UML Collaboration Diagrams and Their Transformation to Java

Gregor Engels¹, Roland Hücking², Stefan Sauer¹, and Annika Wagner¹

¹ University of Paderborn, Dept. of Computer Science, D 33095 Paderborn, Germany
   engels|sauer|java@uni-paderborn.de
² SAP AG, Lo. Dev. PP-PI, Neurottstr. 16, D 69190 Walldorf, Germany
   roland.huecking@sap-ag.de

Abstract. UML provides a variety of diagram types for specifying both the structure and the behavior of a system. During the development process, models specified by use of these diagram types have to be transformed into corresponding code. In the past, mainly class diagrams and state diagrams have been considered for an automatic code generation. In this paper, we focus on collaboration diagrams. As an important prerequisite for a consistent transformation into Java code, we first provide methodical guidelines on how to deploy collaboration diagrams to model functional behavior. This understanding yields a refined meta model and forms the base for the definition of a transformation algorithm. The automatically generated Java code fragments build a substantial part of the functionality and prevent the loss of important information during the transition from a model to its implementation.

Keywords: Collaboration diagram, methodical guidelines, code generation, Java, pattern-based transformation algorithm

1 Introduction

The Unified Modeling Language (UML, [8,9,13]) provides a variety of diagram types for an integrated specification of both the structure and the behavior of a system. Collaboration diagrams belong to the behavioral diagrams like sequence diagrams, statecharts and activity diagrams.

Tools to support the development of software, so-called CASE tools, often do not only support the analysis and design of systems, but also contain code generators to automatically create code fragments of the specified system in a target programming language. Unfortunately, the capabilities of code generators to transform the design to an implementation are often restricted to produce class definitions consisting of attributes and operation signatures captured in class diagrams, but not the methods to implement the procedural flow within the operations.

Using also behavioral information for code generation prevents the loss of substantial information during the transition of a model to its implementation. Existing approaches in this direction transform statecharts into executable code
Statecharts are used as object controllers for specifying when an object is willing to accept requests. CASE tools supporting code generation from statecharts are e.g. Statemate [15], Omate [5], and Rhapsody [11].

In contrast, it is our aim to transform the specification of the functional behavior of objects into code fragments. The functional model can be described in terms of interactions between objects in an abstract way by UML interaction diagrams.

The only tool known to us that is capable of generating code from interaction diagrams is Structure Builder [16]. Sequence diagrams are used there, but code is not directly generated from them, but from an intermediate representation called Sequence Methods. Sequence Methods are based on the concept of Interaction Graphs [14], resulting from the Demeter project [7], which are directed labeled trees with nodes representing object variables and edges representing actions. They basically resemble a representation of additional information that, in agreement with our approach, needs to be interactively entered by a developer to extend the interaction modeled in UML diagrams. Such information being necessary for the generation of working Java code is e.g. how objects can be accessed, how they are transported between methods, and instantiation of links etc. These details can not be specified in sequence diagrams, but most of them are already captured in collaboration diagrams.

Thus, we selected collaboration diagrams from UML interaction diagrams as the source for the transformation process since, in contrast to sequence diagrams, they do not only supply the message flow information of an interaction, but also the underlying structural information building the context of the interaction, i.e. the links via which messages are sent. Additionally, we stay within the diagram types of UML whereas Sequence Methods are outside the UML.

Java was selected as the target language because it is a purely object-oriented programming language of growing importance and it offers concepts for concurrent programming to extend the transformation mechanisms to parallel flow of control.

The paper is organized as follows: In Sect.2, we introduce the main features of collaboration diagrams and state methodical guidelines for their deployment. A general overview of the transformation approach for collaboration diagrams based on the transformation of class diagrams is given in Sect.3. The next section introduces a refined meta model which forms the basis for a detailed description of the transformation algorithm for collaboration diagrams in Sect.5. The paper ends with some concluding remarks and perspectives.

