Thinking in Object Structures: Teaching Modelling in Secondary Schools

Carsten Schulte

Didactics of Computer Science
Department of Mathematics and Computer Science
University of Paderborn
Fürstenallee 11
D-33102 Paderborn, Germany
email: carsten@upb.de

Jörg Niere

Software Engineering Group
Department of Mathematics and Computer Science
University of Paderborn
Warburger Straße 100
D-33098 Paderborn, Germany
email: nierej@upb.de

Abstract We present a teaching strategy for the introduction of object-oriented concepts in secondary schools, whereby we focus on active learning of the basic concepts. The concept uses a learning sequence that fosters activities of the learners. Students start to explore a given object-oriented model on analysis and design level. Guidance is continuously reduced towards a project phase in which the students model and implement a small object-oriented application. Our approach uses a graphical language focusing on object structures. We introduce methods as a means to change the object structure of an application and use tools that support the graphical design and specification of executable object-oriented applications. We also present and discuss first evaluation results from two local secondary schools starting their computer science education using our concepts.

1. Introduction

Our experiences in teaching students object-oriented concepts and models within their first two semesters are that many students, who went through a traditional computer science education starting with algorithms, then recursive functions and then pointer arithmetic, use the object-oriented technology but do not spiritualize the concepts. They do not *think in object structures*, which manifests in the way, e.g. they use the object-oriented technology for implementing abstract data types. Students often use arrays or records as the internal container storing the elements of the abstract data type instead of using associations. This makes the container access methods consist of complex control flows and they have to manage the internal container itself. Consequently, traversing all elements of the abstract data type often means to return the internal container, which prevents information hiding and forces inconsistencies of the internal container. They are not aware of concepts such as Cursor and Iterator, which encapsulate the access and prevent inconsistencies.

Our hypothesis is that first teaching object-oriented concepts combined with 'traditional' data structures and algorithms overcomes most of the learning problems and makes it also easier to learn the 'traditional' subject matter. The major argument against such an approach is that object-oriented concepts have to be practiced on running applications and therefore a certain object-oriented programming language has to be used. Unfortunately, a programming language comes along with many syntactic and semantic details, which means a huge learning effort before implementing the first object-oriented application. More abstract languages, e.g. the Unified Modeling Language (UML), on the one hand do not have as many details as a programming language but on the other hand are not executable. Design applications have to be implemented in a certain programming language.

The LIFE^3 project started in 2001, funded by the education ministry of North Rhine Westphalia until the end of 2002, investigates teaching concepts for object-oriented concepts in secondary schools. The project is a cooperation between two research groups at the university and (at the beginning) two secondary schools in Paderborn. The outcome of the project will be learning sequences and practical examples. To be able to produce executable applications and to abstract from implementation details, the project uses the FUJABA environment. FUJABA combines UML with a complete automatic code generation mechanism for consistent models.

In the following section we will introduce the pedagogical background and the differences between teaching in universities and secondary schools. Subsequently, we will describe the FUJABA environment and present the development of an example application. Section 4 presents the course structure used in the LIFE^3 project and section 5 summarizes some early results of our evaluation in the schools. The paper closes with a current status of the project and conclusions.

2. Pedagogical aspects

Object orientation is a subject area difficult to teach, especially to beginners, who face a lot of abstraction and (from a beginners perspective) isolated topics, such as: the syntax and semantics of a programming language, the functionality of a software development environment, the basic object-oriented concepts.

Today teaching object-oriented concepts means to use a bottom-up advance. In a bottom-up approach, the taught concepts seem precisely defined to the beginner, but in exercises beginners are forced to use isolated facts without the ability to grasp a full picture. This may lead to frustration, because understanding is delayed. It often leads to ad-hoc generalizations which turn out incorrect and therefore disadvantageous in later learning steps and in building abstract overview knowledge. It turns out that learners are not able to use concepts outside the context originally learned. In pedagogy this effect is known as *inert knowledge*: students gain knowledge that can be recalled in a test but is not transferable to a real life situation, e.g. in a software development process.

The effect of a top-down approach is that the abstract overview remains separated from the *hands-on* knowledge and skills needed to produce a running application. For example, in an exercise the learners design small class structures but then are confronted with the problem how to specify the methods. In this case the designed class structure does not seem to support the specification of methods. It is likely that after a roughly outlined design students focus their efforts on algorithmic aspects such as the question how to traverse arrays, instead of questioning their general design. Before beginners can relate class designs to specification details of methods they need to learn quite a lot of basic concepts and have to see suitable examples.

