Teaching Object-Oriented Concepts with Eclipse*

Matthias Meyer, Lothar Wendehals
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Straße 100
33098 Paderborn, Germany
[mm|lowende]@upb.de

Abstract

Object-oriented software development is a subject area difficult to teach, especially to beginners. They face a lot of abstraction and (from a beginners point of view) isolated topics, such as the syntax and semantics of a programming language, the functionality of a software development environment and basic object-oriented concepts. Although many professionals in education believe in the "object first" approach as the best method of introducing object-oriented concepts, there is no common agreement on how to start such courses. Current study programs often begin by teaching a programming language, instead of focusing on the basics of object-oriented concepts.

In the last years a learning environment was developed based on a visual programming language to abstract away from details. It assists teaching step-by-step object-oriented concepts and the syntax and semantics of a programming language in secondary schools and first year university courses. Our goal is to port this learning environment to the widely used IDE Eclipse.

1. Introduction

The FUJABA LIFE³ project started in 2001 as a cooperation between two research groups at the University of Paderborn and several secondary schools in Germany. As the outcome of this one year project a learning environment was developed based on the FUJABA TOOL SUITE [7].

The FUJABA LIFE³ approach [4, 5] teaches object-oriented concepts in three phases. Each phase introduces new concepts and encourages the autonomy and personal responsibility of the students.

The first phase starts with the exploration of the problem domain. Objects, object structures and their interactions are introduced. For this purpose the Dynamic Object Browsing System (DOBS) is used, which is an object structure exploration and manipulation tool. It is part of the FUJABA TOOL SUITE. The students instantiate prefabricated classes from libraries and put the objects together to build their first running program. DOBS visualizes the object structure during runtime and offers the chance to "play" with the objects.

After that classes, attributes and methods as well as associations are introduced. For this purpose FUJABA LIFE³ supports UML class diagrams. Furthermore, it offers Story Driven Modelling (SDM) [2] which enables the user to specify complete method bodies with UML activity diagrams where activities are described by extended UML collaboration diagrams. This combination of an activity diagram and extended collaboration diagrams is called *story diagram*.

Students using SDM are able to specify complete applications without being confronted with the syntactical details of a textual object-oriented programming language. They can concentrate on learning object-oriented concepts. In addition, the diagram editors are restrictive and syntax driven. Showing methods as story diagrams also helps students to conceptualize methods as means to operate on object structures. Note, that this visual description of classes and methods is completely independent from programming languages. Especially the description of the method bodies by story diagrams differs from most UML tools, where activities usually are specified by programming language specific code.

Within the second phase a programming language is introduced - here Java. The FUJABA LIFE³ environment generates Java code from the class diagrams and especially from the method bodies specified by story diagrams. The generated code is human readable so that students can directly compare the graphically specified methods with their textual representations in Java code.

In the first two phases the teacher coaches the students and the whole class works together. In the third phase the students consolidate their knowlegde to produce software by working in small teams (about 4 persons). The students cooperate in their group, whilst competing against other groups. Each group works on the same problem. In regular classroom meetings each group presents the current state of their project and the whole class discusses it.

The objectives of our project are to provide a tool for the given learning sequence based on the widely used integrated development environment Eclipse [1] and to support a complete learning environment from object-oriented design to code generation and programming in Java. For this purpose we are integrating class and story diagrams into Eclipse. Another group at the University of Kassel is porting DOBS to Eclipse [6].

The next section describes our Story Driven Modelling approach and code generation. Section 3 is concerned with the integration of our approach into the Eclipse platform. In Section 4 we conclude and point out future work.

2. The Fujaba Modelling Approach

In the following section we outline our modelling approach

*This work is funded by an IBM Eclipse Innovation Grant 2004.

in more detail. We are using a simulation of a house with an elevator as a running example. The house consists of several levels. Persons may enter the house and use the elevator to change between levels.