Further details can be found in an extended version of this paper that is available as a technical report (see [4]).

2 Deploying UML Collaboration Diagrams

In this section, we outline a methodical approach on how to deploy UML collaboration diagrams to model functional behavior. This approach is based on the general UML specification [8, 9], but it extends it by additional pragmatic
guidelines and constraints. A systematic usage of this approach will ensure that collaboration diagrams describing the functionality of methods can automatically be translated into corresponding Java code. In the following, we assume that the reader is familiar with the standard UML notations (see [8, 13]).

In general, collaboration diagrams can be used to model system functionality or more precisely the control flow within a system. This is described by sending messages between instances of classes. Collaboration diagrams are feasible to model not only the behavioral, but also the structural context of such an interaction, called a collaboration.

In [8], the following two possibilities, among others, of deploying collaboration diagrams in the above sense are introduced:

- Method: specify the implementation of an operation as an interaction.
- Use case: describe the functionality of a main operation of a system on an abstract level.

Both kinds of usage differ not only on their level of abstraction, but also in their main intention. Whereas use cases are deployed in earlier phases of modeling, the method-oriented usage is already close to implementation. Use cases describe scenarios. They are intended to exemplify a certain situation, i.e., very often they describe only one possible control flow path. In contrast, within a method specification, the general situation with all possible control flow paths has to be modeled. As a consequence, collaboration diagrams are used on the instance level in the first case, describing the interaction of different objects with each other. In the second case, they are used on the type level possibly containing iterations or conditional flows [9]. Type level modeling is in accordance with the specification of methods within classes of object-oriented programming languages.

With this background, two main steps can be identified within the development process producing the system's functionality. The first task is stepping from different scenarios to the general situation. And the second task is stepping from a model of the general situation to its implementation. In this paper, we concentrate on the second task, where one type level model for each method, i.e., exactly one collaboration diagram per method, serves as the basis for automatic code generation.

The first task of combining different instance level collaboration diagrams specifying the same operation can not be done automatically in the general case. Collaborations define views on the classes specified in the class diagram. Therefore, problems in combining several collaboration diagrams resemble typical problems of view integration [3]. Input by a developer is necessary to handle conflicts or to specify details of combination like contextual constraints or conditions. This interactive intervention should receive support by code generation tools. Situations where an automatic combination is possible are, e.g., mutual exclusive execution conditions for different occurrences of the same operation for branching as well as iteration.

On the other hand, collaboration diagrams are not able to fully model the functionality of an operation. One restriction is their inability to model oper-
ations on data types, i.e., primitive base types like Integer, Real or predefined enumeration types like Boolean, whose values do not possess an identity. Thus, collaboration diagrams can not serve as a fully-fledged visual programming language. Moreover, usually not all aspects of a system are completely modeled. Exception handling, for example, will usually be separately specified and added later in the implementation. For these reasons, code generation from collaboration diagrams is by their definition restricted to object interactions. Generating this kind of working code, prevents the loss of information during the step from modeling to implementation and simplifies the task of transition what states our objectives.

Before we start explaining our approach in more detail, we introduce a running example for a system to be modeled. Figure 1 shows the class diagram of an example application where a Company object is related to zero or more Store, Order, and Delivery objects. A Store is related to multiple Delivery objects, which in turn are related to one Customer and one Order. A Customer can place several instances of Order, and one or none Delivery objects belongs to an Order.

![Class Diagram](image)

**Figure 1.** Class diagram of a modeling example

A typical scenario within that setting is the situation where a customer orders a product from the company. On the use case level one would model that scenario by sending an order from a customer to the company, followed by forwarding that order from the company to one of its stores, followed by delivering the ordered products from the store to the customer.

