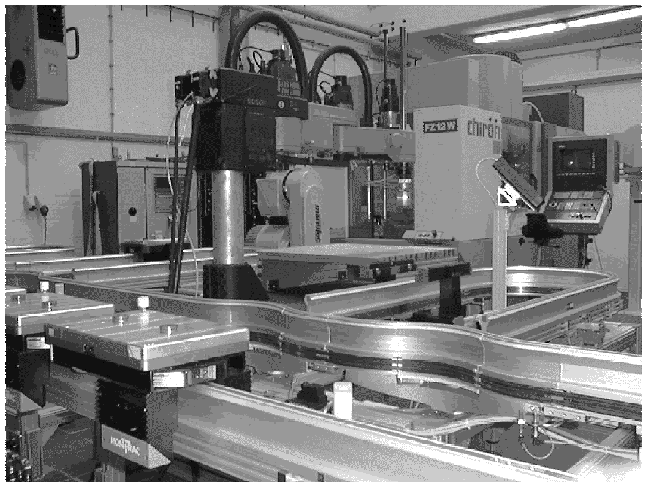
From Uml to Java And Back Again

Thomas Klein, Ulrich Nickel, Jörg Niere, Albert Zündorf

AG-Softwaretechnik
University of Paderborn, Germany
[buko|duke|nierej|zuendorf]@uni-paderborn.de
D-33095 Paderborn



Abstract. FUJABA is a public domain research prototype CASE tool that aims to support round-trip engineering for UML class diagrams as well as for UML behaviour diagrams. Like other CASE tools, Fujaba generates Java class definitions from UML class diagrams. In addition, we combine the UML state-chart, activity diagram, and collaboration diagram notations to a powerfull visual programming language. This enables the generation of method bodies from their visual specification. To be of general use, Fujaba provides round-trip engineering support. This means, the user is allowed to modify the generated code manually and Fujaba is able to load modified code and to (re)establish the corresponding (modified) UML diagrams. This covers class diagrams and (to some extend) behaviour diagrams. In contrast to most existing CASE tools, Fujaba does not use structured comments to seperate generated code and manual modifications but Fujaba relies on common naming conventions and some programming styles. Thus, Fujaba is able to recognize all code that sticks to these programming styles.

1 Introduction

Normally UML is used in the early software development phases. Use-case diagrams serve for requirements analysis. During object-oriented analysis and design, the different use-cases are refined by a number of scenarios using sequence diagrams, collaboration diagrams or activity diagrams. In more elaborated cases, state-charts may be used to specify exact (object) behaviour. Together with these scenarios one developes class diagrams specifying the static aspects of the desired application like classes, attributes, associations, and method declarations.

State-of-the-art CASE tools like Rational Rose [5], TogetherJ [6], and Rhapsody [7], provide editors for various kinds of UML diagrams. However, since most UML behaviour diagrams describe only scenarios, code generation and round-trip engineering support is restricted to class diagrams and (in case of Rhapsody) state-charts. In [2], [8], [9], [10], we proposed to use UML behaviour diagrams for the specification of method bodies and for code generation. In this work, we use state-charts and activity diagrams for the specification of higher-level control flow of classes and methods, respectively.

Normally, the basic actions of state-charts and activity diagrams are described by pseudo code, only. In our work, we employ collaboration diagrams to specify complex object structure look-ups and modifications. Complex computations and access to system calls may be programmed in standard Java code. Thereby, we provide a precise, formal, operational semantics for collaboration with state-charts and activity diagrams. Note, we use UML behaviour diagrams for the specification of class reaction on sent signals and for the specification of method bodies. This is quite different from scenario description and requires a different interpretation and usage of the diagrams and elements.

Altogether, our work allows to use UML class and behaviour diagrams as a very high-level visual programming language called story-diagrams. This paper focuses on round-trip engineering support for this visual programming language by the FUJABA environment. The concepts for code generation have already been described in [2], [10]. This paper adds the concepts for recognizing class and behaviour diagrams from Java code.

Chapter 2 introduces the language, story-diagrams, on hand of a running example. Chapter 3 outlines our code generation concepts as far as necessary for the understanding of our reverse engineering approach. This reverse engineering approach is discussed in chapter 4. Chapter 5 summarizes our work and outlines some future work.