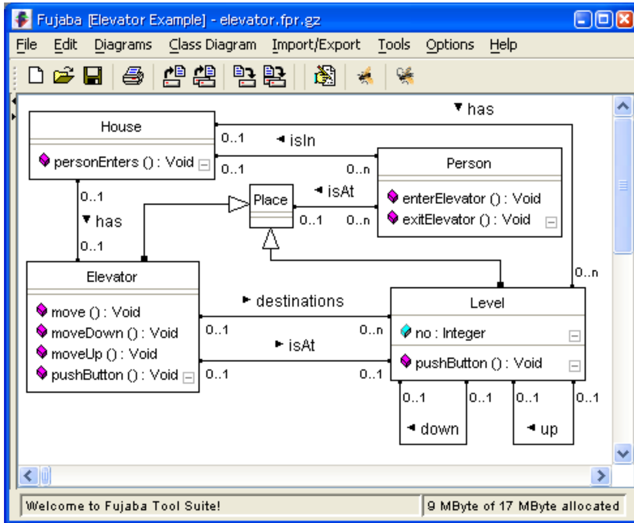

Figure 1: Class diagram of the elevator simulation

The structure of a program is modelled by UML class diagrams. Figure 1 shows a class diagram of the elevator simulation. The house, the levels, the elevator, and the persons are represented as classes. A house may have an arbitrary number of levels and one elevator. Furthermore, an arbitrary number of persons may stay in the house. A person is at only one place at any point of time. The class Place represents a place in the house and the association isAt models the relationship between persons and places. Since the class Place is the super class of Level and Elevator, a person may either be at a level or in the elevator. The elevator in turn may be at a certain level as well. Each level has a number and knows its next and its previous level.

The Elevator is able to handle a queue of destinations, i.e. levels it has to visit to pick up a person for example. This is modelled by the ordered association destinations between Elevator and Level. Destinations can be added by the methods pushButton of Elevator and Level. The movement of the elevator is handled by its method move. The method processes the levels linked to the elevator via destinations in a first in first out order. It uses the methods moveUp and moveDown to go to the current destination level. Entering and exiting the elevator is defined by the methods enterElevator and exitElevator of class Person. Finally, the method personEnters of class House lets a person enter the house.

As mentioned before, our approach also allows for the specification of the behavioural aspects of a program. The behaviour of our elevator simulation is defined by the methods shown in the class diagram. Each method is specified by a corresponding *story diagram.*

2.1 Story Driven Modelling

Story diagrams are UML activity diagrams where activities are specified by extended UML collaboration diagrams called *story patterns.* Story patterns allow for the visual specification of modifications of object structures.

The state of a program at runtime is represented as a state graph and the state changes are specified by graph rewrite rules. A graph rewrite rule consists of two graphs which are called the left-hand side (LHS) and the right-hand side (RHS). The LHS describes a subgraph which has to be found in a host graph (here the program's state graph). The RHS describes the same subgraph after the rule has been applied, thereby specifying the modification of the subgraph. For example, a node or edge in the LHS of a rule which does not appear in the RHS will be deleted. A node or edge which is specified in the RHS but not in the LHS will be created. The host graph and the LHS and RHS of a graph rewrite rule use the same graph scheme which provides node and edge labels to classify the nodes and edges.

Story patterns are graph rewrite rules in which the LHS and the RHS are specified in a single graph. Elements to be deleted or created are marked with stereotypes. As a common graph scheme for the state graph and the story patterns of a program we use the class diagrams defining the program's structure.