After the step of refining and combining different use cases into a method-oriented specification one might end up with a collaboration diagram for a method **processOrder** as depicted in Fig.2. Here, the company first obtains the product number `pNr` and the ordered amount `a` of that order using defined access functions. It then checks all stores to find one that can supply the requested amount of the demanded product. A delivery is created, and the selected store is called to send it out. Finally, the delivery is added to a container holding all deliveries of the company.
We will now introduce methodical guidelines as a foundation of the later on presented transformation approach. As a consequence of deploying a unique collaboration diagram for specifying the implementation of one operation, two basic model entities build the basis for the forthcoming concepts:

- The **specified operation** is the operation whose implementation is modeled by the collaboration diagram (processOrder in Fig. 2).
- The **target object** is that object on which the specified operation is called.

The specified operation belongs to the class of the target object, its signature must be declared in the operation compartment of the corresponding class in the class diagram (:Company in Fig. 2).

As a result of the refinement and combination of different scenario-oriented collaboration diagrams we obtain a collaboration diagram with a single level of nesting. Thus, we specify which operations are called in the specified operation directly, but we do not consider those that are subordinately called within these nested operations. We consider this to be meaningful when we specify the implementation of an operation, since the subordinately called operations belong to collaboration diagrams for the nested operations. This is alike the definition of procedures and procedural calls in programming languages. As an implication, the target object is the sender of the call message for all operations in a collaboration diagram except for the specified operation.

One end of a stereotyped link must be directly connected to the target object (see Fig. 2). Conventional links based on associations can also be indirectly connected to the target object. They can be accessed by traversing along a path of links of which only the first may be a stereotyped link. If a link with the stereotype «parameter» (e.g. to :Order in Fig. 2) is used, then a reference to the object on the other end of the link must be transported to the target object as a parameter of the specified operation. The names of objects that are connected to the target object by a «parameter» link must be identical to the parameter names of the corresponding operation in the collaboration diagram.

Stereotyped links of kind «local» (e.g. between :Company and :Store in Fig. 2) depict that the linked objects are locally accessible within the specified operation.
This stereotype can be used either if the reference to the linked object was obtained as a return value of a previously called operation or if the linked object was initialized by calling a constructor within the specified operation. The same restrictions apply to stereotyped links of kind «global». Additionally, global variables can also be initialized within another collaboration, i.e. in a different collaboration diagram.

To prevent ambiguities, role names on association links are needed in the case that multiple links exist between two objects. Calling a constructor across an association link implies that both the receiver object and the link are implicitly \{new\} (see 4: in Fig.2). Thus, the constraint is optional. In contrast, adding to and deleting from multiobjects (notation for container in UML collaboration diagrams) can be explicitly defined by the modeler in order to specify the exact sequence of messages (see 6: in Fig.2).

Objects may not be marked with the constraint \{destroyed\} because Java does not contain a predefined destructor. Otherwise, one would have to solve the problem that all references to that object must be deleted to make the garbage collector delete the object, even those references specified in other collaborations.

Further details of the implications of our approach will be shown in Sect.4 where the refined meta model for collaborations is presented.

3 Transformation Approach

In Sect.2, methodical guidelines on how to deploy collaboration diagrams have been explained. Following these, all collaboration diagrams to be translated have a well-formed structure. This is an important prerequisite and enables a systematic translation of collaboration diagrams into corresponding Java code.

The translation algorithm for collaboration diagrams is based on a standard algorithm for translating class diagrams. The underlying idea is to translate class definitions into corresponding Java class definitions and to translate associations into bi-directional references between the two participating classes. This standard algorithm has been refined, e.g., with respect to automatically generated "get" and "put" access operations for attributes or a generic search operation to select certain objects from a set of existing objects. Further details on the refined class translation algorithm can be found in [4].

The basic idea of the overall transformation algorithm from a class diagram and associated collaboration diagrams into corresponding Java code is to identify standard patterns in a given diagram and to translate those patterns into corresponding Java code. This pattern-based transformation algorithm will be presented in a technical, formal way in Sect.5. Here, we give two simple examples to sketch informally how this pattern-based translation works.

