Roundtrip Engineering with FUJABA[1]

(Extended Abstract)

Ulrich A. Nickel, Jörg Niere, Jörg P. Wadsack, Albert Zündorf
AG-Softwaretechnik
University of Paderborn, Germany
[duke|nierej|maroc|zuendorf]@uni-paderborn.de
D-33095 Paderborn

1 Introduction

Typically, 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. In addition to these scenarios one develops 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 [4], TogetherJ [5], and Rhapsody [6], 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 and Rational Rose RT) state-charts. In [1], [7], [8], [9], we propose to use the other UML behaviour diagrams for the specification of method bodies and for code generation.

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 [1], [9]. This abstract illustrates the concepts for recognizing class and behaviour diagrams from Java code.

2 Running Example

Figure 1 shows the structure of a switch [transfer gate] as part of a material flow system[2], 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[3]-node, which is connected to a communication network via a bus interface. This LON-node runs the actual application software.

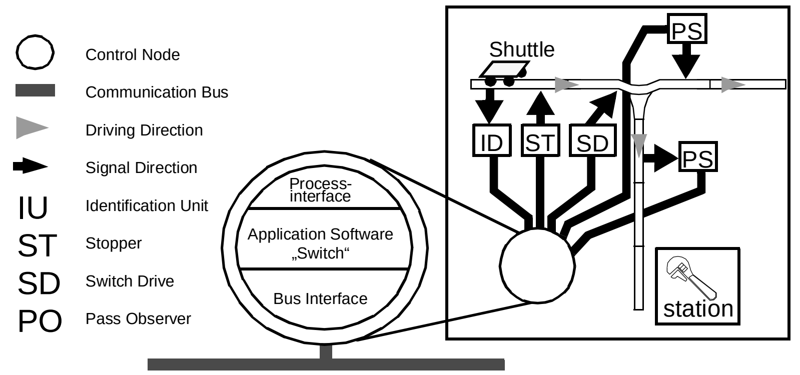

Figure 1 The structure of a switch in the material flow system

3 Reconstruction of class diagrams

According to the generation of Java code out of specifications [1], [9], the reverse step is also devided into two tasks. First, the static information, namely the class diagrams, will be re-

1. **F**rom **U**ML to **J**ava **A**nd **B**ack **A**gain
2. The example stems form our ISILEIT project, funded by the German Research Foundation (DFG).
3. Local Operating Network

constructed and in a second task, the story-diagrams are recognized.

Figure 2 shows a cut-out of the generated code of class `Switch` and `Shuttle`[1]. To reconstruct the class diagram out of these two Java code fragments, first, FUJABA uses a parser to construct a syntax graph for the source code. The parser is generated with JavaCC [10]. 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 rudimentary class diagram out of the parsed information. Such a rudimentary class diagram consists of classes with (*private*) attributes as well as methods, either access methods for attributes and associations and usual methods. Also the inheritance relations (line 1) are recognized directly in this first step.

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

In a second step, the access methods must be filtered out of the classes and associations have to be (re)constructed[2]. 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 such annotations [14]. An example of the annotation structure for the attribute `shuttle_Id` of class `Shuttle` is shown in Figure 3.