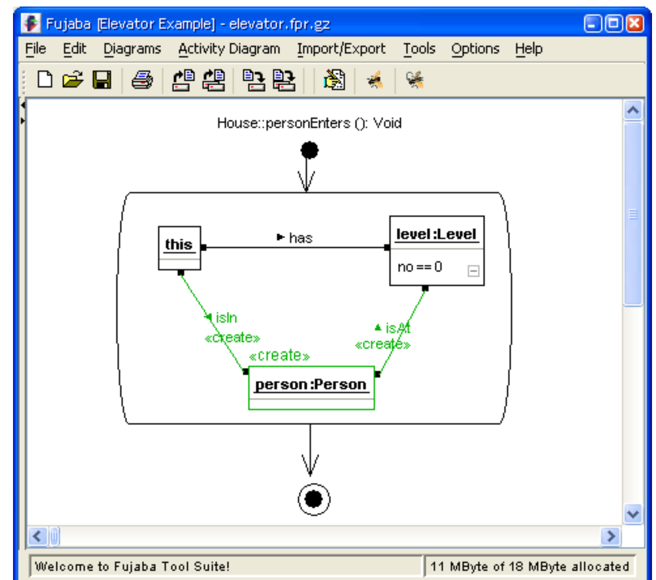

Figure 2: Story diagram for House.personEnters()

As an example, Figure 2 shows the story diagram for the method personEnters of class House. The diagram models a person entering the house at the ground floor. It consists of a start and a stop activity and one activity containing a story pattern in between. The LHS of the story pattern specifies a subgraph which has to be found in the program's state graph. This subgraph consists of the two objects this of type House and level:Level of type Level connected by a link has. In addition, the attribute no of level:Level has to be 0. Since the class diagram shown in Figure 1 is used as the graph scheme, all objects in the story pattern represent instances of classes and all links represent instances of associations specified in the class diagram.

Elements of the story pattern marked with the stereotype ≪create≫ define the modification of the subgraph. They specify the creation of a new instance person:Person of class Person. The new object is connected to the house and the ground level by creating the two links isIn and isAt to the objects this and level:Level, respectively.

The application of a story pattern is done in three steps.

In the first step the specified subgraph has to be found, i.e. the objects belonging to the LHS of the story pattern have to be bound to objects in the state graph of the program. The object this in the given story pattern represents the instance of class House on which the method personEnters is called and thus is implicitly bound. Starting from the object this all other objects in the LHS of the story pattern are bound by following the links between the objects. Any story pattern must contain at least one object which is implicitly bound. If the structure can be found the second step performs all deletions (none in the given story pattern) and a subsequent third step performs all creations specified by the story pattern.

A story diagram is not limited to one story pattern activity. In fact, it may consist of an arbitrary number of activities with story patterns embedded in a complex control flow.

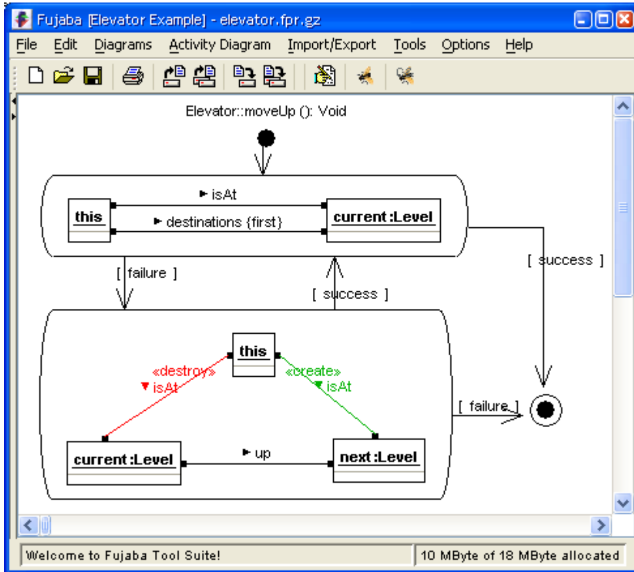

Figure 3: Story diagram for Elevator.moveUp()

Figure 3 shows the story diagram for the method moveUp of the class Elevator. The story diagram lets the elevator move upwards from level to level until the current destination level is reached or the elevator can no longer move upwards.

The first story pattern in the activity immediately following the start activity specifies a subgraph which has to be found. The story pattern checks if the elevator is at the current destination level. The elevator is represented by the object this. The object current:Level is the level the elevator currently is at and thus has to be linked to this via isAt. In addition, if the current level is also the current destination it has to be the first object linked to the elevator via the ordered association destination. If the specified subgraph can not be found, i.e. the application of the story pattern fails, the story diagram proceeds along the transition [failure] to the second story pattern activity.