In school related learning processes it seems even more important to take care of the motivation of the learners, who are not able to handle learning difficulties as learners in higher education. The first introduction to computer science should stimulate and motivate the learners. We teach of overview knowledge to foster early global, somehow ‘fuzzy’ understanding of object-oriented technology. But we add tasks that show how these global concepts are used to construct running applications. The tasks have a severity level that enables the learners to solve them successfully. It seems crucial to provide the learners with those moments of successfully solving tasks on their own to increase their motivation.

The goals and elements of our teaching idea are very similarly described in a well-known general learning framework called cognitive apprenticeship [Co89]. Cognitive apprenticeship was first introduced to learn reading, writing and mathematics. We adopted this approach to teach object orientation in secondary schools. As mentioned we want to support beginners in order to enable them to be active, to model and implement their models (*active learning*), and teach a quite realistic general overview (in *authentic situations*); cognitive apprenticeship focuses on these two main aspects. Learning is seen as a kind of enculturation to a community of experts; learners pick up the appropriate vocabulary and problem-solving strategies from experts. In order to understand the abstract principles actions have to be *articulated* and *reflected* by the learners. The teachers should *model* those expert strategies by solving a problem (in our context, by performing an object-oriented modelling task) in front of the class, explaining the advance. Following, learners attempt a problem with the guidance of their teacher, who is *coaching* them rather than instructing. Further on the teacher withdraws guidance (*fading*) and gives the learners more complex tasks (*scaffolding*). Therefore an important factor is a suitable *sequencing* of learning activities.

A starting point for such a pedagogical *modelling* can be the use of CRC-cards in a role play [see Berg]. In his article, Bergin gives some advice on choosing appropriate examples for such a role play. Thereby, only the design of an object-

