UML Collaboration Diagrams and Their Transformation to Java
—Extended Version*—

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

1 University of Paderborn, Dept. of Computer Science, D 33095 Paderborn, Germany
   engels@sauerjava@uni-paderborn.de
2 SAP AG, Lo. Dev. PP-PI, Neuottstr. 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 trans-
formed 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 pre-
requisite 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 au-
tomatically 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: UML, collaboration diagram, methodical guidelines, code
generation, Java, pattern-based transformation algorithm

1 Introduction

The Unified Modeling Language (UML [8, 9, 13])1 is an object-oriented modeling
language that provides a variety of diagram types for an integrated specification
of both the structure and the behavior of a system. UML consists of four catego-
ries of diagrams: use case diagrams, structural diagrams, behavioral diagrams,
and implementation diagrams. Collaboration diagrams—which are considered in
this report—belong to the behavioral diagrams like sequence diagrams, state-
charts 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

* Extended Version of [4]
1 This report refers to UML version 1.1 of September 1997 that has been adopted as
   a standard by the Object Management Group (OMG).
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 from a model to its implementation. Existing approaches in this direction transform statecharts into executable code [5, 1, 2]. Statecharts are used as object controllers for specifying when an object is willing to accept requests. CASE tools supporting code generation from statecharts are for example 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—via menus and text fields of Structure Builder's graphical user interface—to extend the interaction modeled in UML diagrams. Such information being necessary for the generation of working Java code is, for example, how objects can be accessed, how they are transported between methods, how links are instantiated, 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.

This report 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. This technical report is an extended version of [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]). We start by discussing the modeling objective of UML collaboration diagrams. As a preparation for the detailed description of the methodical approach, we then introduce a running example for a system and give an overview of the model elements of UML collaboration diagrams.

2.1 Modeling with UML Collaboration Diagrams

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 aspect (i.e., the sequence of message flow), but also the structural context of such an interaction, called a collaboration. The collaboration aspect consists of the structure of objects that collaborate to achieve the specified function and the relationships amongst them.

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

- Class: take the collaborations of all collaboration diagrams of operations of a class to form the context of the class implementation,
- Subsystem: map the specification part of a subsystem package onto its implementation part,
- 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.

In the above sense, the latter two kinds of usage are of interest. They differ not only in their level of abstraction, but also in their main intention. While use cases are deployed in earlier phases of modeling, the method-oriented usage is

---

7 The first kind can be obtained from the consequent and complete application of method specifications, but it is restricted to the structural aspect, abstracting from behavior. The second is just important if systems are structurally decomposed into subsystems during design, and specification and implementation models are distinguished.
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, i.e., the general functional behavior in a given context. As a consequence, collaboration diagrams are used on the instance level (i.e., one particular message flow of a system is shown) in the case of use case specification, describing the interaction of different objects with each other. In the case of method specification, they are used on the type level possibly containing iterations or conditional flows [9]. Type-level collaboration diagrams can have different instantiations (i.e., concrete message flow sequences conforming to such a specification), e.g., depending on the fulfillment of guard or iteration conditions. 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 systems 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, for instance, 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 operations 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. For example, exception handling 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.
2.2 Introducing an Example Application

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 object belongs to an Order.

![Class Diagram]

**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.

2.3 Model Elements of UML Collaboration Diagrams

UML contains a set of predefined elements suited to represent the entities of collaborations and interactions within collaboration diagrams. We will introduce these model elements and illustrate their use referring to the example of Fig. 2. We distinguish between objects, links, and message flow.
**Objects.** Objects are depicted as rectangles holding a name, a type, an optional list of state information, and an optional compartment for (typed) attributes. Four kinds of objects are distinguished: active and passive objects, multi-objects, and composite objects. Passive objects contain data, active objects, additionally own a thread of control. If the flow of control in an interaction is purely sequential, there are only passive objects within a collaboration diagram (like in Fig. 2). Multi-objects are used in a collaboration if an association end in the corresponding class diagram has multiplicity greater than one (see multi-objects for :Store and :Delivery). Composite objects can be visualized either as nested objects or by using the composition relation. An object can be characterized by a name and/or a type. UML distinguishes named objects not carrying type information, typed named objects, and anonymous objects just carrying a type. For each object in a collaboration diagram, the object type should be directly or indirectly inferable to allow for consistency checking with respect to the related class diagram. To mark objects (and links) being constructed or destructed during an interaction, predefined constraints exist to classify generated {new}, deleted {destroyed}, and objects (and links) that are both generated and deleted {transient} within the given interaction.

