SDSIB generated using the EPM toolset in which operations are defined according to a precise formal syntax. Indeed, this toolset defines list of interactions between systems objects as following: request for object creation or destruction, object association or objects association break, check operation, display operation and calculate operation. Understanding the semantic of each one of these operations allows an easy code generation for the suited platform. Moreover, since this SDSIB implements the MVC design pattern to build a system that is easy to maintain and to evolve, we will generate the code for the controllers methods which contains the main code to execute the use case which is generally a set of systems object's method calls, thereby another contribution of this work.

The platform used in this paper is JAVA. However we will generate a structural model for the suited platform instead generating directly the code, i.e. we will first generate a platform specific model (PSM) form the PIM SDSIB by the means of a model-to-model transformation before generating the code from this PSM by a model to code transformation. Such intermediate models allow for extending the platform with additional features before generating the plan text. To illustrate this aspect, we will extends the JAVA platform to support the EJB capabilities by applying EMF profile for EJB 3 to generate the code for data manipulation methods (Create/Retrieve/Update/Delete). Finally, the code source will be generated according to the JAVA Enterprise platform.

To perform these transformations we will use ATLAS Transformation Language (ATL) [36, 37, 38] which is a domain-specific language for specifying model-to-model transformations. It is a part of the AMMA (ATLAS Model Management Architecture) platform. ATL is inspired by the OMG QVT requirements [39] and builds upon the OCL formalism [31]. The choice of using OCL is motivated by its wide adoption in MDE and the fact that it is a standard language supported by OMG and the major tool vendors. It is also considered as a hybrid language, i.e. it provides a mix of declarative and imperative constructs.

In section *A* we present the metamodel of the JAVA platform [5], and in the next one we will present the different mapping rules performed to generate this JAVA model with the method bodies. Section *C* is dedicated to controller's methods generation. Section *D* introduces the EMF profile for the EJB 3 and CRUD operations generation.

A. Java meta model

The choice of the JAVA platform was arbitrary. Indeed, we had to choose a platform supporting oriented-object programming language. Also, the JAVA platform is a good example to illustrate the possibility of extension by applying profile. We have used the metamodel proposed by the OMG consortium [5] in which all the elements of the Java platform are represented with the same semantic as it specified by Sun Microsystems such as *JavaClass* , types packages and methods. The *EClass Method* was enriched by an additionally body attribute to represent the body code of the generated method.

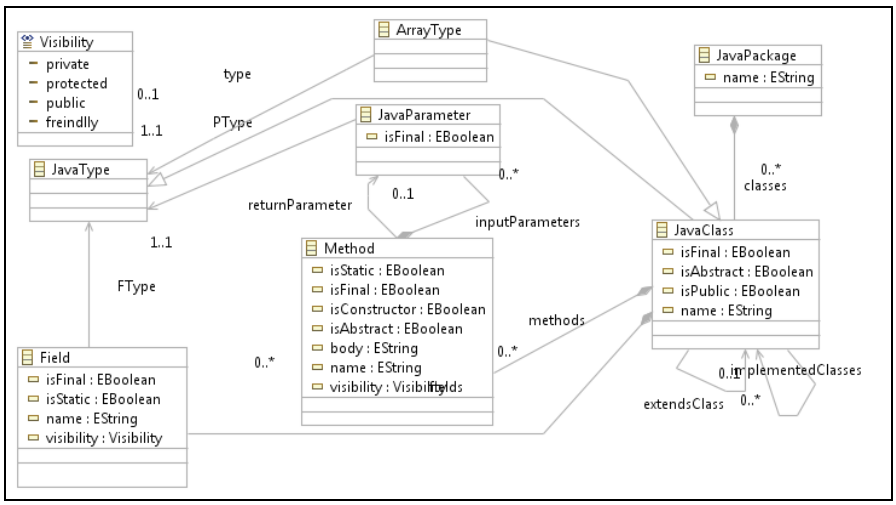

Figure 4: The JAVA metamodel [5]

Since the Java platform allows only simple inheritance, designer have to restrict their domain class diagram with respecting this java metamodel, otherwise code generated will not be compiled correctly. Designer can use interfaces and other mechanism to fix this issue at the DCD metamodel level. It is to remember that interfaces are generalized elements and a class can implement any number of them.

B. Mapping rules for Java model generation

In this section we present the mapping rules that allowed us to generate the java PSM from the PIM SDSIB. This last one is build based on the EPM toolset which means that every message or operation in this diagram has a specified semantic as it was determined in that toolset allowing thus its body's code generation. Before generating the body code of the methods, we will first generate the corresponding *JavaClass* for the objects that interact in the SDSIB based on the Domain Class diagram (DCD).

Every Model class in the SDSIB will be mapped to a *JavaClass* with same name. The other information like visibility, modifier, package name, supper class and fields are retrieved (using the helper [*helper context MMDCD!Class def: getSuperClass : MMJava!JavaClass*=]) from the DCD as shown in Figure 5 below. The fields are mapped from the Model class's Attribute in the DCD with their Type that is mapped also to the corresponding *JavaType*. If the package is undefined, a default package called models is used. The helper [*helper def: getPackage(className:String): MMJava!JavaPackage*=] allows packages generations. To map the various associations between classes which is represented by the EReference Extremity (i.e. Association end) in the DCD, a pseudo field of the same type as the object at the end of the association will be created and assigned to the class if the maximum multiplicity is less than or equal to 1, otherwise it will be mapped with a field of type array of the same type as the object at the end of this association.

To implement the design Pattern MVC a controller and at least a view will be created for each use cases. For the running example and regarding to the EPM toolset controller are prefixed by the term "Handler_" and Views by "View_". Also, we have opted for choice of one controller by uses case.

Referring to the *operations* which represent the interactions between objects, they are mapped to Java methods and their parameters to a *JavaParameter* with the corresponding *JavaType*. It is to remember that the operations are predefined and each one has a very accurate semantics allowing thus to generate automatically the source code for most of them. These operations are: *find, set, add, remove, get, create, calculate, display, check and constructor*.