2 Story-Diagrams, the language

In this chapter, we introduce the story-diagram language on hand of a running example, which deals with the application domain of production control systems. Nowadays, the market demands a very flexible production process. To meet these changes, production control systems become more and more decentralized. Moreover, frequent changes of the production process result in permanent adaptions of the control software. To

perform these adaptions more rapidly, one aspires to specify the control software on a high abstraction level and to generate the appropriate code. In turn, the existing code has to be reengineered to modify the specification according to the new production requirements.The building blocks of such a production control system are different, selfacting and computer controlled resources like e.g. switches, shuttles, machines or robots. The picture on the frontpage shows such a production control system used by our mechanical engineer faculty. Shuttles move on rails and transport goods between various production places. Each production place can be reached using switches in the rail-way system. Such switches route shuttles to certain production places or to pass it, if nothing has to be done with the good at the production place.

Figure 1 shows the structure of a switch as part of a material flow system, which we specify by employing FUJABA, currently. The switch has a switch drive, which changes its direction, some sensors, which observe the environment and a LON[1]-node, which is connected to a communication network via a bus interface, where the application software is nested. In our example, the identification unit detects an arriving shuttle and reports the shuttle's id to the control node. Now, the control software decides in which direction the shuttle should be send. If the switch has to change its direction, it activates the stopper in order to let shuttles wait. One has to asure, that no shuttle is in the switching area, when the switch drive is activated, because otherwise the switch drive could be damaged. For that reason, the switch has a pass observer at each exit, which reports every shuttle leaving the area. Note, that we have a determined driving direction, so that we have one entry and two exits (which means that its a 'branching switch').

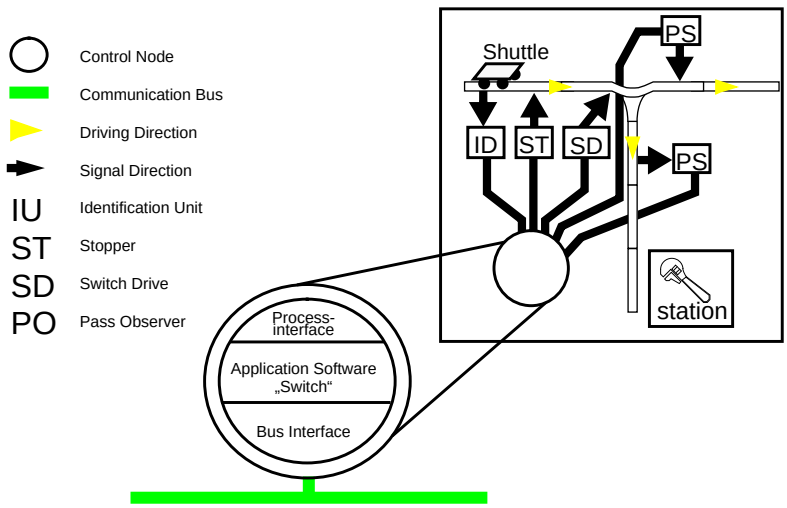


Figure 1 The structure of a switch in the material flow system

1. Local Operating Network

In the first step, we identify and model the static parts of the switch control software. FUJABA provides an editor for UML class diagrams. Figure 2 shows a screen shot of the FUJABA environment with a UML class diagramm in the main part of the window, corresponding to Figure 1. Note, that the communication between the nodes on the one hand, and the node and its peripheral equipment on the other hand is signal driven. Thus, we plan to add an optional „signal-compartment“ to the classes of the diagram, which means, that we have asynchronous communication, here. The implementation is scheduled for summer.