**Links.** Links connect objects within a collaboration. A link can either correspond to an instance of an association in the related class diagram (e.g. between :Company and the multi-object :Delivery), or it can be a stereotyped link. Links based on associations can carry role names (like store at the multi-object :Store in the example) and qualifiers at their link ends whose actual values identify the linked object. Links in collaboration diagrams are allowed to carry navigability, aggregation, and composition information, e.g. messages can only be sent in the direction of navigability.

If a collaboration diagram is deployed to specify the implementation of an operation, it may be necessary to access objects to which no association relationship exists. UML offers predefined stereotyped links for typical visibility relationships in the context of interactions such as «parameter» (to access o of type Order in the example of Fig. 2), «global» and «local» variables (e.g. s of type Store, d
of type Delivery), and a «self» link. Predefined constraints for construction and
destruction exist as for objects.

**Messages and Flow of Control.** Messages are sent across links in a col-
layeration diagram. To exchange a message, a sender object raises an action
which causes a request in a receiver object. The request triggers an event in
the receiver. In a collaboration diagram, a request is represented by a message.
These messages can be of two types: operation and signal. An operation is a
service that may be called by an object using a CallAction in order to achieve
a specified behavior in the receiver where it triggers a CallEvent. It can be ei-
ther synchronous or asynchronous. Its implementation is described by a method
which can be specified using an interaction diagram. A signal is an asyncronous
notification caused by a SendAction in the sender and triggering a SendEvent in
the receiver. The reaction in the receiving object is specified by a statechart and
therefore beyond the scope of collaboration diagrams. There is no syntactical
distinction between operations and signals.

A message consists of the message label and the control flow type. The mes-
gage label (e.g. 3: s := search(pNr, a)) consists of a predecessor list and a sequence
expression, both optional, and the signature. The signature contains the message
name (search) and optionally a return value (s) and an argument list (pNr, a).
In the case of a multi-object being the receiver, the message must be defined
for the container type, not the element type, as for the search method sent to
the Store multi-object in Fig. 2. The sequence expression specifies the order of
messages, nesting of calls, and the conditions of their sending as well as poten-
tial iterations. Sequence numbers are used for sequential arrangement (3: in the
example), letters for parallelism. The predecessor list specifies synchronization
constraints with other messages. Predecessors are only permissible if multiple
active objects participate in an interaction. Thus, the predecessor list is empty
in our example.

The control flow of a message is specified by the type of arrow pointing in the
direction of message flow. Synchronous, asynchronous, and undefined control flow
types are defined. In the first case, the next message on a specific level is sent only
after the direct predecessor has been fully processed, i.e., including processing all
its directly nested messages. In the asynchronous case, the following message on
this level can be immediately called after calling the predecessor message without
waiting for processing the predecessor itself. Therefore, processing has different
semantics for synchronous and asynchronous messages. These semantics apply
recursively. If a synchronous message contains no nested messages, it is regarded
as processed when all of its statements have been executed. In the given example
diagram, only synchronous messages are depicted by solid arrowheads.

### 2.4 Methodical Guidelines for Deploying UML Collaboration
Diagrams

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 (Company in Fig. 2); its signature must be declared in the operation compartment of the corresponding class in the class diagram.

We can distinguish between guidelines that are imposed by considering collaboration diagrams that specify the implementation of an operation and those that result from the constraints caused by the target language of the transformation approach.

**Specifying Methods.** 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 include those that are subordinate called within these nested operations. We consider this to be meaningful when we specify the implementation of an operation, since the subordinate operation calls are specified in individual collaboration diagrams for each nested operation. This is alike the definition of procedures and procedural calls in programming languages (respectively, methods and method calls in object-oriented 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.

In UML, an object within a collaboration can be further specified by adding state information textually or by attribute values to senders or receivers of messages as preconditions. In contrast, we propose to model these preconditions by
messages that include condition expressions on the appropriate attributes. This approach is more flexible because the conditions can be defined for each message separately. Therefore, objects in our collaboration diagrams are restricted to not carry explicit attribute values.