Before generating the body code, first we have to determine the kind of the operation, for this we have developed in ATL the following helper [*helper context MMSDSIB!Operation def: getOperationKind: String*=], then another helper [*helper context MMSDSIB!Operation def :getMethodSignature: String =]* allows us to get the method signature with full detail : [*visibility modifier returnType methodName(param1Type param1,…)*], finally, the method's body will be generated with the helper [*helper context MMSDSIB!Operation def: getMethodBody: String*=]. This last one uses the *getOperationKind* helper to determine method type and generate then the code according to the semantic of this operation. It uses also the *getMethodSignature* to get complete name for the method including parameters and their types.

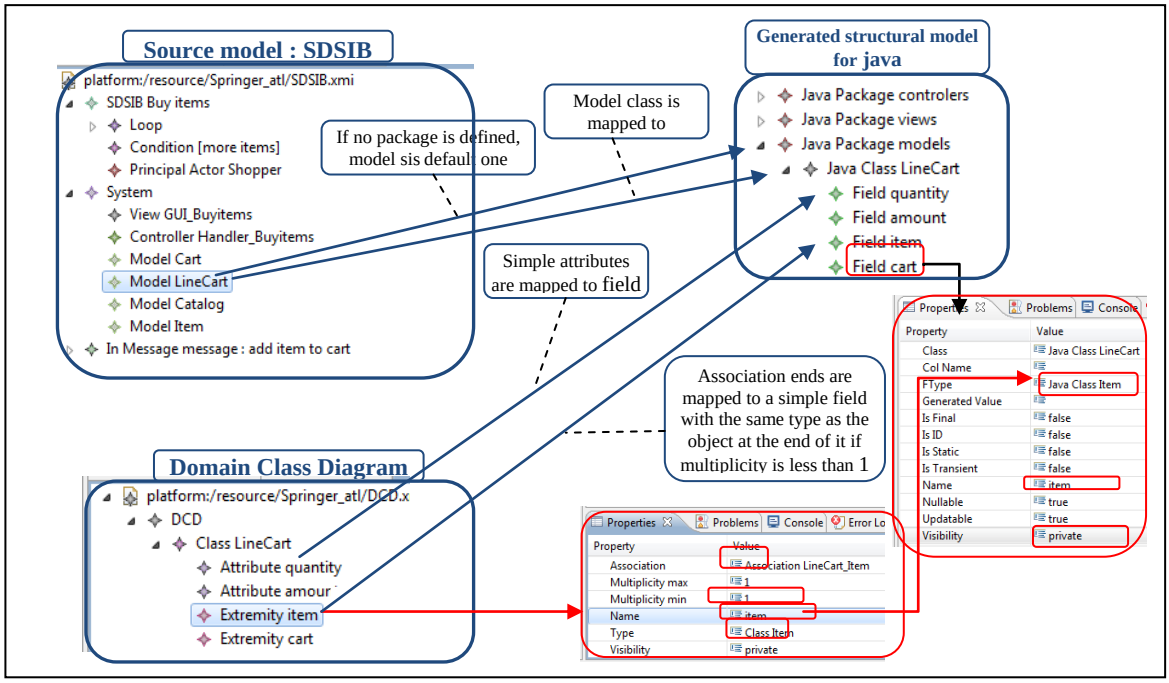

Figure 5: Mapping rules for Classes, attributes and association ends.

To generate the body of the method we need first to know whether it is a constructor or not. Indeed, the operations in the SDSIB do not specify this information. The helper [*helper context MMSDSIB!Operation def: isConstructor: Boolean*=] resolves this problem by analyzing the name of operation and its return type. A constructor has an undefined type (not void or anything else) and its name corresponds to a model class in the DCD. Thus, the helper *getMethodBody* tests whether it is constructor, if so then the body of the method will be in the form:

```
public ClassName (param1Type param1,…){
        this.param1=param1;
        …
}
```

The *ClassName* and constructor visibility are deduced respectively from the target and visibility properties of the operation. If the Model class subject to the constructor has a supper class then the method will contains at the beginning a call of the constructor of the supper class with the parameters related to it.

For example in the running example, the operation + *LineCart* has an undefined return type and corresponds to an domain class, it will be mapped to a constructor for the class *LineCart*.

The operation *create* consists in creating object according to the Creator GRASP Pattern. The target object for this operation is the responsible or the "creator" object for creating contained objects. This method uses the Constructor of the subject class to create and return the created object. The operation *createLineCart* is an example of this kind (Figure 6).

In the case of *add* and *set* methods which are used to express in term of EPM toolset a formed association. These methods allow thus to associate an object to subject class depending on the multiplicity of the association end. If this multiplicity equals 1 then the association is mapped by a simple field in the subject class with type of the associate object. In this case the *set* method will be used, otherwise, the association is mapped by field of type array and then a *add* method is needed to associate the object. The method *set* can be used also to express attributes modification i.e a simple setter. For example, while the operation *setQuantity* is used to modify the property quantity of the LineCart, *setItem* is used to associate the object *Item* passed as parameter with the subject class *LineCart*.

The operation *set* will thus generate a simple setter for fields with primitive type and a customized setter for field with type of *JavaClass* to express the association. Before associating this last one to the subject class, we check if the object is whether associated with another object, if so, we have to break this association first by setting this property to null or to remove it from the Collection property. The same algorithm is applied to the add method. Figure 7 illustrates these examples.