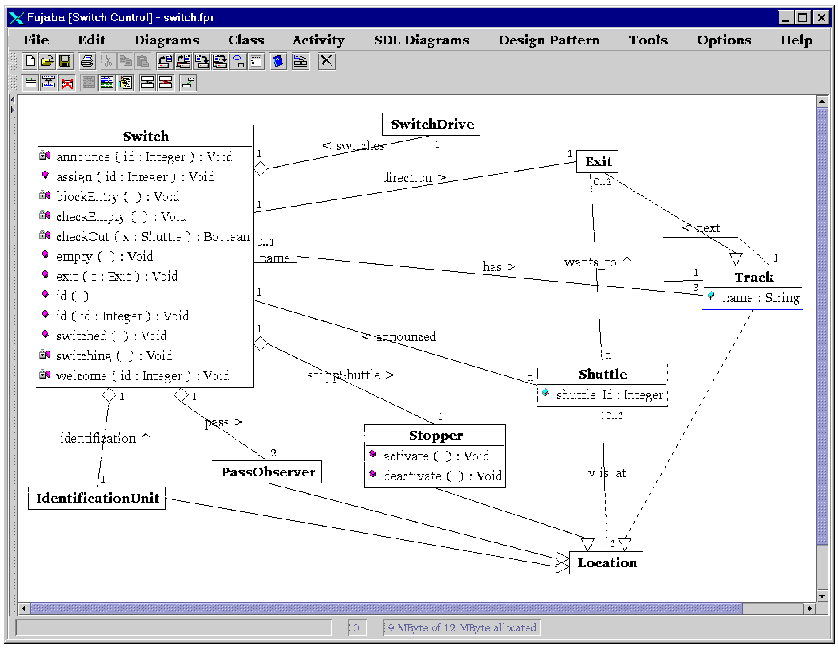


Figure 2 FUJABA class diagram of a switch

To specify the dynamic aspects of the switch, we employ so-called story-diagrams. Figure 3 shows the specification of the method `Switch::welcome`[1]. The control flow of the method is modeled by using an UML activity diagram. To specify the manipulation of the object structure, we enriched the activity diagram by two graph rewriting rules.

Basically, a graph grammar rule allows the specification of changes to complex-object-structures by a pair of before and after snapshots. The before snapshot specifies which part of the object-structure should be changed and the after snapshot specifies how it should look like afterwards, without caring how this changes are achieved.

In order to facilitate the use of graph rewriting rules for object-oriented designers and programmers, we adopt UML collaboration diagrams as a notation for object-structure

1. The welcome item in the navigation tree on the left hand side is highlighted.

rewriting rules. In UML, collaboration diagrams do not have a precise execution semantics, but model only possible scenarios. Using graph grammar theory we are able to define an execution semantics for collaboration diagrams, easily, thus enabling their translation to an object-oriented programming language.

Originally, collaboration diagrams are intended to model scenarios of complex message flows between a group of collaborating objects.[1] In addition, collaboration diagrams allow to depict the effects of operations in terms of changed attribute values and created and destroyed objects and links. Thus, the intial situation modeled by a collaboration diagram corresponds to the left-hand side of a graph grammar rule. Accordingly, the situation resulting from the execution of the collaboration diagram corresponds to the right-hand side of that graph grammar rule. This view allows the execution and translation of collaboration diagrams using techniques known from the graph grammar field, cf. [2],[4], [15].

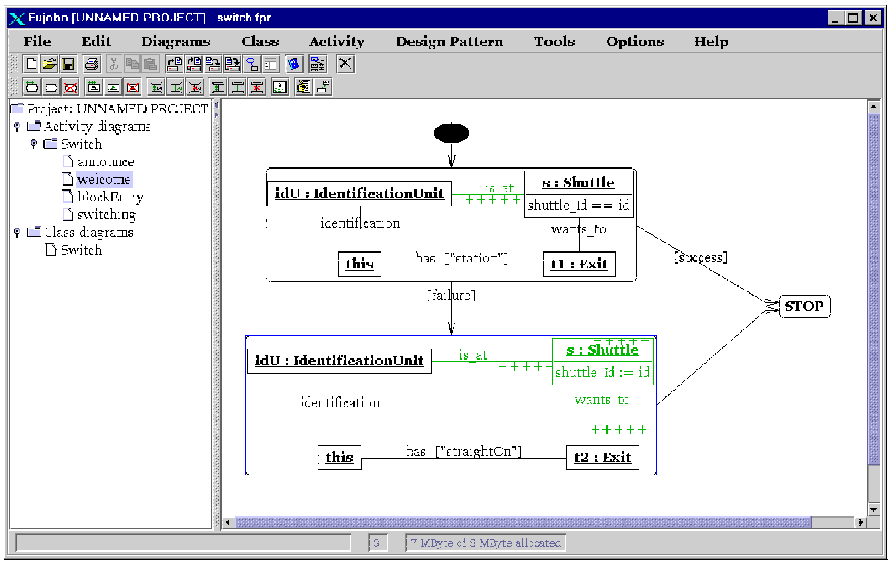


Figure 3 Story-diagram of the method Switch::welcome

In our example, the first activity is specified via an object-structure rewriting rule that shows four objects: *this*, *idU*, *t1*, and *s*. The *this* object is attached to the *IdentificationUnit* object *idU* and the *Exit* object *t1* via an *identification* link and a *has* link. Note, that the class *Exit* is is a subclass of *Track*, which in turn has a qualified association to the *switch* class (cf. Figure 2). We interprete *this*, *idU*, *t1*, and *s* as variables and the shown links as logical constraints on the allowed values of these variables. Based on this interpretation, such a diagram is executed by binding the specified variables to concrete object instances such that all specified constraints are fulfilled.