Consequently, collaboration diagrams with multiple occurrences of the same object name can be regarded semantically equivalent to a diagram with a unique occurrence of that object name. We thus assume that an object appears only once within a collaboration diagram. Regarding the naming of objects, we require that (anonymous) objects with role names on all links are deployed throughout the collaboration diagrams (compare Fig. 2) to avoid naming conflicts. Role names are used for accessing linked objects. Anonymous objects without a name of their own are to be used if multiple links carrying role names end at a given object. Role names on association links are especially needed to prevent ambiguities 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 multi-objects (notion 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).

**Generating Java Code.** 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 for example been refined 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 will be explained in the next subsection.
Based on this, Sect. 3.2 explains informally how collaboration diagrams are automatically transformed into Java code (fragments). We give two simple examples to sketch how this translation works.

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.

3.1 Transforming Class Diagrams

The transformation of class diagrams into corresponding Java code can reuse well-known techniques from the database design process where standard rules are known how an Entity-Relationship diagram can be translated into a corresponding relational database schema (see [12]).

The basic idea is to translate entity types or class definitions into corresponding relations or classes, and to translate associations into bi-directional references between the two participating relations or classes.

This standard transformation approach can be refined by accounting for cardinality constraints of associations as well as visibility constraints of attributes of classes and roles of associations. In addition, it has to be decided how language concepts from the class diagram level are handled on the code level if a direct translation is not possible due to lacking language features on the code level. An example is the multiple inheritance concept known in UML class diagrams, but unknown to Java.

In our approach, we deploy the following transformation approach for class diagrams:

1. UML class definitions are translated into Java class definitions.
2. By following the idea of strict encapsulation of all object attributes, attribute definitions are always translated into private attribute definitions in the corresponding Java class, independent of the defined visibility within the UML class definition. In case of public and protected attribute definitions in the UML class definition, additional public "put" and "get" operations are defined in a Java class definition, enabling write and read access on such an attribute. Furthermore, a generic search operation is generated for each class which enables to select a certain object from the set of all currently existing object instances of a class. The generic parameter describes the concrete selection criterion and has to be added by the developer to the automatically generated Java code fragment.
3. The translation of associations depends on the defined cardinality. In case of a to-1 association between class A and class B, a simple reference attribute of type B is added to the class definition A. In case of a to-* association, an attribute of type OrderedSet has to be defined to handle an arbitrary number of links between an object of class A and several objects of class B. In all cases, additional operations have to be added to all participating classes in
order to add, to delete, or to check the existence of a link between two objects, as well as to iterate on all existing links. In addition, the implementation of add and delete operations of links have to take care that no dangling references remain after changing the links and that the defined cardinality constraints are obeyed.

![UML Diagram]

**class Store**
- private int [] Products;
- private int [] Amounts;
- private Company StoreCompany;
- public Store () {
-   public void putProducts (int Index, int PNumber ) [...]);
-   public int getProducts ( int Index ) [...]);
-   private void putAmounts ( int Index, int Amount ) [...]);
-   private int getAmounts ( int Index ) [...]);
-   public void addStoreCompany ( Company c ) [...]);
-   public Company getStoreCompany () [...]);
-   public void delStoreCompany () [...]);
-   public boolean hasStoreCompany () [...]);
-   public Store search (<genericParameterList> ) [...]);
-   public void deliver (Delivery d) [...]}
- }

**Figure 3.** Translation of a class definition into Java code

Figure 3 gives an example how a UML class will be translated into corresponding Java code according to the above sketched algorithm. As it is not the topic of this report, the generated code in the body of automatically added operations has been omitted here. In this report, we focus on the generation of Java code for operations to which a corresponding collaboration diagram exists to define their semantics. This kind of code generation will be sketched informally in the next subsection and explained in detail in Sect. 5.

### 3.2 Transforming Collaboration Diagrams

We will now present by examples how collaboration diagrams are automatically transformed into Java code (fragments) based on the presented translation of class diagrams. First, Fig. 4(a) shows a part of the collaboration diagram given in Fig. 2 where operation `getpNr()` is sent via a parameter link with role `o` to an object of class `Order`. This collaboration diagram is depicted in the lower left part of Fig. 4(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. 4(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. It is assumed within this pattern that the object of class Company somehow obtains the value of the second parameter s of delivery(o,s) prior to this method’s invocation, but how the parameter is obtained is specified elsewhere.