Operations remove and *setNull* are used to break an association between an object and a subject class. While, the remove is used when the associate object is mapped with a Set or a collection (array type), the set Null is used when the associate object is just a simple field. Thus, the *remove* method consists in finding the related object passed as parameter and removes it from the collection, but before that, it must be dissociated from the subject class by eliminating its reference. For example if we suppose that we want to dissociate the object *LineCart* with the Cart having a field with type array of *LineCart*, the generated code will be as following:

```
public void removeLineCart(LineCart lineCart){
if(lineCart !=null){
      if(this.lineCarts.contains(lineCart)
        this.lineCarts.remove(lineCart);
      lineCart.setCart(null);
}
}
```

The *setNull* method allows dissociating the object from the subject class in the case of a simple field. For example if we want to dissociate the object *Item* from the subject class *LineCart* the *setNull* method will use the setter for the property item of the subject class and put it at null as following:

```
public void setNull(){
      item.getLineCarts().remove(this);
      setItem(null);
}
```

The *find* method allows retrieving objects by identifier. It is used before associating or dissociating objects. To implement this method the GRASP pattern Creator was applied which assign the responsibility of finding objects to the subject class that contains these objects. In other words, it has a field of array type with this object. Finding the object by identifier is then done by comparing the ID of each object of this array with the key search. The first one that matches will be returned. The generated code will be as following for the operation *findItem(int code):*

```
public Item findItem(int code){
      for(Item item: items)
        if(item.getCode()==code)
           return item;
      return null;
}
```

The *check* operation is used to perform a logic test like authentication, comparison and others. It allows executing a set of logical test by comparing the parameters with the fields of the subject class and returning a Boolean value. The owner of the field is the responsible for the test as it is mentioned by the Expert GRASP pattern. In order to build the condition to test, we have first to determine the logical operator to use for each parameter which depends on the parameter type itself. Indeed, if the parameter type is a primitive type then the operator "==" is used, otherwise, it is a wrapper type (String for example) and then the "equals" operator is used.

```
public boolean checkAuth(String login, int code){
      return              login.euals(this.login)&&
code==this.code;
}
```

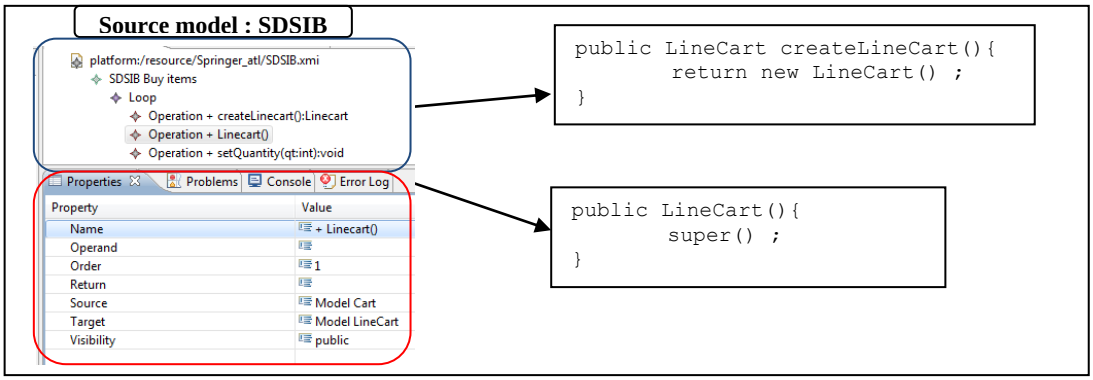

Figure 6: Generated code for constructor and create method.

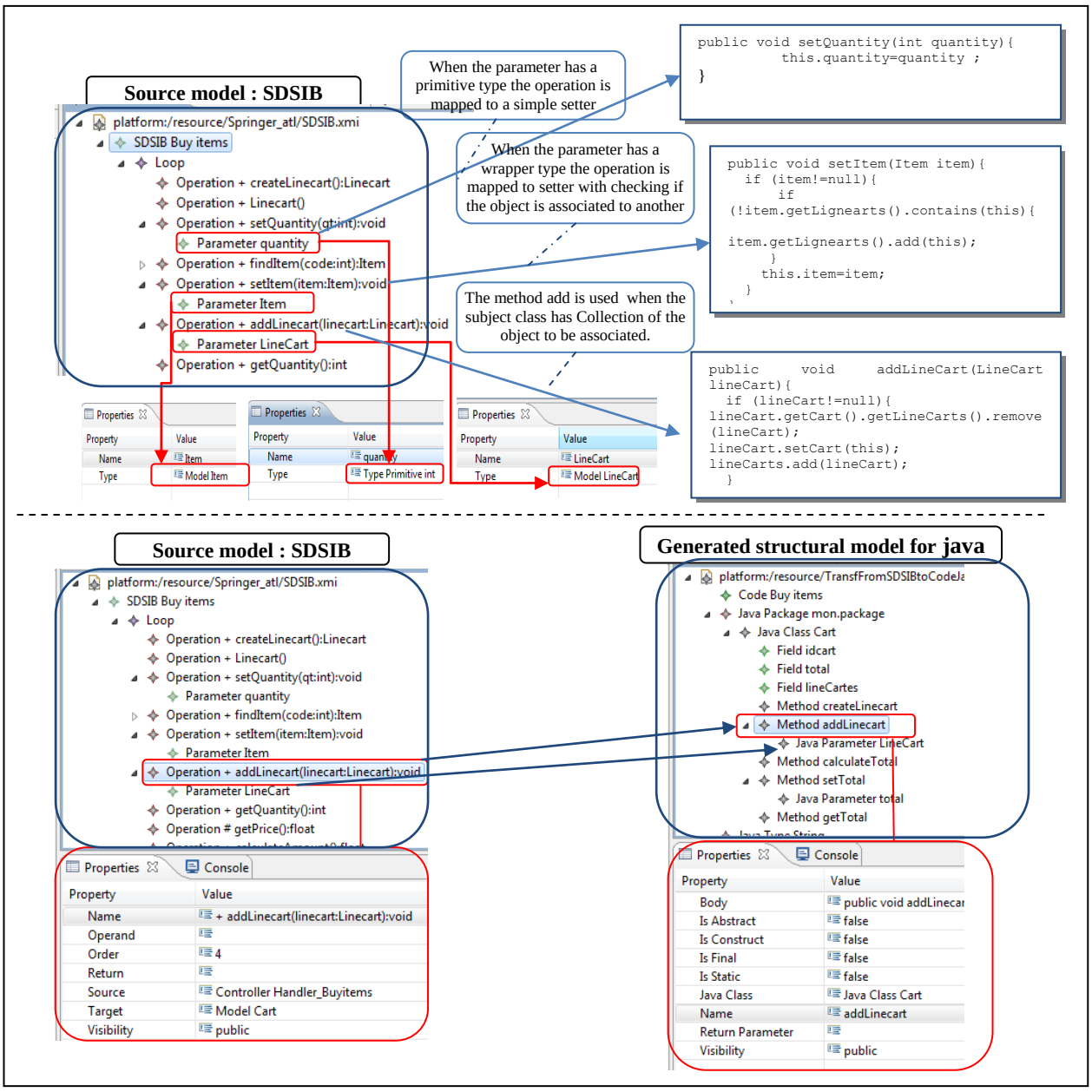

Figure 7: Mapping rules for add and set methods.