1. This use of collaboration diagrams is equivalent to UML sequence diagrams, cf. [14].

In our example the variable *idU* is bound to the identification unit which is linked to the current switch object (bound to the variable this). The same holds for the variable t1 which is bound the the (qualified) *Exit* object which is linked to the switch object via the "station-link". If the identified shuttle formerly was announced to the switch, the switch routes the shuttle to the *station* exit. In that case, an instance of the identified shuttle already exists and is linked to the station's exit by a *wants_to* link. Now, we just have to model, that the current position of the shuttle is at the identification unit. This is specified by creating a *is_at* link between the shuttle and the identification unit of the switch. If the identified shuttle wasn't announced before, there is no corresponding instance and the variable *s* cannot be bound. The graph rewriting rule fails. In that case, the control flow follows the failure transition. The second graph rewriting rule is interpreted in the same manner. It models the creation of a new shuttle object, which is linked to the identification unit and the *straightOn* exit of the switch, which means that it is not routed to the station. In the next chapter it is described how code can be generated for class-, and stroy-diagrams.

3 Source Code Generation for Story-Diagrams

The code generation is devided into two steps. In the first step Java code for class diagrams will be generated. The method bodies, specified with story-diagrams, are generated in a second generation step.

For each class in the diagram, a corresponding Java class is generated including the inheritance specified in the diagram. The class' attributs are mapped to Java attributes. According to software development standards, we use private Java attributes accessible via appropriate public get- and set-methods. In addition, these access methods ease adaptability and side effect handling e.g. change notifications. The class' method declarations are mapped directly into Java code.

The last and complex part of the code generation out of class diagrams is the mapping of associations. Generally there are different ways how to implement associations in Java. Fujaba provides private attributes and access methods for an association in the attached classes. The benefit of these standarized methods is that the write access methods ensure that there are no dangling references and that the links in the object structure are consistent. At this, consistent means that a link in the object structure is bidirectional and so the attributes in the corresponding objects have to refer the other object. To differ between associations, Fujaba discriminates between two kinds of associations or better cardinalities of the corresponding roles. For to-one associations a single valued attribute of the partner class' type and the appropriate get and set methods are generated. For to-many associations, a container attribute and further access methods are generated. These access methods provide the read and write access of the container as well as the possibility of iterating through the container, checking the containment of an object, getting the number of contained objects, and so on. The methods for adding objects to and removing objects from the container asure the consistency of the object structure as well as the corresponding methods for to-one associations. For further details see [2], [10].

The Java code generation for story-diagrams is divided into two tasks. First, the control flow is mapped to imperativ control structures like if, and while statements. To enable this translation, story-diagrams are restricted to so-called well-formed transition structures that correspond directly to nested branches and loops. Figure 4 shows the Java implementation of the welcome method of class switch. The UML method declaration is placed as comment above the Java method declaration (line 1 to 3). The success/failure decision after the first graph rewriting rule corresponds to the if-then/else construct (line 7 to 12).

```
 1:  /**
 2:    * UMLMethod: '- * welcome (id : Integer) : Void'
 3:    */
 4:    private void welcome (int id)
 5:    {
 6:       // first graph rewriting rule
 7:       if (sdmSuccess)
 8:       { }  // then
 9:       else
10:       {
11:          // second graph rewriting rule
12:       }  // else
13:       return;
14:    }  // welcome
```

Figure 4 Control flow of method welcome

```
 1:  try
 2:    {
 3:    // bind t2 : Exit
 4:    sdmtmpObject = this.getFromHas("straightOn");
 5:    JavaSDM.ensure (sdmtmpObject != null
 6:                    && sdmtmpObject instanceof Exit);
 7:    t2 = (Exit) sdmtmpObject;
 8:
 9:    // bind idU : IdentificationUnit
10:    idU = this.getRevIdentification();
11:    JavaSDM.ensure (idU != null);
12:    // create object
13:    s = new Shuttle();
14:    // assign statement
15:    s.setShuttle_Id (id);
16:    // create link
17:    idU.setRevIs_at (s);
18:    // create link
19:    t2.addToRevWants_to (s);
20:    sdmSuccess = true;
21:    } catch (JavaSDMException sdmInternalExcep-
tion)
22:    {
23:       sdmSuccess = false;
24:    }  // try catch
```

Figure 5 Java code for second graph rewrite rule

In a second task, the code for activities is generated. Story diagrams support two kinds of activities, statement activities and graph rewriting rules. Statement activities just contain Java code and are copied one-to-one. For graph rewriting rules we employ translation mechnisms as described in [2], [10]. Figure 5 shows the generated code for the second graph rewriting rule. The execution starts with binding objects to the variables specified in the rule. For example in line 10 the variable *idU* is bound to an object which is accessable via the *identification* association among the switch and the identification unit object. Line 11 checks whether an object could be bound or not and throws an exception in case of a failure. This exception is caught within the catch-statement (line 21) and it is

signaled that the execution fails (line 23). If all unbound variables are bound to objects, the specified modifications are executed. For example, a new object is created in line 13 and in line 17 and 19 two new links are created. Then, line 20 signals that the graph rewriting rule has been executed successfully.