These two examples give already a very good impression that, due to the well-formed structure of collaboration diagrams, a corresponding well-structured translation is supported. 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.

![Diagram showing transformation of parameter and local links into Java code](image)

**Figure 4.** Transformation of (a) parameter and (b) local links into Java code.
4 Refined Meta Model

The abstract syntax of UML is defined by a meta model in (a subset of) UML class diagram notation. To model a system, all concrete classes of the meta model can be instantiated.

Based on the UML meta model [9], we present a refined meta model (see Fig. 5 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 5 depicts the changes to the original UML meta model [9]. 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.

Objects used in a type-level collaboration diagram are actually instances of meta class ClassifierRole, links are instances of AssociationRole. A classifier role is based on the abstract meta model class Classifier that describes a model element owning static and dynamic features. Looking at a role, only selected features of the base class are considered that are important in the context of the specified interaction. Objects that play that role must conform to these features. Other features are not shown. In the same way, association roles offer a special, possibly restricted view on an association.

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 transformation can now detour the class interaction. The association with the role name first and the association predecessor on the class Message are used to navigate 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:
Figure 5. Extended meta model for transforming collaboration diagrams
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. Both classes possess an attribute to hold the affiliated, string-valued role name. 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.D.Kind has been added to the class AssociationEndRole. It has a supertype because it can also have the constraint {destroyed}. 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 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.

The class Feature can be deleted from the meta model because we do not consider explicit attributes on objects in collaborations.

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, DestroyAction and TerminateAction are obsolete as subclasses of Action. 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.

5.1 Pattern-based Transformation Algorithm

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 that are 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. 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.

5.2 Transformation Rules for Class Diagrams

Figure 6 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 that is actually translated by the meta rule is shown by concrete UML syntax as it appears on the diagram of the model. In the middle, we give its translation to part of an instance of the meta model. This constitutes abstract syntax of the corresponding model part. This instance 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. 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.

The first meta rule shown in Fig. 6 specifies the transformation of 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, i.e., the object identifier of the classifier in the pattern. 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 partly determines the occurrence of the pattern that belongs to the meta rule for replacing this non-terminal. The second meta rule shown in Fig. 6 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.

![Diagram](image_url)

**Figure 6.** Meta rules for class diagram

Let us again consider 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, i.e., on the abstract syntax level. 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
\begin{align*}
\text{START}_{\text{com}} & \rightarrow \text{public class Company} \{ \\
\text{ATTRIBUTES}_{\text{com}} & \\
\text{METHODS}_{\text{com}} & \\
\} \\
\end{align*}

Note that the instantiation process leads to a set of different start symbols—
i.e., more concrete non-terminal symbols, marked by the concrete subscript (e.g.,
the classifier \text{com} that appears as an object identifier in the pattern) replacing
parameter \text{c} of the non-terminal symbol \text{START}_{\text{c}} in the rule scheme—indicating
the particular classifier for which code is being generated. Thus, the generated
Java code can be stored in different files as required.

In the second step, we want to replace the non-terminal \text{METHODS}_{\text{com}}. This
could be done by using the second meta rule. But we need a particular instantiation
of the according rule scheme (METHODS_{\text{com}} instead of METHODS_{\text{c,1}}). Hence
the occurrence for the pattern has to obey this constraint. If an occurrence is
found that maps operation \text{o} to operation \text{procOrd} with name "processOrder"
and visibility "public", the rule scheme can be instantiated to:

\begin{align*}
\text{METHODS}_{\text{com}} & \rightarrow \text{public void processOrder } ( \text{ARGS}_{\text{procOrd}} ) \{ \\
\text{COLLAB}_{\text{procOrd}} & \\
\} \\
\text{METHODS}_{\text{com}} & \\
\end{align*}

With the above two rules, we can deduce a primitive class frame from the
start symbol \text{START}_{\text{com}}. Another rule is needed that terminates the recursive
production of method declarations by replacing the non-terminal \text{METHODS}_{\text{com}}
by the empty string in analogy to the bottom rule in Fig. 7.

\begin{align*}
\text{METHODS}_{\text{com}} & \rightarrow \emptyset \\
\end{align*}

This rule may only be applied if its alternative for replacing the non-terminal
is not applicable any more. (The rule scheme for this rule has been omitted in
Fig. 6.) Although the whole transformation of class diagrams into Java code
could be described analogously, we refer to the informal description in Sect. 3
and step to the transformation of the collaboration diagram now.