Finally, the destructor method allows destructing objects. Since we use Java as a platform to implement the code in this paper, we must remember that this platform does not define its own destructor. This task is delegated to a special process called the Garbage collector. However, we will map this operation with finalize method that is executed just before destructing the objects. Thus, developer can implement some code to free the resources used by the object or to dereference other objects in order to implement the "composition" association.

While the get operation is a simple getter of a given attribute, the display and calculate operation are used to express a methods that perform a complex or a business logic treatment. Generating their code source will need more details and a language to express it. The proposed generator generates code skeletons from the structural models for these methods and the programmers could complete their behavior manually by writing code in a specific programming language and for a specific technology platform. However, this approach breaks the portability, interoperability, and reuse objectives of MDA. UML 2.0 addresses the shortcomings of the UML in behavioral modeling by introducing UML action semantics, which defines atomic behavioral units that allow behavioral modeling of methods at the Platform Independent Modeling (PIM) layer. Although UML 2.0 was released a few years ago, there is no mapping from UML actions to any programming language and there is also no tool for code generation from UML action models. In [22] a tool for behavioral modeling was provided, it defined textual and visual notations for UML actions and built supporting editors. Further, it defined also a mapping from UML actions to Java and model compilers were built, which support the generation of complete and compile-ready applications including their behavioral parts. There is also many other works allowing the code generation for such method based on the Petri nets or defining a special DSML.

C. Code generation for Controllers

The source model of the performed transformation implements the design pattern MVC in order to produce software that is easy to maintain and to evolve. Therefore, besides model classes, we have a view and a controller classes. Note that the designer may indicate that a model class is the controller. The number of these controllers depends on the strategy made by the designer which can be a unique controller and here it is called the MVC2, it can be also a controller for each actor. We opt for a controller for each use case. In this paper we will generate also the code for these controller's methods to execute the use case.

Indeed, the used SDSIB as a source model for the transformation by the means of the EPM toolset enriched with many detail such as the source object which invokes the operation and the responsible for this operation as well as the order of this operation in the whole sequences and the different interaction operand within it is used. Thereby, the body generation of controller's method is possible. The controller coordinates the system objects to execute the use case, in fact for each message coming from the view a controller method is defined to provide a response. This method usually contains only methods invocation, either for object creation, attribute's modification or other depending on the type of operation and some control structure such as loops and conditions.

The code generator will generate for each use case a controller if no domain class is referred as a controller with methods corresponding for each incoming message (InMessage in the SDSIB). The helper [*helper context MMSDSIB!InMessage def: getControlerMethodBody:String=*] is used to generate the body of these methods. It provides the full signature with implemented code. Therefore, this helper focuses only on resulting operations of the concerned incoming message which have as source object attribute the controller. These operations are sorted by their order and are processed according to their return type. For example, before invoking a method we have first to get the instance of the object into which the method bellow, so we have to look first for the predecessor method that return the instance of this object. Or if the method needs a parameter, we have to look for it first from the incoming message parameter, if it is not found then we check of the returned parameter of the previous ones. Concerning the interaction operands, that have to be mapped to their corresponding control structure with the included code block corresponding to interaction within this operand, the helper *getControlerMethodBody* the helper [*helper context MMSDSIB!Operation def: startOfOperande: String=*] which for each operation insert the statement corresponding of the type of the interaction operand at the correct place. For this it uses the helper [***helper context*** *MMSDSIB!Operation* ***def****: getPreviousInteractionOperand(): MMSDSIB!InteractionOperand=*] to check whether is this operation is the first one in the interaction operand or not, if yes the structure is inserted here, otherwise it is already inserted and no statement is placed here. Also for each operation we have to check if it is the last one of the block so we can place the bracket to indicate the end of the block. For this we use the helper [*helper context MMSDSIB!Operation def: finOperande: String=*] which uses the *getPreviousInteractionOperand()* and *getNextInteractionOperand()* to detect the end of the block and also it is a recursive function because the interaction operand can be simple or a Combined fragment of many interaction operands so many brackets have to be placed correctly at end of the block.

According to running example, we have generated one controller with one method. Below the code that was generated.

```
public static void add_item_to_cart(int code, int
quantity){
        LineCart lineCart =cart.createLinecart();
        lineCart.setQuantity(quantity);
        Item item =catalog.findItem(code);
        lineCart.setItem(item);
        cart.addLinecart(lineCart);
        float Total =cart.getTotal();
        GUI_Buyitems.displayTotal();

}
```

D. Applying EJB profile

1.Introduction to profiles

UML is currently considered as the standard for object-oriented systems modeling. However, it remains largely undefined for specific domain. To resolve this problem, Domain-Specific Modeling Languages (DSMLs) have been introduced to provide designer modeling languages appropriate to their business domain. However, DSMLs should continuously evolve to adapt to the changing needs of the domain they represent. Changing the metamodel is a very costly process that requires changing its metamodel and possible re-creating the complete modeling environment.

UML has avoided these problems by promoting the use of profiles that provide a lightweight, language-inherent extension mechanism to the UML by defining custom stereotypes, tagged values, and constraints. Profiles allow for adaptation of the UML metamodel for different platforms (such as J2EE or .NET), or domains (such as real-time or business process modeling) [35]. The profiles mechanism is not a first-class extension mechanism. It does not allow to modify existing metamodels or to create a new metamodel as MOF does. Profile only allows adaptation or customization of an existing metamodel with constructs that are specific to a particular domain, platform, or method. It is not possible to take away any of the constraints that apply to a metamodel, but it is possible to add new constraints that are specific to the profile. Metamodel customizations are defined in a profile, which is then applied to a package. Stereotypes are specific metaclasses, tagged values are standard metaattributes, and profiles are specific kinds of packages.