4 Class- and Story-Diagram reconstruction

According to the generation of Java code out of specifications, the reverse step is also devided into two tasks. First, the static information, the class diagrams, will be reconstructed and in a second task, the story-diagrams will be recognized.

Figure 6 shows a cut-out of the generated code of class `Switch`, described in the previous chapter. Comments and most of the method(bodie)s are skipped. Only the specified welcome method (line 3) and the implementation of the announced association is listed (line 6 to 12). Beside this, Figure 7 shows the generated Java code for class `Shuttle`[1]. In line 17 to 19 the generated code for attribute `shuttle_Id` is presented. Lines 21 to 23 show the code generated for the association *announced* in class `Shuttle`. The reason why the association is generated in two different ways in the two corresponding classes is that the announce association is a one-to-many association. A detailed description and motivation of the generated code can be found in [1] and [2].

```
 1: public class Switch extends TrackElement {
 2: ...
 3:    private void welcome (int id) {...}
 4:
 5: ...
 6:    private OrderedSet revAnnounced = new OrderedSet ();
 7:    public boolean hasInRevAnnounced (Shuttle elem) {...}
 8:    public Enumeration elementsOfRevAnnounced () {...}
 9:    public void addToRevAnnounced (Shuttle elem) {...}
10:    public void removeFromRevAnnounced (Shuttle elem){...}
11:    public int sizeOfRevAnnounced() {...}
12:    public void removeAllFromRevAnnounced() {...}
13: ...
14: } // class Switch
```

Figure 6 Java code for class Switch

To reconstruct the class diagram out of this two pure Java code fragments, first, FUJABA uses a parser to construct a syntax graph for the source code. The parser is generated with JavaCC [11]. JavaCC generates a front-end of a parser for a given grammar. We added a back-end, so that the parser is able to construct a

```
15: public class Shuttle
16: ...
17:    private int shuttle_Id;
18:    public int getShuttle_Id () {...}
19:    public void setShuttle_Id (int shuttle_Id) {...}
20: ...
21:    private Switch announced;
22:    public Switch getAnnounced () {...}
23:    public void setAnnounced (Switch announced) {...}
24: ...
25: } // class Shuttle
```

Figure 7 Java code for class Shuttle

1. Like before, only the necessary parts for the recognition are shown.

rudimentary class diagram out of the parsed information (cf. [16]). Such a rudimentary class diagram consists of classes with (*private*) attributes and methods, either generated access methods and 'real' methods. Also the inheritance relations (line 1) are recognized directly out of this first step. Figure 8 shows the rudimentary class diagram of the Java source code for class `Switch` and class `Shuttle`.

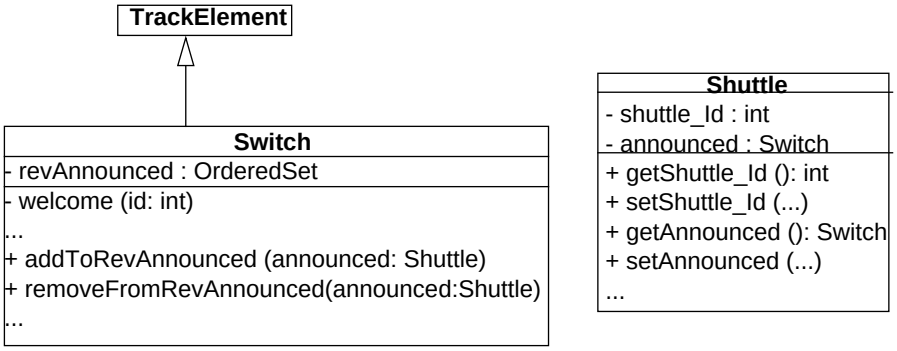


Figure 8 Rudimentary class diagram

In a second step, the access methods must be filtered out of the classes and associations have to be (re)constructed. Therefore, FUJABA contains an incremental, generic annotation process.