First, Fig.3(a) shows a part of the collaboration diagram given in Fig.2 where operation getpNr() is sent via a parameter link with role \(\sigma\) to an object of class Order. This collaboration diagram is depicted in the lower left part of Fig.3(a), while the corresponding class diagram can be found in the upper left part. The
right hand side shows the generated Java code for such a parameter link pattern within a collaboration diagram.

Second, Fig.3(b) shows another pattern taken from Fig.2. Here, the collaboration diagram comprises a pattern consisting of a local link combined with a newly created object of class Delivery. The resulting Java code comprises a definition of a local variable d of type Delivery, as well as the invocation of the constructor of class Delivery in order to create a new instance.

The complete structured and pattern-based transformation algorithm will be explained in Sect.5. In order to be able to describe certain patterns within a class or collaboration diagram, a uniform internal representation of diagrams is an important prerequisite. As known from the UML language definition, such an internal representation can best be defined by a meta model. Therefore, the next section will present an adapted UML meta model, which incorporates the restrictions introduced in Sect.2.

Figure 3. Transformation of (a) parameter and (b) local links into Java code

4 Refined Meta Model

Based on the UML meta model, we present a refined meta model for collaborations, that has been adapted according to the assumptions and restrictions described in Sect.2. The methodical guidelines for deploying collaboration diagrams to model method implementations have been integrated and are thus
reflected on the meta model level now. Since the transformation algorithm presented in the next section is based on this meta model representation, the methodical guidelines also affect the code generation. The benefits of this adapted meta model are two-fold. First, the methodical guidelines have become part of the modeling language. Thus, only well-structured collaboration diagrams can be instantiated from this meta model. Second, the adapted meta model shows a granularity which is very well suited as basis for the pattern-based transformation algorithm.

Figure 4 depicts the changes to the original UML meta model. Elements that are replaced or deleted are crossed out, while new or changed meta classes and associations are shaded. Note that associations connected to new classes are also new even if they are not explicitly marked for simplicity. Some classes from other meta model packages of UML have been included, but all changes to existing associations have been marked.

Due to the use of collaboration diagrams for specifying the implementation of operations, the upper left occurrence of class Classifier disappears from the meta model. Additionally, we argued (see Sect.2) that the implementation of every operation is specified by exactly one collaboration diagram what is reflected by the one-to-one association between the corresponding classes.

Two new associations between the meta model classes Collaboration and Message are added to simplify navigation through the meta model according to the specified message sequence. The multiplicities on the predecessor association are changed and the activator association is removed because the transformed diagrams contain only one level of nesting. For the same reason, the association to ClassifierRole with the role name sender is bent, now connecting ClassifierRole and Collaboration: The sender for messages within this collaboration is the target object on which the specified operation is called (see Sect.2).

To account for the distinct algorithmic transformation of the different link types, we introduce meta model classes for stereotyped links LocalEdge, GlobalEdge, ParameterEdge, and SelfEdge, and the abstract super class Edge. The new class EdgeEnd builds the counterpart to AssociationEndRole for the stereotyped links. We replace the composition relation between AssociationRole and AssociationEndRole by two associations modeling directed association roles. This is possible since we have only one level of nesting and we restrict the transformation to binary associations. The transformation algorithm uses the roles to and from to traverse association links in the direction of message flow.

New is also the abstract meta class Node as a super class of ClassifierRole. Its purpose is to hold an attribute of type N.T.Kind representing the default constraints {new} and {transient} that can be attached to an object (ClassifierRole) in a collaboration diagram. An equivalent attribute of the super type N T T Kind has been added to the class AssociationEndRole. Due to this extension, the mapping of constraints on the appropriate subclasses of the meta model class Action [8] is no longer needed.