One of the major advantages of UML Profiles is the ability to systematically introduce further language elements without having to re-create the whole modeling environment such as editors, transformations, and model APIs. In contrast to direct metamodel extensions, also already existing models may be dynamically extended by additional profile information without recreating the extended model elements. One model element may further be annotated with several stereotypes (even contained in different profiles) at the same time which is equivalent to the model element having multiple types. Furthermore, the additional information introduced by the profile application is kept separated from the model and, therefore, does not pollute the actual model instances.

2. EMF profiles

Since UML profile is located at the same level of abstraction as UML itself, it can only used to extend UML models. In this paper we use the EMF platform that uses the ECORE metametalanguage to create different metamodels, therefore we cannot use UML profiles to extend our DSML, hence the need to use EMF profile [22]. Also, the model transformation language ATL used in this proposal does not support UML model, only models based on Ecore metamodel can be used as a source models of the transformation.

The main objective of applying profiles in this paper is to show how the generated structural model of the performed transformation can be extended with new features before generating the implementation code. As an example we have applied the EJB3 profile [5, 6] that is presented in the next section. The use of profiles has many advantages like the ability of annotating model slightly as possible; hence, no adaptation of existing metamodels should be required. Also, It avoids polluting existing metamodels with concerns not directly related to the modeling domain separating annotations from the base model to allow importing only those annotations which are of current interest for a particular modeler in a particular situation.

To incorporate the profile mechanism into EMF, a language for specifying profiles is needed as a first ingredient. This is easily achieved by creating an Ecore-based metamodel which is referred to as Profile MetaModel that will be instantiated to create a specific profile, containing stereotypes and tagged values. Once a specific profile is at hand, users should now be enabled to apply this profile to arbitrary models by creating stereotype applications containing concrete values for tagged values defined in the stereotypes [35].

3. EJB3 profile

Here we provide an example for extending the generated java model of the performed transformation by applying the EJB 3 profile [5, 6, 40] established based on EMF profiles. Thus, we can generate an additionally source code like CRUD operation allowing persistence by the mean of the EJB entities and the EJB sessions.

Figure 8 presents the different elements of this EJB profiles represented by a set of Stereotypes and tagged values. We have presented also the metaclasses of the Java metamodel that were extended. For example, the stereotype “Field” is used to extends the metaclass *JavaField* (java class attribute) with the necessary information for the complete mapping of the related column such as nullable or updatable constraints, whether the field is an identifier or not and the column name that is necessary if the attribute and the column have different names. It defines both stateful which is a session bean that represents a conversational session with a particular client, such session objects automatically maintain their conversational state across multiple client-invoked methods, and stateless session beans that represent an EJB Bean without state for a client that will invoke only one method. The client of a session bean may be a local client, a remote client or a web service client depending on the interface provided by the bean and used by the client. An entity object represents a fine-grained persistent object. The client of an entity bean may be a local client or the client may be a remote client. An EJB Method is declared by a Java Method declaration within an EJB Home or Remote Interface. The EJB Method is an EJB Home Method, if declared within an EJB Home Interface, or an EJB Remote Method, if declared within an EJB Remote Interface. The declaration of an EJB Remote Interface extends the declaration of a Java Interface with EJB Deployment Descriptor elements for an EJB Enterprise Bean. The name of the EJB Remote Interface and the related EJB Home Interface are specified by remote and home elements in the entity or session element for the EJB Enterprise Bean.