\subsection{5.3 Transformation Rules for Collaboration Diagrams}

The following presentation of transformation rules for collaboration diagrams
considers rules for declaring global and local variables as well as for method
invocations and structural dynamics.

\textbf{Global Variables.} We start transforming collaboration diagrams by transforming
their global variables. Since Java does not allow to define global variables,
we decided to define them as static variables in an additional class Globals. The
according meta rules are depicted in Fig. 7. The first meta rule contains an
instance of meta model class Model as its pattern and can hence only be ap-
plied once during the transformation of the diagrams of one model. The second
meta rule generates a declaration statement for each global variable occurring
on at least one of the collaboration diagrams of the model. The next meta rule
actually inserts the Java declaration of the global variable. The name of the
global variable is taken from the EdgeEnd of a GlobalEdge associated with the
global variable. The type is taken from the base Classifier of the target Classi-

fierRole of the EdgeEnd. Remember that all EdgeEnds of GlobalEdges of the
same GlobalVar have the same RoleName attribute value by construction of the
instance of the meta model. Note that the second and the third meta rule can
not be merged together, since we have to ensure that only one declaration for
each global variable is generated even if this variable occurs on more than one
collaboration diagram. Our mechanism of applying the meta rule only once for
each occurrence of the pattern ensures this property. The last meta rule of this
figure finally replaces the non-terminal GLOB_DECL by the empty string. It may
only be applied if its alternative for replacing the non-terminal is not applicable
any more.

![Diagram with collaboration diagram, pattern, and rule schema]

**Figure 7.** Meta rules for global variables
**Structure of Method Bodies.** In the next step, we apply meta rules for replacing the non-terminal \texttt{COLLAB}_0 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 that is indicated by the sequence numbers in the collaboration diagram. 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. 8.

![Diagram of the collaboration diagram, pattern, and rule schema for splitting of \texttt{COLLAB}](image)

**Figure 8.** Meta-Rule for splitting of \texttt{COLLAB}

In the sequel, the first two meta rules generated by this substitution are explained in detail.

![Diagram of the collaboration diagram, pattern, and rule schema for local variable declaration](image)

**Figure 9.** Meta rule for local variable declaration

**Declaration of Local Variables.** Figure 9 shows the meta rule for declaring local variables. Remember that we also assume that indirectly declared local variables (return values of method invocations that yield references to 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 attribute 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 ClassifierRole of the EdgeEnd. This information is represented in the pattern. Moreover, it is used in the rule scheme by the parameters \( c\text{.name} \) for the type and \( e\text{.RoleName} \) 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_DECL\(_o\). 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. 9 serves for the end of the declaration process. The rule scheme replaces the non-terminal LOCALVAR_DECL\(_o\) by the empty string. It may only be applied if the upper meta rule is not applicable any more.

**Method Calls.** 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. Thus, 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. Consequently, the application of the rule shown in Fig. 10 must be restricted to cases where message \( m \) does not have a successor message.

**Figure 10.** Meta rule for completing method invocations

For the first two kinds of meta rules, we additionally have to distinguish many different cases:

- whether the receiver of the message is a multi-object.
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.

We do not present all according meta rules in this report, but restrict the presentation to a fundamental selection from which the missing meta rules are constructed as slight modifications.

![Diagram](image)

**Figure 11.** Meta rule for method invocation on parameter object

Figure 11 shows 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 same holds for the non-terminal symbol ASSIGN_r. It is deployed to evaluate whether the called operation has a return value and to which variable this return value is assigned.