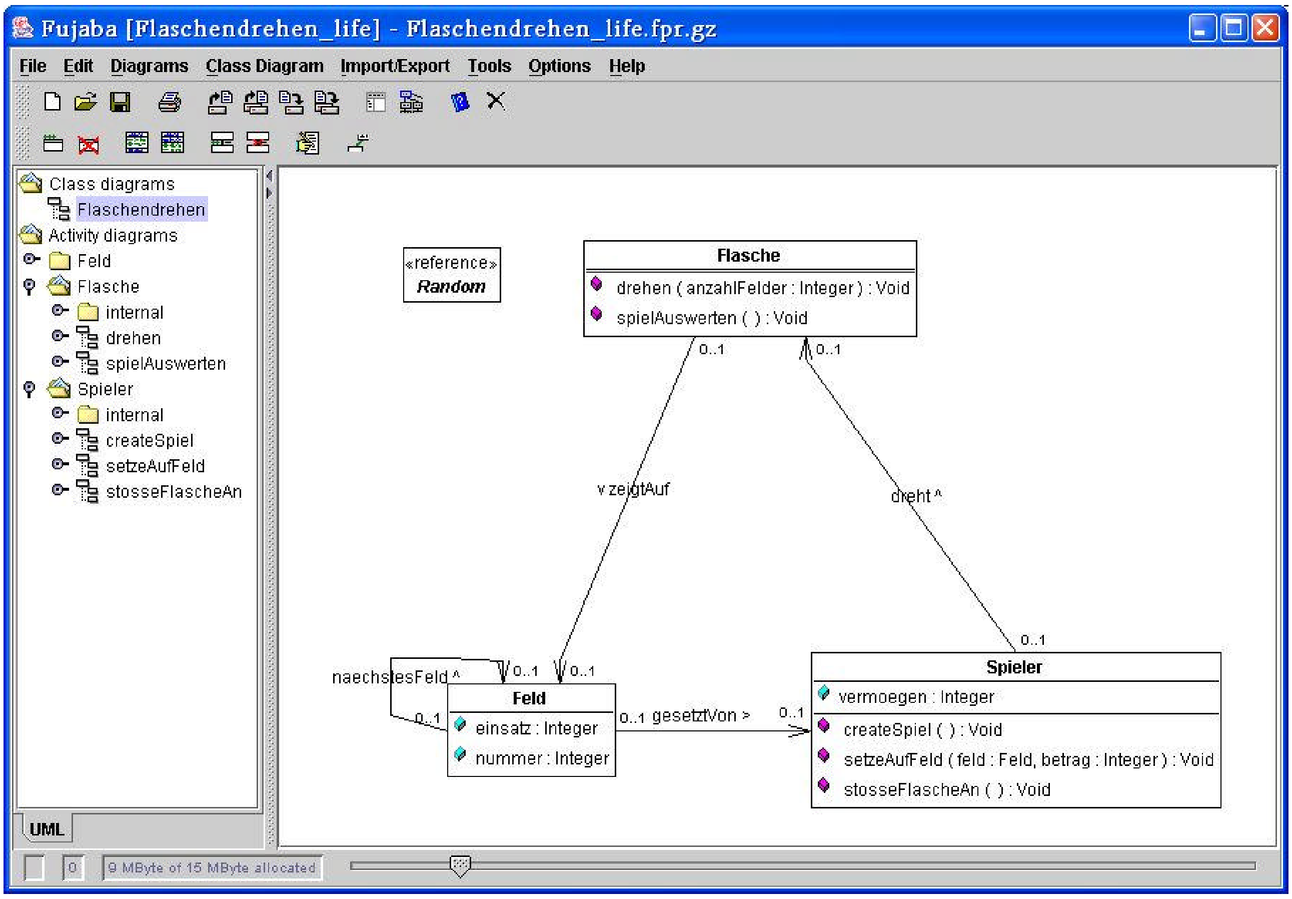


Figure 1 Class diagram of the Flaschendrehen example

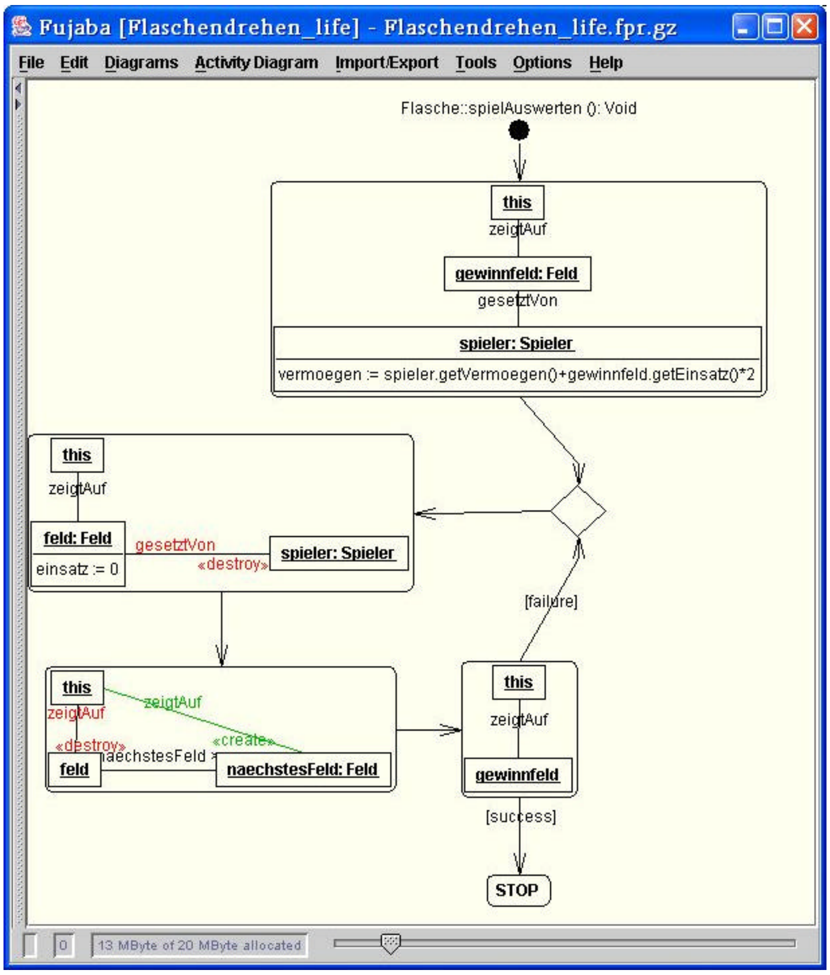
oriented application can be made explicitly. For example, to implement access to array structures a pointer-object can use an attribute to store the number of the actual field in the array; shifting one field further would then be implemented by an increment on the integer attribute. We try to show such implementation aspects through an object-oriented-perspective. We choose examples in which functionality is implemented by methods which manipulate links between objects. In the mentioned example this functionality could be achieved by linked data-objects instead of array-fields. The information which object is actually chosen, is then stored in an association between a data-object and the pointer-object. The shift-operation now can be implemented as a method that manipulates this association. The pointer-object asks the data-object for its successor, deletes the original association and then builds up an association to the next data-object. Methods are introduced as a means to change the object structure of an application. We call these kinds of methods collaborative methods.