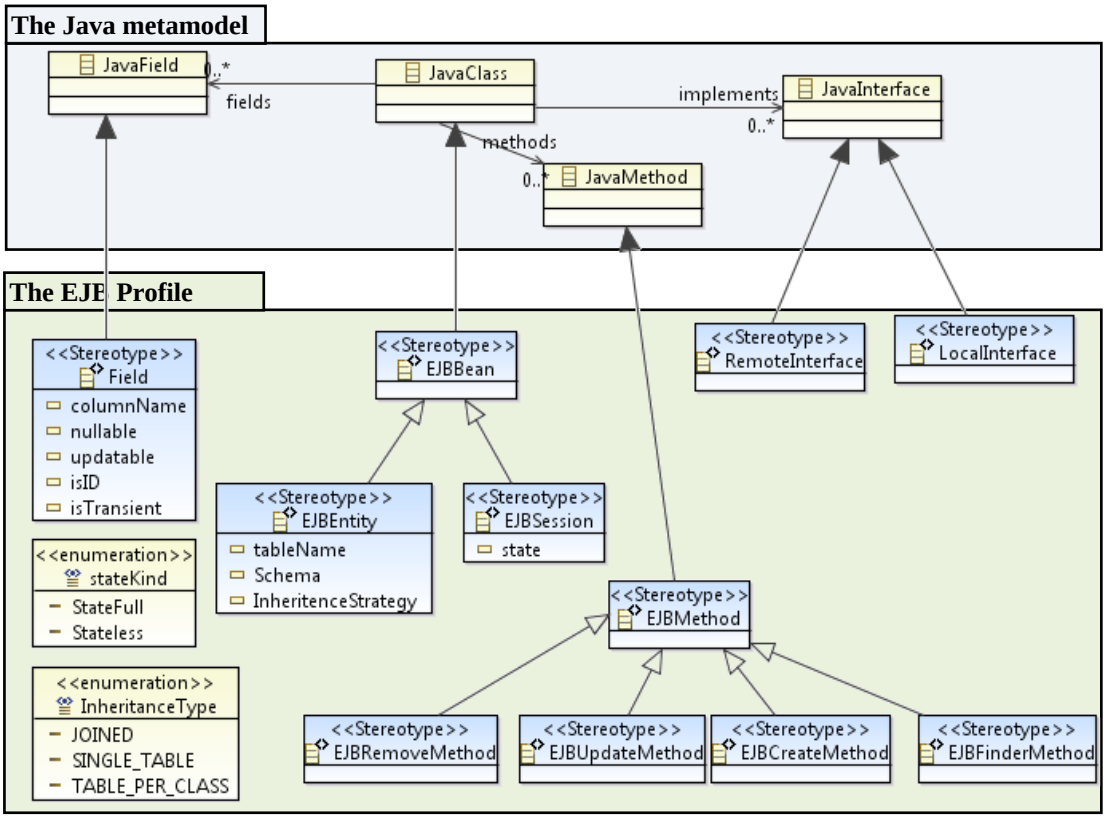

Figure 8: The EJB profile with the different extended metaclasses

The second transformation performed in this paper is a model-to-code transformation that will allow for transforming the generated structural model of the java model which has been enriched by applying the EJB profile, the code source according to the JAVA platform. Thus, for each Java class a java file will be generated containing the source code of the class including their methods (signature and body) and EJB annotation for the JAVA class that are stereotyped with "*EJBEntity*". Moreover, for each persistent class, a Stateless *EJBSessionbean* and its corresponding Remote or Home interface will be generated implementing the detailed source code of CRUD operation for data manipulation as well as the configuration file for persistence "persistence.xml". These operation are : save() which perform an initial save of a previously unsaved *EntityClass* entity; delete() to delete a persistent *EntityClass* entity; update() which allows to persist a previously saved *EntityClass* entity and return it or a copy of it to the sender, a copy of the *EntityClass* entity parameter is returned when the JPA persistence mechanism has not previously been tracking the updated entity; the *findById()* allowing the retrieve an *EntityClass* by identifier and the *findAll* to retrieve all instance of this *EntityClass*.

The Figure 9 below shows the generated code for the running example shown in Eclipse editor and Figure 10 shows the generated stateless bean and Interface for an example of a persistent Entity Class *EntityClass.*

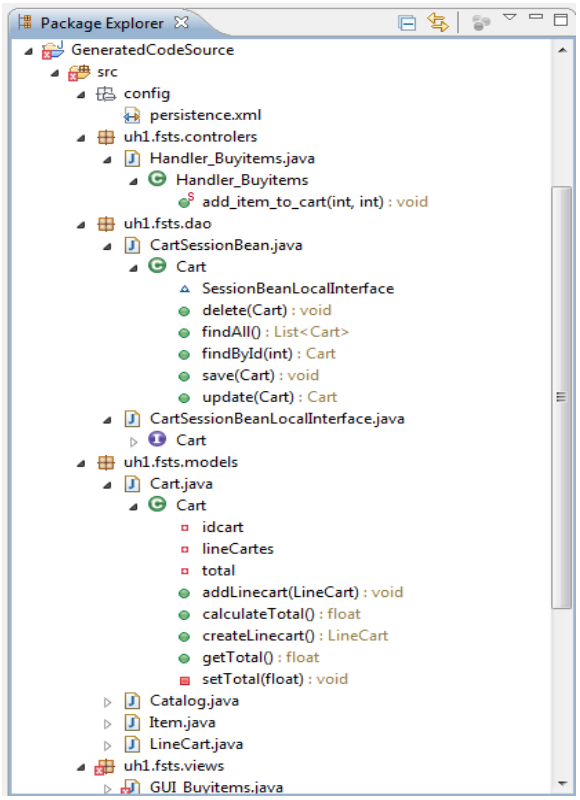

Figure 9: The generated source code for the running example.

```
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

@Stateless
public class EntityClass implements
EntityClassLocal, EntityClassRemote {

@PersistenceContext
private EntityManager entityManager;


public void save(EntityClass entity) {
                try {
entityManager.persist(entity);
        } catch (RuntimeException re) {
                throw re;
        }
}

public void delete(EntityClass entity) {
        try {
                entity =
entityManager.getReference(EntityClass.class,
                                entity.getId());
        entityManager.remove(entity);
        } catch (RuntimeException re) {
                        throw re;
        }
}


public EntityClass update(EntityClass entity) {
        try {
                EntityClass result =
entityManager.merge(entity);
                return result;
        } catch (RuntimeException re) {
                throw re;
        }
}

public EntityClass findById(IDType id) {
        try {
        EntityClass instance =
entityManager.find(EntityClass.class, id);
        return instance;
        } catch (RuntimeException re) {
                throw re;
        }
}


public List<EntityClass> findAll() {
        try {
        final String queryString = "select model
from EntityClass model";
        Query query =
entityManager.createQuery(queryString);
        return query.getResultList();
                } catch (RuntimeException re) {
                        throw re;
                }
        }

}
```

Figure 10: A universal example of the generated session bean for an Entity class

VI. EVALUATION

The best way to evaluate the presented generator is to compare it with a successful framework in its cadre. In related work section we present many tools and code generators implementing MDA for code generation. So far, there are no complete tools to manner the whole IT project using a software engineering process. Moreover, the generator uses the SDSIB that allows a true oriented object modeling for compartmental behavior of system and implements the MVC design pattern for easy software to maintain and evolve. Thus, system's objects with their methods body including controller's methods were generated. However, the generator does not allow the generation for special method that performs some business logic such as calculating and displaying. The result for some systems such as management of book loaning in a library is too impressing. In fact, in such systems, most operations are association formed, association broken or attributes modifications. Anyway, even the generator cannot generate completely the code in some case it allows to generate a large and important part of the software. This software ensures the following quality criteria of software engineering:

Evolution: since, the SDSIB PIM used in the transformation implements the design pattern MVC, the generated software is thereby easy to maintain and evolve. The resulting software is structured in many layers, so we can separate concerns of application. This architecture enables software evolution without affecting the whole structure. For example, if we need to change the API for data access, all we need to do is to change the DAO layer, other layers as GUI and Business Object will not need any adaptation.