The second story pattern describes how the elevator moves one level upwards. The elevator and the current level are again represented by the objects this and current:Level which have to be linked via isAt. The next level (next:Level) can be obtained from the current level via the association up. In order to move from the current to the next level, the existing link isAt between this and current:Level has to be deleted. Thus the link is marked with the stereotype ≪destroy≫. Then a new link isAt between this and next:Level has to be created.

If the story pattern can be applied the story diagram proceeds along the transition [success] and the first story pattern is executed again. When the destination level is finally reached the application of the first story pattern will be successful and the story diagram will reach the stop activity. If the elevator may not move upwards any further the second story pattern fails and the stop activity is reached as well.

The two story diagrams show how program behaviour can be specified in an easy to learn visual notation. The examples do not use all available syntactical constructs for clarity and space reasons. Story patterns also provide optional and negative objects and links, path expressions, constraints, and collaboration statements. Furthermore, story diagrams allow for more complex control flow such as conditional branches or iterated story pattern activities. For details see the formally defined semantics of story diagrams based on graph grammar theory [8].

2.2 Code Generation

The formal semantics facilitate automatic code generation for story diagrams. The Fujaba Tool Suite provides automatic Java source code generation for story diagrams as well as for class diagrams. The generated source code is executable and human readable.

The code generation for class diagrams is straight forward. Each class in the diagram is translated into a Java class. Attributes are mapped to private fields with access methods which have appropriate visibility. Methods are translated to method definitions. The code for the method bodies is generated from the corresponding story diagrams. Associations are realised by private member variables with appropriate access methods in the participating Java classes. The one-to-one association isAt between classes Elevator and Level in Figure 1 for example results in a private member variable of type Level in the class Elevator and a private member variable of type Elevator in the class Level since the association is bidirectional. Furthermore, public getter and setter methods are generated for both member variables which also ensure bidirectional links at runtime. For to-n associations container classes are used together with appropriate access methods to store an arbitrary number of elements.

These access methods are used by the code generated for story diagrams. To give an example, Figure 4 shows the code generated for the story diagram describing the method personEnters of class House (cf. Figure 2). The story diagram models a person entering the house at the ground level and specifies the creation of a new person object which is linked to the house and ground level objects.

The generated code starts with the declaration of some variables (lines 2 to 6). The first try/catch block (lines 8 to 38) contains the code generated for the story pattern activity. In line 9 the boolean variable success is initialised with the value false.

As mentioned before, a story pattern is applied in three steps. In the first step (lines 11 to 20) the structure specified in the LHS of the story pattern has to be found. The object this is implicitly bound. Thus, only the ground level object has to be bound which must be linked to the object this via the has association (cf. Figure 1). The one-to-n association has allows an arbitrary number of levels to be linked to

```
01 public void personEnters() {
02   boolean success = false;
03   Iterator iter = null;
04   Object tmp = null;
05   Person person = null;
06   Level level = null;
07
08   try {
09     success = false ;
10
11     // bind level: Level
12     iter = this.iteratorOfLevels();
13     while (!(success) && iter.hasNext()) {
14       try {
15         tmp = iter.next();
16         SDM.ensure(tmp instanceof Level);
17         level = (Level) tmp;
18
19         // attribute condition
20         SDM.ensure ((level.getNo() == 0));
21
22         // create object
23         person = new Person();
24
25         // create link
26         level.addToPersons(person);
27
28         // create link
29         this.addToPersons(person);
30
31         success = true;
32       }
33       catch (SDMException exception) {}
34     }
35   }
36   catch (SDMException exception) {
37     success = false;
38   }
39   return;
40 }
```