Each element in the syntax graph is passed to a set of annotation engines and can be annotated by them. Such an annotation is again an element in the syntax graph and so, other annotation engines can annotate this annotation. An example of the annotation structure for the attribute `shuttle_Id` of class `Shuttle` is shown in Figure 9. In the

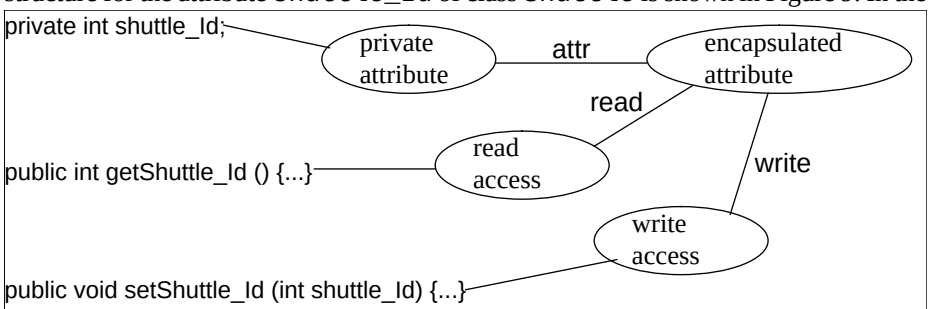


Figure 9 Annotation structure for encapsulated attributes

first level the parsed declarations (elements of the syntax graph) are annotated[1]. There are, for example, the attribute itself, annotated with a *private attribute* annotation and the access methods, classified in read and write access. The annotation process uses the naming conventions mentioned in the previous chapter to recognize such methods. In case of association access methods, there are other annotations and thereby, classifications needed. Constructive on these three annotations, an engine recognizes

1. The real object structure is more complex, but this simplification sufficies for the understanding of the concepts.

that this is not a standalone attribute and methods, but an encapsulated attribute. So the engine constructs another annotation called *encapsulated attribute* and connects this second-level annotation with the three first-level annotations. To provide a quick access for the connected annotations and diagram elements, the connectors may be guarded with names e.g *attr*, *read*, *write*. Now, after the second-level annotation is constructed, and thereby, the attribute and the methods have been classified as an encapsulated attribute, the annotation engine marks the methods as hidden and sets the visibility of the attribute to *public*. This can be done, because FUJABA has recognized that the access methods and the visibility have been generated by itself. Note, that the methods are only marked as hidden and not be delete in order to rescue changes a developer has made in the generated source code before he/she starts the parsing process.

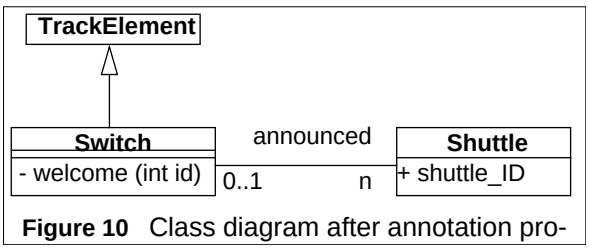


Figure 10 Class diagram after annotation pro-

In case of attributes and methods, which have been generated for associations, the corresponding annotation structure is more complex, but looks like the above. Thereby, to recognize, that a method is a e.g. write access method, the annotation engine has to look into the method's body in order to get the partner class on the other side of the association. Using naming conventions is not enough in the case of a to-many association, because to find the access methods of a attribute of type `OrderedSet`, the method bodies have to be examined in order to check, if ,the object passed as parameter, is really added to the container. We use traditional compiler techniques to extract the informations, cf. [16]. Once an annotation engine recognizes that an attribute e.g. `revAnnounced` and some access methods e.g. `addToRevAnnounced` or `removeFromRevAnnounced` correspond to a *reference* association between two classes (`Switch` and `Shuttle`), either the attribute and the access methods will be hidden and a reference association will be added from class `Switch` to class `Shuttle`. If the same engine recognizes that there is another reference assoziation between the two classes, but directed the other way around, the engine will not construct another reference, but a normal (two side navigational) association. To recognize that two reference associations represent a normal associations, we examine the bodies of the corresponding write access methods in order to look for a write access method calls in the partner class. Figure 10 shows the class diagram after the annotation process has been finished. The access visibility of the attribute `shuttle_Id` of class `Shuttle` has been set to *public* and the access methods either of the attribute and of the association are hidden as well as the attributes for the association.

The described annotation process also works for e.g aggregation, composition, and qualified associations, so that class diagrams can be recognized from Java code if the code is generated from FUJABA itself, or a developer uses the naming conventions and implementation concepts of FUJABA. It is also possible to use the annotation concept to recognize, create, and complete design patterns [3] within a class diagram. This is also part of the environment. The support of a more flexible recognition system, in order

to reengineer ’legacy’ code, is current and future work. But we assume that it can be done easily if the reengineer is able to interact with the system and is able to configure it.