Maintainability: During the development process, if any changes occur in the first increments such as Domain Class Diagram or even in the nominal scenario of the uses case, the first transformation will be executed to regenerate the SDSIB and then the second transformation will be performed to regenerate the source code automatically, especially that the generation of the code from the SDSIB does not requires designer involvement. Thus, we can reduce the time and cost of software production. Also, with implementing MVC design pattern the software became easier to maintain.

Extensibility: The approach we proposed to implement the code generator use a structural model to represent the code for the specified platform instead plain text. Thus the platform can be extended by other features or customized for some to deal with some constraint by applying an EMF profiles.

Table 2 below presents a comparison study of some code generator based on criteria of the generated code details and the architecture of the generator. We can see that the Stratego/XT which is a language for creating generators cannot be used directly for code generation. In fact we need first to create the generator using this language and then implementing the specific details for the chosen platform. While, classic code generator based on Petri nets still not yet standardized and used only for requirement validation purpose and not adapted to complex Oriented object systems, [12] provides a code generator that covers many phases of software development process and supports complex O.O systems, but it does not

Table 2: Comparison study of tools for code generation.

Criteria / code generator	Don't uses concrete syntax	Uses structural model	Support the entire dev. process	Supports O. O. systems	Generates CRUD operations	Generates Controllers	Generates method bodies	Generates GUI interfaces
Stratego/XT	✓	✓	×	×	×	×	✓	×
WebDSL	×	✓	×	×	✓	✓	✓	✓
Acceleo	×	×	✓	✓	✓	×	✓	✓
Classic Tools based on Petri nets	✓	×	×	×	×	×	✓	×
code generation from high-level Petri-Nets [26]	✓	✓	✓	✓	×	×	✓	×
Generating operation specifications [44]	×	×	×	×	✓	×	×	×
WebML	×	×	×	✓	✓	×	✓	✓
The proposal	✓	✓	✓	✓	✓	✓	✓	×

allows GUI interfaces, CRUD operation and controllers generation. WebDSL is efficient code generator for only web applications which allows the generation of the whole application including GUI and CRUD operation. However, it uses special concrete syntax and does not support the whole development process and also it is not suitable for complex O.O systems. WebML is also a code generator for web application but it does not use a structured model nor supports the whole development process. Acceleo is another relevant code generator that allows the generations of almost of the source code for the software, even it does not uses a structural model instead generating directly the plain text which affects the extensibility of the target language and it does not implements the MVC design pattern. This comparison study shows also that the proposal supports the whole development process and O.O complex systems and allows generating code without using a concrete syntax. It uses a structural model for the code to enable the languages extensibility and generates automatically the almost of the methods bodies including controllers. However, it does not allow generating GUI.

The main advantage of the proposed generator is that it covers the whole development process from requirement specification to code generation. Thus the generated code presents all the functional software quality criteria offered by such process like validity and reliability and robustness. Also, the increments are deduces automatically from each other. The generator does not require the involvement of the designer for generating every increment. For example, the code is generated from the SDSIB automatically which is also generated automatically from the SDSEB.

VII. CONCLUSION AND PERSPECTIVES

Due to the model driven architecture initiative of the OMG, automatic code generation is nowadays a topic of major interest. The main theme of this article is code generation by model transformation. The core idea is generation of code for the java platform from the SDSIB source model, one of the most important diagrams of the design phase and which is generated automatically by means of the EPM toolset developed in our previous work that extends the operation contract of LARMAN with a new semantic. The key idea behind this transformation is generating an intermediate structural model instead of the plain text directly. We have demonstrated that generating such intermediate model has many advantages like enabling extensibility of the target platform. For an example we have extended the java model to support EJB capabilities allowing thus generation of different data access operations through the Java beans entity and java session beans by applying an EMF profile for EJB3. The generated code provides more details such as complete methods signatures and methods body with full source code for the almost operations. An introduction of UML profile and EMF profile was also given in this proposal. We have also shown how we can produce software that is easy to maintain and evolve by implementing the design pattern MVC. Thus, the code for the controllers was also generated in this paper. This work this work crowns its predecessor that have presented an approach for automating software development process from the requirement specification to code generation.

In this paper, we described several techniques for code generation and especially these subscribed in the MDA approaches and concerns object oriented modeling. An overview of the relevant works to our topic was introduced as well as our previous works in which we have introduced an approach to automate the whole software development process. We have demonstrated that even there are several tools for code generation; these tools do not really allow a true automatic code generation for the whole software development process.

The proposed code generator is too efficient for true object oriented systems where almost interactions between objects to execute the uses case are creating or destructing objects, association formed or broken and attributes modification. In such systems the generator allows the generation of the totality of the code. On the other hand, for systems where there are many businesses methods such as calculating some values, the generator does not allow code generation for such methods. As

for future studies we intend to complete this code generator to generate source code for such calculate methods by using for example UML 2 action that allows representing compartmental behavior of method in object oriented systems by means the activity or state transition diagrams or in a specific DSML. Also, we intend to improve this generator to support generating GUI interfaces for different uses cases.

REFERENCES

[1] OMG: MDA GUIDE, Version 1.0.1 Object Management Group document number omg/2003-06-01 available at http://www.omg.org/docs/omg/03-06-01.pdf

[2] OMG :Object Management Group. www.omg.org http://www.omg.org/docs/omg/03-06-01.pdf

[3] Girault, C., Valk, R., 2003. Petri-Nets for Systems Engineering. Springer, berlin.

[4] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot: " From UML Profiles to EMF Profiles and Beyond " In Proceedings of the 49th Conference on Objects, Models, Components and Patterns (TOOLS'11), 2011. LNCS 6705, Springer.

[5] OMG, Metamodel and UML Profile for Java and EJB Specification , February 2004, Version 1.0 formal/04-02-02

[6] EJB profile, Java Community Process under JSR-000026 (see http://jcp.org/jsr/detail/26.jsp).

[7] EL BEGGAR Omar, BOUSETTA Brahim, GADI Taoufiq. Generating methods signatures from transition state diagram : A model transformation approach. Accepted for orale prsentation in Colloquim on Information Science and Technology CIST 24-26/10/2012 à fez.