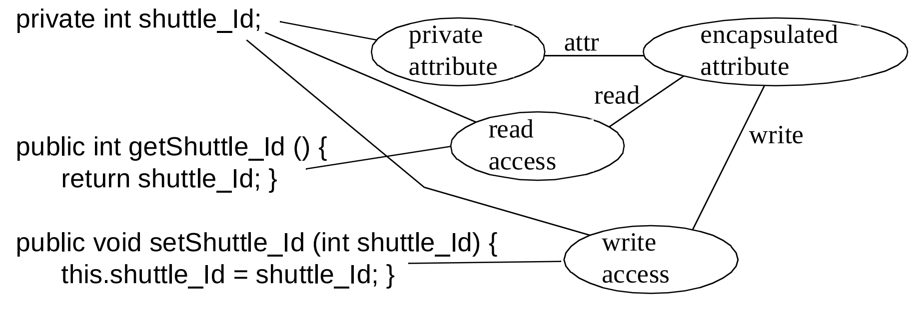

Figure 3 Annotation structure for encapsulated attributes

In the first level the parsed declarations (elements of the syntax graph) are annotated[3]. 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 first uses naming conventions to recognize access methods for attributes. Triggered by appropriate names, it checks the bodies of candidate methods for read or write usage of the attribute. Identified access methods are marked by read and write access annotations. Triggered by these annotations, another engine recognizes that these first-level annotations form an *encapsulated attribute*. So the engine combines the first-level annotations to a second-level encapsulated attribute annotation. To provide a quick access for the connected annotations and diagram elements, the connectors may be tagged with names e.g *attr*, *read*, *write*. Once the second-level annotation is constructed, and thereby, the

1. Only the necessary parts for the recognition process are shown.
2. Fujaba generates an attribute and appropriate access methods for an association as well as specified attributes.
3. The object structure is more complex, but this simplification sufficies for the understanding of the concepts.

attribute and the methods have been classified as an encapsulated attribute, the annotation engine marks the methods as hidden and derives the visibility of the attribute from its access methods. [deutlich genug]

In case of attributes and methods, which serve as access methods for associations, the corresponding annotation structure is more complex, but looks like the above. We assume, that bi-directional associations are implemented as pairs of forward and backward pointers. Thus, write access methods encapsulating an association should manipulate both pointers in order to guarantee the consistency of all pointer pairs. This habit serves as an indicator for the detection of associations and their access methods. Associations are usually [often] implemented using generic container classes. In order to identify the entry type of such containers, we look for calls to their add methods and try to identify the type of the inserted elements, statically. We use traditional compiler techniques to extract these informations, cf. [13].

Figure 4 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 `announced` are hidden as well as the attributes for the association. The described annotation process also works for e.g aggregation, composition, and qualified associations. 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.

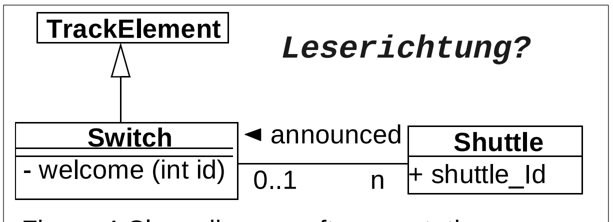

Figure 4 Class diagram after annotation process

4 Reconstruction of Story-Diagrams

FUJABA uses Story-Diagrams for the specification of dynamic aspects. Story-Diagrams are a combination of UML activity diagrams and UML collaboration diagrams. We defined some abbreviations allowing to use collaboration diagrams like graph rewrite rules [3]. Activity diagrams are used to specify the control flow and each activity can contain either pure Java source code as well as a graph rewrite rule. The control flow can be constructed directly out of the syntax graph and like a rudimentary class diagram (see above). Each activity contains exactly one Java statement and branches and loops are displayed as transitions with guards.

ap. 4 ist er gerafft!

Like for the recognition of class diagrams such rudimentary activity diagram are annotated in order to reconstruct the graph rewrite rules (collaboration diagrams). If no graph rewrite 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 the rewrite rule cannot be recognized any more.

Figure 5 shows the annotation structure and the annotated source code for the first reconstructed activity. The top-level annotation is the *graph rewrite rule* annotation, which signals that all containing annotations refer to a graph rewrite rule. Such a graph re-