The meta rules for invoking an operation on a local or global variable look quite similar. Only the ParameterEdge in the pattern is replaced by a LocalEdge or GlobalEdge, respectively. The transformation of a <self> link is handled analogously, distinguishing between using a this-pointer or a super-pointer to call a redifined method of a superclass.
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 regarding attributes values of the CallAction and the receiving ClassifierRole ensure, for instance, that one deals with the simplest case and not with multi-objects. 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 PATH₀,t which is replaced by an expression determining the shortest existing link path from the sender, i.e., the target object of the specified operation o, to the receiver that is referenced by AssociationEndRole t. The according rule is shown in Fig. 12.

Figure 12. Meta rule for method invocation via association link

Deriving from this meta rule the meta rule for the method invocation if the receiver is a multi-object is rather simple. The pattern stays the same with the only exception that the attribute multiplicity of the ClassifierRole must have a value greater than "1". In the rule scheme for multi-objects (not depicted), we add the name of the AssociationEndRole t (compare Fig. 12), preceded by an underscore, i.e., \_t.name after \_r.name in order to extend the name of the invoked method by the RoleName used in the collaboration diagram. Thus, we ensure uniqueness of the operation in case of more than one association connecting to a multi-object.

If the method invocation is to be sent via a newly created link, the rule of Fig. 13 is applied. The application of this rule is constrained by the value "new" of the attribute \_t.type of AssociationEndRole t. In this case, the new link must
Figure 13. Meta rule for method invocation via a \{new\} association link

Figure 14. Meta rule for conditional method invocation via association link
be set for the counterpart link end (i.e., AssociationEndRole f) by calling the
 corresponding put method put_t.name prior to invoking the operation r.Name.
The non-terminal symbol PATH_o,f yields a link path from the target object of
operation o to the object attached to the counterpart link end. The put method,
in turn, is parameterized by a link path PATH_o,t referring to the receiver object
of the called method (op1).

Message calls that are restricted by a condition expression require to be trans-
formed into some conditional clause in the target language. Fig. 14 shows the
rule for a conditional method invocation via an existing association link. Mandatory
for this rule to be applicable is that the attribute isCond of the CallAction
node "true". The recurrence expression of the condition is evaluated by the
non-terminal symbol CONT. Loop conditions are handled analogously.

![Collaboration Diagram](image)

**Figure 15.** Meta rule for traversing message sequences based on method invocation
on existing association link

In order to allow more than one method to be invoked in the body of an op-
eration, the rule schemata in the above meta rules generate a new non-terminal
symbol MESSAGE_INVOC_0,m. Note that it is not the original non-terminal sym-
bol for method invocations since the parameter expression (encoded in the sub-
script of the non-terminal symbol) has been changed. This second kind of non-
terminals for method invocations can be replaced by the second kind of meta rules, i.e., those for traversing predecessor edges. Figure 15 shows an example of such a meta rule for invoking a message on an object referred to via an existing association. The important application condition for this rule is, in contrast to the rule depicted in Fig. 13, that attribute Type of the target AssociationEndRole (referenced by role name to) has a value other than "new". The pattern contains messages m1 and m2. The code for message m1 has already been generated as indicated by the parameter expression o,m1 of the non-terminal to be replaced.

![Diagram](image)

**Figure 16.** Meta rule for constructor invocation

The last meta rule that we explicitly consider is the one for invoking a constructor via an association. In this case, not only a new object is created, but also the new link is set (compare Fig. 16). The pattern differs from the simpler case depicted in Fig. 15 in three places. First, the base Classifier d of the receiver of the message is needed, since it provides the name of the constructor d.name. Second, we have a CreateAction instead of a CallAction. Third, the source AssociationEndRole t is additionally needed in order to be able to link
the new association to the new object. On the side of the rule scheme, we first call the constructor and then insert the new association.

The meta rule for replacing the non-terminal ASSOC\_DYNAMICS\_\_\_, which also inserts the code for creating a new object, is a reduction of the meta rule shown in Fig. 16.

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, in the context of the class diagram in Fig. 1.

```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_store(pNr, a);
        d = new Delivery(o,s);
        s.deliver(d);
        add_delivery(d);
    }
}
```

6 Conclusion and Perspectives

In this paper, we have investigated the modeling of behavior by UML collaboration 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 substantial information during the transition from a model to its implementation. But, this does not imply that UML collaboration diagrams offer a means to specify 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 descriptions. 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 re-used 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