In addition to this proceeding we use an abstract graphical language to reduce the learning effort. Today, the Unified Modelling Language (UML) provides a high abstraction graphical language to model several aspects of software systems. Unfortunately, the UML is not formally defined, which makes tools only support the generation of code fragments, which have to be refined in the programming language itself.

3. UML Modeling Tool

Our approach uses the FUJABA environment [FNTZ98, KNNZ00] as development and learning tool. FUJABA supports UML class diagrams to specify static aspect of a software system. Figure 1 shows the class diagram of our introductory example Flaschendrehen. Flaschendrehen is a simple game where a bottle (Flasche) lies in the middle of a circle divided in fields (Feld). Each player (Spieler) can place some coins (vermoegen) on a field. Flasche, Feld and Spieler are classes in the diagram and vermoegen is an attribute of class Spieler. The relations between the classes are shown as references or in UML terms directed associations with cardinality '0..1', which means that no or only one object is reachable traversing the relation. A special relation is the naechstesFeld relation from class Feld to class Feld. This relation is used to specify the cycle of the fields. The class diagram does not specify the arrangement of fields as a cycle explicitly, but it is contained in the initial method createSpiel.

Placing coins on a field is specified by the method setzteAufFeld. After all bets the bottle starts to rotate (drehen) and stops pointing to a certain field. All players who have bet on the field get a profit and all others loose their coins. Method spielAuswerten calculates the profit and sets the vermoegen attribute of each player appropriately.



```
vermoegen := spieler.getVermoegen()+gewinnfeld.getEinsatz()*2
einsatz := 0
```

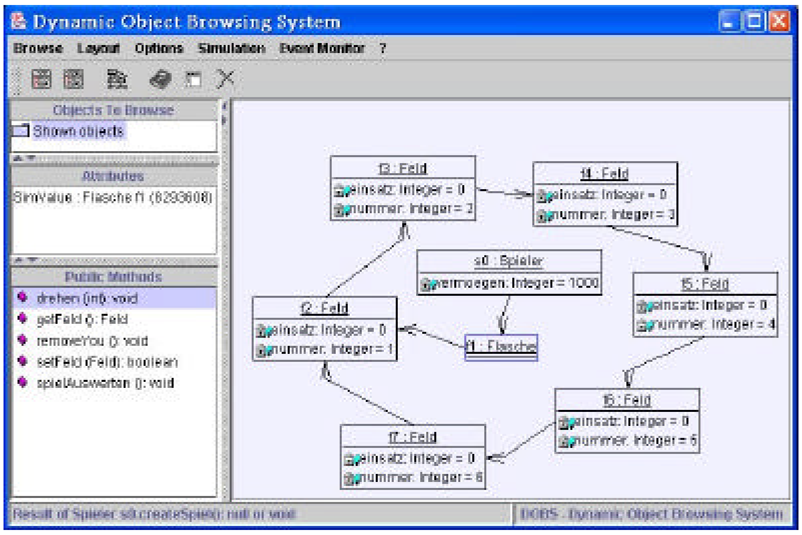
Figure 2 Method body spielAuswerten

In contrast to other tools FUJABA allows the developer of a system to specify also method bodies. Thereby, UML activity diagrams specify the control flow of the method. The NOP-activity (no operation) is shown as a diamond and all others are represented as round rectangles. A transition connects two activities and the attached guard, which is typically a Boolean expression, specifies which activity is next when leaving an activity. The start activity represented as a bullet specifies the entry and the stop activity specifies the exit of the method. Note, the activity diagram has to be well-formed to ensure a direct mapping on control structures in conventional programming languages.