We further introduce a meta model class Expression and subclasses (not shown on the diagram) for data values, operators and their operands, etc. to
Figure 4. Extended meta model for transforming collaboration diagrams
decompose expressions in their components. This enables the definition of access functions for objects that are referenced by a link based on an association. The recurrence attribute of the class Action is changed into an association. Simple expressions are either instances of a base type or a variable identifier.

If an operation yields a result, the return action in UML is specified by a separate return message [8]. In contrast, the return message is not explicitly modeled in the refined meta model. Instead, the name of a variable for the return value is explicitly stored in the meta model class VarIdent. This variable name is related to either an attribute of the target object, a stereotyped link, or an association link, represented by alternative associations to Attribute and Edge. The role name belonging to such an edge is equivalent to the variable name. The meta model class MethodCall is used to specify a method called on a variable identifier using the dot notation.

Another meta model class GlobalVar is added to hold the names of global variables that are referenced by «global» links within all collaboration diagrams.

Only three subclasses of the meta model class Action remain in the meta model for the transformation of collaboration diagrams to Java, since Java has no predefined destructor. For every instance of Action or its subclasses, exactly one Request instance is linked.

5 Transformation Algorithm

In this section, the algorithm for transforming collaboration diagrams to Java is specified in a rule-based way. In order for the algorithm to work correctly, collaboration diagrams are assumed to be syntactically and static-semantically correct. Moreover, the whole model consisting of a class diagram and a collaboration diagram for each operation defined in the class diagram has already been translated into an instance of the meta model as described in Sect.4.

We use a kind of meta rules consisting of a rule scheme and an additional pattern. The rule scheme describes the generation of syntactically correct Java code. It has the form of a context free rule expression. But it is still independent of a concrete collaboration diagram. It contains two kinds of non-terminal symbols. The first are non-terminals in the usual sense replaced by sequences of non-terminals and terminals by the application of rules. Only those will be called non-terminals in the following. The second kind are parameters of the rule scheme, which allow its instantiation for a concrete diagram to be transformed. These parameters are formulated using terms of the meta model. This approach stems originally from the compiler construction area, where it is known as a two-level grammar approach ([17]).

The pattern is a part of an instance diagram of the meta model. It is used to represent those parts of a concrete diagram which shall be actually transformed. Hence, the occurrence of the pattern in the instance diagram for the example application for which code shall be generated serves as an application condition for the whole meta rule to be applied. Moreover, the concrete occurrence links together the general code generation possibilities, described by the rule scheme,
and the actual elements of the concrete collaboration diagram that has to be transformed. The parameters of the rule scheme occur in the pattern and can hence be replaced by actual values in order to instantiate the rule scheme.

Figure 5 shows two meta rules for the transformation of class diagrams. These meta rules will be used in the following to illustrate how the algorithm is specified in principle. On the left hand side, the part of the class diagram actually translated by the meta rule is shown. In the middle, we give its translation to part of an instance of the meta model, which forms the pattern. On the right hand side, the rule scheme for generating Java code is shown. Words in capital letters denote non-terminal symbols, whereas words in small letters denote terminal symbols if they are underlined, or they denote parameter expressions over the pattern if not. These parameters will be evaluated to terminal symbols as soon as a concrete occurrence of the pattern is chosen, leading to an instantiation of the rule scheme for the concrete diagram.

![Figure 5. Meta rules for class diagram](image)

The first meta rule shown in Fig.5 allows to transform a single class into the frame for a class declaration in Java. Here, c refers to the instance of classifier which represents the class in the instance of the meta model. Hence c.name is a parameter which will be replaced by the name of the class, i.e., the concrete value of this attribute in an occurrence of the pattern. The non-terminal symbols STARTc, ATTRIBUTESc, and METHODSc will also be instantiated with more concrete non-terminal symbols. The name c of the classifier object is used to keep track of the concrete classifier object currently dealt with during the next steps of code generation. It already determines partly the occurrence of the pattern belonging to the meta rule for replacing this non-terminal. The second meta rule shown in Fig.5 serves for the generation of the method frames for each operation defined in the class diagram in an analogous way. The instantiation of the non-terminal symbol COLLABc will be replaced by the code generated for the collaboration diagram of this operation.
Note that the meta rules are only applied once for each occurrence of the according pattern. Different occurrences may overlap. For example, in case of the second rule the same classifier object may occur as owner of an operation and as parameter of another or even of the same operation.