After the class diagram has been recognized, the method bodies must be examined in order to reconstruct the story-diagrams. As mentioned in the previous chapter, story-diagrams consist of activity diagrams to specify the control flow and Java code or graph rewriting rules within activities. Thereby graph rewriting rules are a kind of collaboration diagrams.

To reconstruct story-diagrams out of Java code, the first step is to reconstruct the control flow (activity diagram). Such a control flow can be constructed directly out of the syntax graph and is like a rudimentary class diagram (see above). Each activity contains exactly one line of Java code and branches and loops are displayed as transitions with guards. Figure 11 shows such a rudimentary activity diagram of the `welcome` method of class `Switch.` The loop transition at the first activity derives from the generated code for the graph rewriting rule and will be replaced in the annotation process. Each activity contains a sequence of statements and the control flow is specified with guarded transitions (the guards are just abbreviations due to the lack of space).

Like the recognition of class diagrams such rudimentary activity diagram will be annotated in order to reconstruct the graph rewriting rules (collaboration diagrams). If no graph rewriting rule can be recognized in the whole or in parts of the activity diagram, it is left untouched. This might be the case if the method does not contain a rewrite rule or a developer has made changes in the source code in such a way that

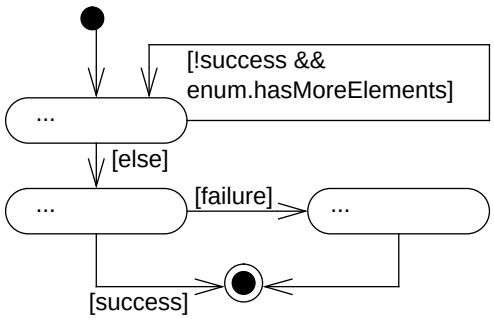


Figure 11 Activity diagram for method welco-

the rewrite rule can't be recognized. The annotation process constructs a multi-level annotation structure like in Figure 9 for attributes.

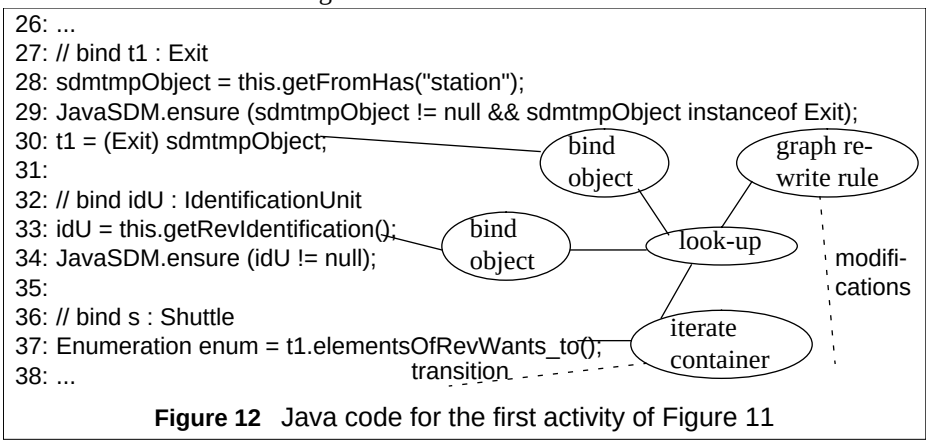


Figure 12 Java code for the first activity of Figure 11

Figure 12 shows the annotation structure and the annotated source code for the first activity of Figure 11. The first level annotations *bind object* just connect the source code (line 30, 34). To annotate such a line, the annotation engine uses the information of the class diagram and the annotation process, that the methods `getFromHas` and `getRevIdentification` are read access methods for associations. Thereby, the return type of the access method and the association itself is known. The *iterate container* annotation connects either line 37 as the self loop transition in Figure 11 displayed as the dashed line guarded with *transition*. The *look up* annotation connects all annotations on the first level, which signal that they bind objects in the object structure to variables. They deal as something like a group annotation. The top-level annotation is the *graph rewrite rule* annotation, which signals that all containing annotations refer to a graph rewriting rule and replaces all activities and transitions refering to that graph rewrite rule in the activity diagram by one activity containing the corresponding rule. The resulting activity diagram looks the same like in Figure 3. Note, that the loop in Figure 11 is not pertinent any longer, because it has become part of the graph rewriting rule annotated by the *iterate container* annotation.