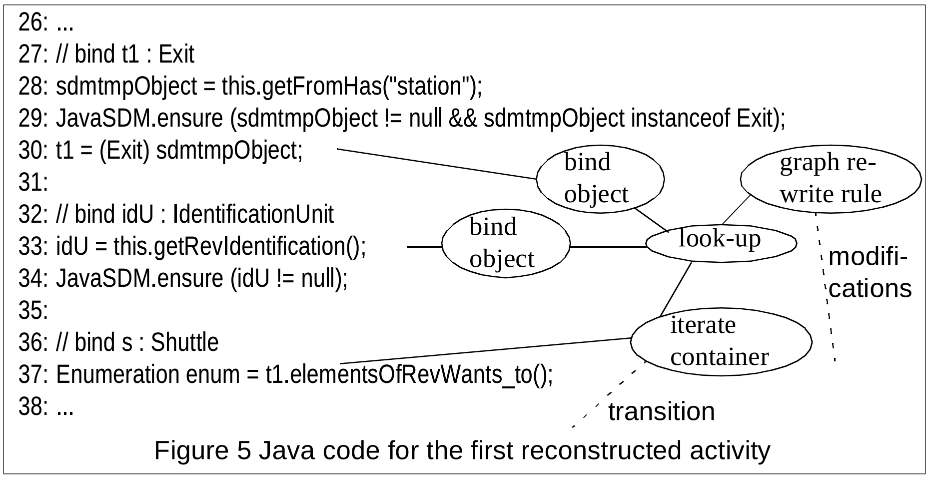

Figure 5 Java code for the first reconstructed activity

write rule annotation replaces all activities and transitions reffering to that graph rewrite rule in the activity diagram by one activity containing the corresponding rule. Thus, the reconstructed Story Diagram is shown, cf. Figure 6.

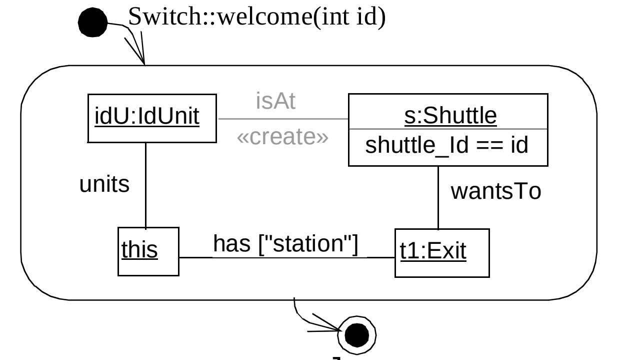

Figure 6 Reconstructed `welcome` Story-Diagramy

Using similar concepts, FUJABA is able to provide support of recognition, creation and completion of design patterns [2]. The round-trip engineering also works if a developer makes manual changes in the source code as long as she/he uses the naming conventions and implementation concepts of FUJABA. To provide a more flexible recognition, we investigate the use of generic fuzzy reasoning nets (GFRN) [11]. We hope that we will be able to reengineer 'legacy' Java code then. For example, the SWING library [12] contains many methods that look like a kind of graph rewrite rule. To deal with vague situations, GFRN's provide a percentual uncertainty. In these cases the reengineer can decide if a part of a source code corresponds to a graph rewrite rule or not.

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.

References

[1] T. Fischer, J. Niere, L. Torunski, A. Zündorf. *Story Diagrams: A New Graph Grammer Language based on the Unified Modelling Language and Java,* in Proc. of TAGT '98 (Theory and Application of Graph Transformations), LNCS 1764, pp. 296-309, ISBN 3-540-67203-6, Springer 1999.

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

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

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

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

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

[7] 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, 1998.

[8] U. Nickel, J. Niere, W. Schäfer, A. Zündorf, *Combining Statecharts and Collaboration Diagrams for the Development of Production Control Systems*. In Proc. of Object-oriented modelling of embedded real-time systems (OMER) workshop, Technical Repport 1999-01 University of Armed Force München, May 1999.

[9] H.J. Köhler, U. Nickel, J. Niere, A. Zündorf, *Integrating UML Diagrams for Production Control System,* to appear in Proc. of the 22nd Intl. Conf. on Software Engineering, Limerick, Ireland, June 2000.

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

[11] 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.

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

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

[14] M.T. Harandi, J.Q. Ning, *Knowledge-Based Program Analysis*, IEEE Software, pp.74 - 81, Jan 1990.