Consider again our example application introduced in Sect.2. The class diagram shown in Fig.1 can be transformed into Java code using the above meta rules in the following way: We search for an occurrence of the pattern of the first meta rule in the instance diagram of the meta model. Classifier c is mapped to classifier com, whose name attribute has the value "Company". For this occurrence of the pattern we instantiate the rule scheme leading to

\[
\text{START}_{com} \rightarrow \text{public class Company \{ }
\]

\[
\text{ATTRIBUTES}_{com} \downarrow
\]

\[
\text{METHODS}_{com}
\]

In the second step, we want to replace the non-terminal METHODS_{com}. This could be done by using the second meta rule. But we need a particular instantiation of the according rule scheme (METHODS_{com} instead of METHODS_{c1}). Hence the occurrence for the pattern has to obey this constraint. If an occurrence is found that maps operation o to operation procOrd with name processOrder and visibility public, the rule scheme can be instantiated to

\[
\text{METHODS}_{com} \rightarrow \text{public void processOrder \{ ARGSProcOrd \} \} \downarrow
\]

\[
\text{COLLAB}{\text{ProcOrd}}
\]

With the above two rules, we can deduce a primitive class frame from the start symbol START_{com}. Note that the instantiation process leads to a set of different start symbols since the generated Java code has to be stored in different files.

We now advance to the transformation of collaboration diagrams. We start with meta rules for replacing the non-terminal COLLAB_{o} by a sequence of other non-terminals in order to determine the structure of the generated code of the body of a method. First, the local variables have to be declared. Then, we invoke the methods in the order which is indicated in the collaboration diagram by the sequence numbers. Finally, we have to add newly inserted links, which are not used to invoke a method, and to remove links, which are indicated as destroyed. The according meta rule is depicted in Fig.6.

In the sequel, the first two meta rules generated by this substitution are explained in detail. Figure 7 shows the meta rule for declaring local variables. Remember that we also assume that indirectly declared local variables (return values of method invocations referencing objects) are to be represented as local edges in the instance of the meta model. Hence each LocalEdge uniquely represents a local variable, the name of which is stored as the RoleName of its EdgeEnd. The LocalEdge belongs to the collaboration of operation o. The type of a local variable is given by the name of the base (classifier) of the target of the EdgeEnd. This information is represented in the pattern. Moreover, it is used
in the rule scheme by the parameters c.name for the type and e.RoleName for the name of the local variable. We add the possibility of declaring more than one local variable within the same operation o by repeating the non-terminal LOCALVAR_DECLo. Again, different applications of the meta rule imply different occurrences of the pattern ensuring that each local variable is declared only once. The meta rule in the lower part of Fig.7 serves for the end of the declaration process. The rule scheme replaces the non-terminal LOCALVAR_DECLo by the empty string. It may only be applied, if the upper meta rule is not applicable any more.

Now we come to the generation of the real body of an operation, namely the invocation of methods. Generally, we have to generate the method invocation code in the order indicated by the sequence numbers in the collaboration diagram. This order is represented in the meta model by the predecessor edge between messages and by the edge assigning the first message to a collaboration. Hence, we have three kinds of meta rules: The first kind serves for invoking the first method. The second kind traverses the predecessor edge from the previous to the next message. The third kind ends the process. Meta rules of the last kind look like the last one discussed for the local variable declaration above. They are neglected in the following.
For the first two kinds of meta rules, we additionally have to distinguish many different cases: whether the receiver of the message is a multiobject, whether it is a newly created object, whether a parameter, a local or global variable or an existing resp. new association is used to send the message to the receiver, whether a return value is expected or not, whether the method invocation is conditional or an iteration, and whether the method itself or the method of the super class is called. Due to space limitations, we are not able to present all according meta rules in this paper (see [4] for an exhaustive presentation).