[8] Bousetta B., EL Beggar O., Gadi T., Krouile A., Transformation of analysis class diagram to design class diagram using transition state diagram : AN MDE approach, International Workshop on Information Technologies and Communication, ENSEM 2011.

[9] EL BEGGAR Omar, BOUSETTA Brahim, GADI Taoufiq, Automating software development process: Analysis-PIMs to Design-PIM model transformation , Journal of Systems and Software - Manuscript ID JSS-D-12-00721, under review.

[10] Uzam, M., Jones, A.H., 1998. Discrete event control system design using automation Petri nets and their ladder diagram implementation. International Journal of Advanced Manufacturing Systems 14 (10), 716–728 (Special Issue on Petri Nets Applications in Manufacturing Systems).

[11] Lee, G., Zandong, H., Lee, J., 2004. Automatic generation of ladder diagram with control Petri Nets. Journal of Intelligent Manufacturing 15.

[12] Stephan Philippi, Automatic code generation from high-level Petri-Nets for model driven systems engineering, The Journal of Systems and Software 79 (2006) 1444–1455

[13] Warmer, J.B., Kleppe A.G.: Building a flexible software factory using partial domain specific models. In: Domain-Specific Modeling (DSM'06), Portland, Oregon, USA, pp. 15–22 (2006)

[14] Huang, S.S. Smaragdakis, Y.: Easy language extension with Meta-AspectJ. In: ICSE '06: Proceeding of the 28th International Conference on Software Engineering, pp. 865–868. ACM, New York (2006)

[15] Zook, D., Huang, S.S., Smaragdakis, Y.: Generating AspectJ Programs with Meta-AspectJ. In: Karsai, G. Visser, E. (eds.) Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings, volume 3286 of Lecture Notes in Computer Science, pp. 1–18. Springer, Heidelberg (2004)

[16] Kulkarni, V., Reddy, S.: An abstraction for reusable mdd components: model-based generation of model-based code generators. In: GPCE '08: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, pp. 181– 184. ACM, New York (2008)

[17] Suzuki, J., Yamamoto, Y.: Extending UML with aspects: aspect support in the design phase. Lecture Notes in Computer Science, pp. 299–299 (1999)

[18] Zef Hemel · Lennart C. L. Kats ·Danny M. Groenewegen · Eelco Visser, Code generation by model transformation: a case study in transformation modularity, Softw Syst Model (2010) 9:375–402

[19] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, Domain-Specific Program Generation, volume 3016 of LNCS, pages 216–238. Spinger-Verlag, June 2004.

[20] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, Rewriting Techniques and Applications (RTA'01), volume 2051 of LNCS, pages 357–361. Springer, May 2001.

[21] M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand and D. Perigot, editors, Workshop on Language Descriptions, Tools and Applications (LDTA'01), volume 44 of ENTCS. Elsevier, April 2001.

[22] Anis Charfi, Heiko Müller, Andreas Roth, Axel Spriestersbach, From UML Actions to Java, IDM 2009, Actes des 5emes journées sur l'Ingénierie Dirigée par les Modèles ,Nancy, 25-26 mars 2009

[23] Manoli Albert , Jordi Cabot , Cristina Gómez , Vicente Pelechano , Generating operation specifications from UML class diagrams: A model transformation approach, Data & Knowledge Engineering 70 (2011) 365–389

[24] Brambilla, P.F.M., Comai, S., Matera, M.: Designing web applications with WebML and WebRatio. In: Rossi, G. et al. (eds.) Web Engineering: Modelling and ImplementingWebApplications, Human–Computer Interaction Series. Springer, October (2007)

[25] Cáceres,B.V.P., Marcos, E.:AMDA-based approach forweb information system development. In: Proceedings ofWorkshop in Software Model Engineering (2003)

[26] Pastor, V.P.O., Fons, J.: OOWS: a method to develop web applications from web-oriented conceptual models. In: Web Oriented Software Technology (IWWOST'03), pp. 65–70 (2003)

[27] Pierre-Alain Muller, F.F., Studer, P., Bézivin, J.: Platform independent web application modeling and development with Netsilon. Softw. Syst. Model. 4(4), 424–442 (2005)

[28] Kraus, A.K.A., Koch, N.: Model-driven generation of web applications in UWE. In: Model-DrivenWeb Engineering (MDWE'07), Como, Italy, July (2007)

[29] Voelter, M., Groher, I.: Handling Variability in Model Transformations and Generators. In: Domain-Specific Modeling (DSM'07) (2007)

[30] Craig Larman, Applying UML and Patterns, 3rd Edition, Prentice Hall, 2002, ISBN 0-13-148906-2

[31] OMG, « Object Constraint Language (OCL) Specification, version 2.0 », 2006. http ://www.omg.org/spec/OCL/2.0/.

[32] Object Management Group, Inc. Unified Modeling Language (UML) 2.1.2 Infrastructure,November 2007. Final Adopted Specification.

[33] Object Management Group, Inc. Unified Modeling Language (UML) 2.1.2 Superstructure,November 2007. Final Adopted Specification.

[34] Rumbaugh, Jacobson, et al. - The Unified Modelling Language Reference Manual - 1999

[35] Object Management Group, Inc. Meta Object Facility (MOF) 2.0 Core Specification, January2006. Final Adopted Specification.

[36] Freddy Allilaire , Jean Bézivin , Frédéric Jouault , Ivan Kurtev, ATL – Eclipse Support for Model Transformation (2006) : Proc. of the Eclipse Technology eXchange Workshop (eTX) at ECOOP

[37] ATL - a model transformation technology, http://www.eclipse.org/atl/, 2012.

[38] F. Jouault, F. Allilaire, J. Bezivin, I. Kurtev, ATL: a model transformation tool, Science of Computer Programming 72 (1–2) (2008) 31–39.

[39] Object Management Group, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, OMG Adopted Specification ptc/05-11-01, 2005,

[40] Linda DeMichiel (Sun Microsystems), Michael Keith (Oracle Corporation), JSR 220: Enterprise JavaBeansTM,Version 3.0, May, 2006