Figure 4: Generated code for House.personEnters()

a house. Therefore the code generated for the class House uses a container variable to store several level objects. Line 12 fetches all level objects from this container using one of the generated access methods. A while loop iterates through them as long as the variable success is false and there are more level objects. The following lines are enclosed again in a try block. Lines 15 to 17 fetch the next level object, check if it is of the correct type and store it in a variable. If the type check fails an SDMException is thrown by the method ensure. The exception is caught at the end of the while loop, so that in case of a failure the execution continues with the next iteration. When a level object is found it has to be checked if it is the ground level, i.e. its attribute no must be 0. Line 20 uses the getter method generated for the attribute to check its value. If the check is successful the structure specified by the story pattern could be bound and the first step of the story pattern application is done.

The second step performs all deletions (none in this case) and subsequently the third step (lines 22 to 29) performs the creations. First, a new person object is created (line 23) and is linked to the level object (line 26) and to the house object (line 29) by calling access methods. Note, since the access methods ensure bidirectional links at runtime they have to be called on one of both objects only.

In line 31 the variable success is set to true, so that the while loop terminates and the application of the story pattern ends. In this example the execution of the story diagram ends as well.

3. Integration into Eclipse

Students should not be confused by becoming aquainted with different development environments during the course. All three phases should be supported by one environment. The widely used open source platform Eclipse offers a complete Java development environment. Furthermore, it is easily extensible by plug-ins, which makes it an ideal environment for a seamless integration of the FUJABA LIFE^3 functionality.

The features of FUJABA LIFE^3 to be integrated into Eclipse include class and story diagrams for the visual specification of complete programs. The FUJABA LIFE^3 plug-in features an own perspective. A project tree lists all class and story diagrams specified within the project. The editors for class and story diagrams will be restrictive and syntax driven, i.e. for example if an object is used within a story diagram its class has to be specified in a class diagram beforehand.

Figure 5 depicts an early state of the FUJABA LIFE^3 plug-in. On the left side the Fujaba Navigator lists the given diagrams divided into class and story diagrams. The diagram editor in the middle is showing the class diagram for the elevator example. Classes, associations and generalizations can be directly added to the class diagram by using the tool bar of the diagram editor. Methods and attributes are added via the context menu of a class. Properties of diagram elements can be edited with the properties editor in the lower right corner. An overview of the diagram is given in the outline view in the upper right corner.

The FUJABA LIFE^3 plug-in furthermore supports Java code generation for class and story diagrams. The code will be exported into an Eclipse project. So it can be directly compiled and started from within Eclipse.

4. Conclusions and Future Work

The extension of Eclipse by a plug-in supporting class and story diagrams offers a complete learning environment for object-oriented concepts. Students learn to think of programs as modifications of object structures. The whole program can be specified by an easy to understand visual programming language.

The FUJABA LIFE^3 plug-in furthermore provides an automatic generation of executable Java code. When learning a programming language, students are able to generate code from the class and story diagram specifications and modify it using the well known and widely used IDE Eclipse. Therefore they do not have to change to and become acquainted with a new platform.

The next step in programming education is to teach the students (good) software design. In several lectures given at our research group we experienced that design patterns [3] are well-suited for teaching good design. We therefore plan to extend Eclipse by a design pattern editor. The editor not only offers given design patterns but also enables the user