Instead, Fig.8 shows as an example a meta rule for invoking a method on a parameter object. It is a rule of the first kind, meaning that the method invocation is the first one in the actually transformed collaboration. A method for operation o is invoked. The kernel of this method invocation is that an operation r.name is called on the parameter object referred to by e.RoleName. The arguments for this call are generated from the non-terminal symbol ARGS_{r}. We omit a more detailed view on that, since we left out the specialization of class Expression in the meta model in Sect.4 that is necessary for this purpose. The meta rules for invoking an operation on a local or global variable look quite similar. Only the parameter edge in the pattern is replaced by a local or global edge, respectively. The transformation of a "self" link is handled analogously, distinguishing between using a this-pointer or a super-pointer to call a redefined method of a super class.

![Collaboration Diagram](image_url)

**Figure 8.** Meta rule for method invocation on parameter object

The pattern of the meta rule for method invocation via an association link differs in that ParameterEdge and EdgeEnd are replaced by AssociationRole and AssociationEndRole, respectively. Other additional requirements on attributes of the CallAction and the receiving ClassifierRole ensure that one deals with the simplest case and not with multiobjects, for instance. Another difference is that the method may not directly be called using the RoleName stored in the AssociationEndRole if sender and receiver are only indirectly linked. Hence we
include a non-terminal symbol $PATH_{s,r}$ which has to be replaced by an expression
determining the shortest existing link path from the sender $s$ to the receiver $r$.

In order to allow more than one method invocation in the body of an op-
eration, the rule scheme in the above meta rule generates a new non-terminal
symbol $MESSAGE\_INV_{o,m}$, distinguished by the differing parameter expression.
This second kind of non-terminals for method invocations can be replaced by
the second kind of meta rules.

Using the complete set of meta rules as shown for the generation of the code
for the class diagram by instantiating the meta rules and reducing the non-
terminals to terminal symbols, the following Java code is generated from the
collaboration diagram depicted in Fig.2:

```java
public class Company {
    public void processOrder (Order o) {
        Delivery d;
        Store s;
        int pNr;
        int a;
        pNr = o.getPNr();
        a = o.getAmount();
        s = search_stores(pNr, a);
        d = new Delivery(o,s);
        s.deliver(d);
        add_deliveries(d);  
    }
}
```

6 Conclusion and Perspectives

In this paper, we have investigated the modeling of behavior by UML collabor-
arion diagrams and their automatic transformation into Java code. We have
introduced methodical guidelines how to deploy collaboration diagrams in a
structured way. This formed the basis for the formulation of a transformation
algorithm.

The objective of this automatic transformation is to prevent a loss of sub-
stantial information during the transition from a model to its implementation.
But, this does not imply that UML collaboration diagrams offer a means to spec-
ify the behavior of a system completely and that UML can be used as a visual
programming language. UML collaboration diagrams focus on the modeling of
object interactions, while computations on data values are neglected, and thus
have to be added to the generated Java code by hand.

This paper focussed on the transformation of sequential behavior descrip-
tions. The next steps will be to implement the transformation algorithm by
extending the often used, commercial tool Rational Rose [10] and to extend the
transformation algorithm to the transformation of concurrent behavior as well
as of asynchronous and synchronous communication descriptions. The already
chosen target language Java will facilitate this development. First results of that
extension can be found in [6].
Finally, it is intended to investigate whether and how the in this paper reused approach of two-level grammars (cf. [17]) is an appropriate means to specify and realize easily adaptable code generators for forthcoming versions of UML and for a visual modelling language in general.

References