Within these concepts, FUJABA is able to provide round-trip enginering and, as mentioned above, the support of recognition, creation and completion of design patterns [3]. The round-trip engineering works also if a developer makes manual changes in the source code and uses the naming conventions and implementation concepts of FUJABA. To provide a more flexible recognition, we will use generic fuzzy reasoning nets (GFRN) [12] and we assume that we are able to reengineer 'legacy' Java code then. For example, the SWING library [13] also contains a kind of graph rewriting rules. Therefore GFRN's provide a percentual uncertainty and the reengineer can decide if a part of a source code corresponds to a graph-rewriting rule or this transformation can be done automatically.

The recognition of state-charts has not been mentioned here, because it works like the described process, as well. Since, we use state-tables to implement state-charts, it is

only necessary to analyze the setup method of the state-table to recognize the information.

5 Conclusions and Future Work

Most current UML tools provide round trip engineering support for class diagrams only. Our work allows to use UML behaviour diagrams like state-charts, activity diagrams, collaboration diagrams as a visual programming language with well defined semantics. The FUJABA environment provides editors for various diagrams, and a code generator for Java code. This paper describes the parsing and recognition concepts of the FUJABA environment, especially for class-, activity-, and collaboration diagrams (graph rewriting rules). We use Java beans naming conventions to identify field access operations. (To-many) associations are identified having a closer look at such access operations. State-charts are reconstructed from their state table setup method. Activity diagrams are recognized by looking at control flow statements.

Groups of object structure look-ups and modification statements will be turned into collaboration diagrams. Diagrams derived from code are carefully merged with existing design diagrams in order not to loose existing layout information. Altogether, this will result in a sound round-trip engineering support by the FUJABA environment. Currently class diagram recognition is done. State-chart, activity diagrams, and collaboration diagram recognition is scheduled for August 1999, first results are encouraging.

Future work will try to analyze legacy code and provide reengineering support for it.

6 References

[1] T. Fischer, J. Niere, L. Torunski. *Design and Implementation of an integrated Development Environment for UML, Java, and Story Driven Modeling*, Master Thesises, Paderborn 1998 (in German)

[2] T. Fischer, J. Niere, L. Torunski, A. Zündorf. *Story Diagrams: A New Graph Grammer Language based on the Unified Modelling Language and Java*. to appear in Proceedings of TAGT '98 (Theory and Application of Graph Transformations), LNCS, Springer 1999

[3] E.Gamma, R.Helm, R. Johnson, J.Vlissides. *Design Patterns*. Addison-Weslay, 1995.

[4] G.Rozenberg (ed). *Handbook of Graph Grammars and Computing by Graph Transformation*. World Science, 1997.

[5] *The Rational Rose case tool*, Rational, http://www.rational.com

[6] *The TogetherJ case tool*, Object International, http://www.topethersoft.com/press

[7] *Rhapsody case tool*, ILogix, http://www.ilogix.com

[8] J.H. Jahnke, A.Zündorf, *Specification and Implementation of a Distributed Planning and Information System for Courses based on Story Driven Modelling*, in Proc. of 9th International Workshop on Software Specification and Design, Ise-Shima, Japan, IEEE Computer Society, pp. 77-86, ISBN 0-8186-8439-9

[9] U. Nickel, J. Niere, W. Schäfer, A. Zündorf, *Combining Statecharts and Collaboration Diagrams for the Development of Production Control Systems*, to appear in Proc. of Object-oriented modelling of embedded real-time systems (OMER) workshop, May 1999

[10] H.J. Köhler, U. Nickel, J. Niere, A. Zündorf, *Using UML as Visual Programming Language*, to appear as technical report, University of Paderborn, 1999

[11] The SUN Java Compiler Compiler (JavaCC), http://www.suntest.com/JavaCC

[12] J.H. Jahnke, W. Schäfer, A.Zündorf, *Generic Fuzzy Reasoning Nets as a basis for Reverse Engineering Relational Database Applications*, in Proc. of European Software Engineering Conference (ESEC/FSE), LNCS 1302, Springer, 1997

[13] *The SWING library, Java Foundation Classes*, http://www.sun.com/products/swingdoc-current

[14] G. Booch, J. Rambough, I. Jacobson, *The Unified Modelling Language User Guide*, Addison-Wesley, 1999, ISBN 0-201-57168-4

[15] A. Schürr, A. Winter, A. Zündorf, *Graph Grammar Engineering with PROGRESS*, in W. Schäfer (ed.) Software Engineering ESEC '95, LNCS 989, pp. 219 - 234, Springer 1995

[16] A.V. Aho, J.D. Ullmann, *Principles of Compiler Design* (The Dragon Book), Reading, Addison-Wesley, 1986