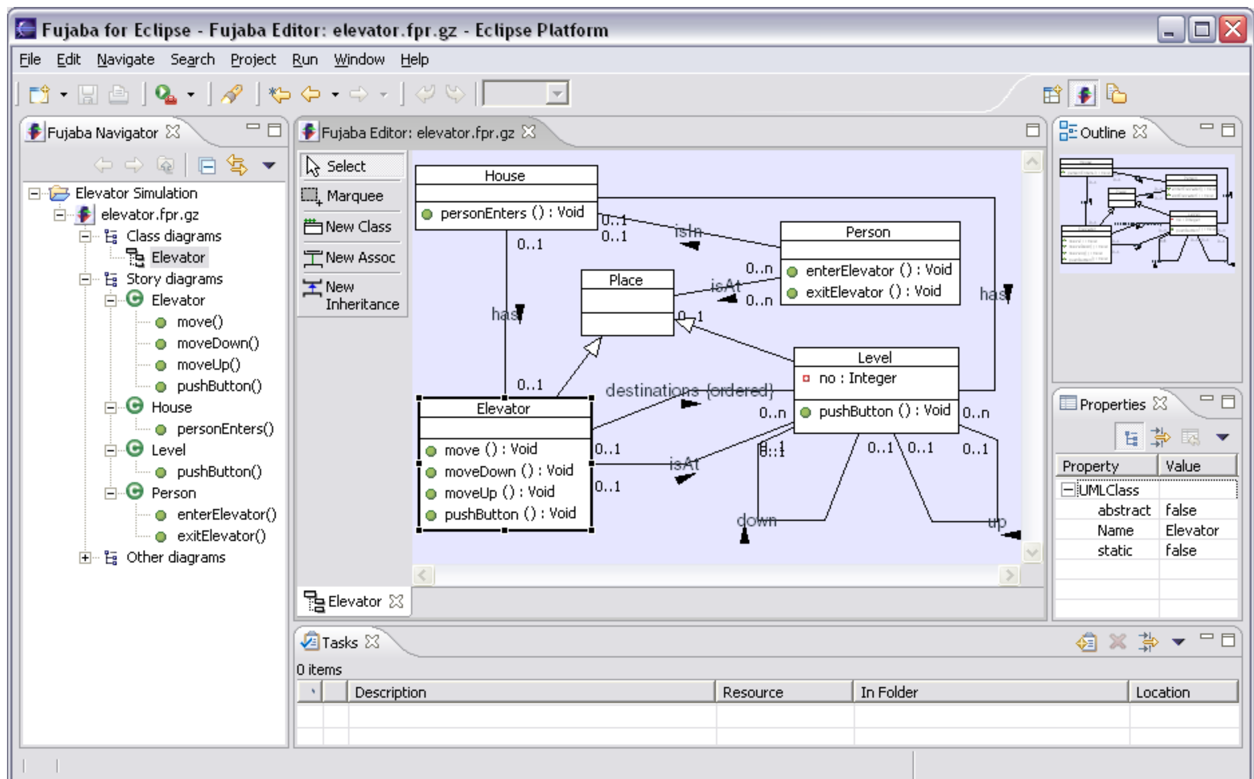

Figure 5: The Fujaba perspective in Eclipse

to define customized or new design patterns. The design patterns can be instantiated, i.e. they can be integrated into a class diagram. Such an editor enables students to actively experiment with design patterns. Thus they are able to add practical experience to the theoretical knowledge offered by the lectures.

Acknowledgements

We thank Oliver Rohe and Dietrich Travkin who are implementing the FUJABA LIFE³ plug-in for Eclipse. Thanks to IBM Corporation for the Eclipse Innovation Grant 2004.

References

[1] Eclipse Foundation. *Eclipse. Online at http://www.eclipse.org (last visited: August 2004).*

[2] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the* 6th *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764, pages 296–309. Springer Verlag, November 1998.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[4] J. Niere and C. Schulte. Thinking in object structures: Teaching modelling in secondary schools. In *Proceedings of the ECOOP Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts*, 2002.

[5] C. Schulte, J. Magenheim, J. Niere, and W. Schäfer. Thinking in objects and their collaboration: Introducing object-oriented technology. *Computer Science Education*, 13(4), December 2003.

[6] University of Kassel, Germany. *eDobs. Online at http://www.se.e-technik.uni-kassel.de/se/index.php?edobs (last visited: August 2004).*

[7] University of Paderborn, Germany. *Fujaba Tool Suite. Online at http://www.fujaba.de/.*

[8] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001. (draft available online: http://www.uni-paderborn.de/fachbereich/AG/schaefer/Personen/Ehemalige/Zuendorf/AZRigSoftDraft_0_2.pdf).