Inside an activity, graph transformation rules [Roz97] specify the behaviour of the activity. The graph transformation rules are notated as simplified UML collaboration diagrams. We call those activities story-patterns and the whole method body specification is called story diagram. For example, the story-pattern in the bottom left corner of Figure 2 refers to the current object with the this object represented as a rectangle. Traversing the links, which correspond to associations in the class diagram, the other two objects are reachable. If all objects and links in the rule could be mapped the modifications will be executed. Links between this and feld are deleted and new link between this and naechstesFeld is created. For more details see [FNTZ98, KNNZ00].

3.1. Testing environment

FUJABA allows a developer to specify the complete application with classes, attributes and methods where the methods are specified by story diagrams. Consequently, the application can be executed and tested. FUJABA generates pure Java source code and uses a conventional Java compiler to generate byte code. The byte code is executable by a conventional Java virtual machine. Executing the running example will not show anything on the screen, because the application has no user interface. Fortunately, the FUJABA environment contains the dynamic object browsing system (DOBS), which visualizes the internal object structure of executed application inside the Java virtual machine. In addition to snapshots of the object structure, DOBS provides an interactive creation and deletion of objects. In addition, DOBS allows the user to invoke methods on certain objects including parameters. Figure 3 shows the DOBS screenshot of the running



SimValue : Flasche f1 (8293608)

drehen (int): void
getFeld (): Feld
removeYou (): void
setFeld (Feld): boolean
spielAuswerten (): void

f3 : Feld
einsatz: Integer = 0
nummer: Integer = 2

f4 : Feld
einsatz: Integer = 0
nummer: Integer = 3

s0 : Spieler
vermoegen: Integer = 1000

f5 : Feld
einsatz: Integer = 0
nummer: Integer = 4

f2 : Feld
einsatz: Integer = 0
nummer: Integer = 1

f6 : Feld
einsatz: Integer = 0
nummer: Integer = 5

f7 : Feld
einsatz: Integer = 0
nummer: Integer = 6

Result of Spieler s0.createSpiel(): null or void

Figure 3 DOBS screenshot after creating the initial situation

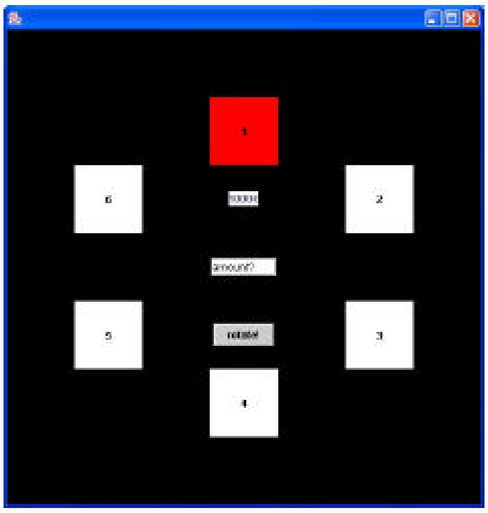


Figure 4 Simple graphical user interface

Flaschendrehen example consisting of the initial object structure. The middle part on the left-hand side shows the attribute values of the currently selected object and the part below shows all methods of the object. Testing an application means invoking methods in a certain sequence. Mistakes found in the application can be eliminated in the specification and a new test run using DOBS can start after generating and compiling the Java source code again.

In order to produce a stand-alone application after a thoroughly testing with DOBS a graphical user interface can be added. Java provides the Swing library to create user interfaces, but the whole library consists of many details possibly confusing learners and also the whole functionality is not needed for our small applications. Therefore, we have developed a simple library based on Swing. The library contains classes for windows, text fields for outputs only, input fields for inputs only, buttons and a panel consisting of circles, lines, rectangles, etc. Each element can listen to mouse events, which provides interaction with the user. Figure 4 shows an example graphical user interface for the Flaschendrehen application. In comparison to Figure 3, the current field, the bottle is pointing to, is marked by a red field and the drehen method will be called when the button is pressed. This is a stand-alone application which only needs the simple graphical library and little runtime library from FUJABA. Thereby, the runtime library contains only some helper functions, which are abbreviations to simplify the generated source code.

4. Course structure

We used the outlined teaching concept and the presented example in an introductory course for 16 or 17 year old students. This concept includes also a kind of didactical software development process. The development process exploits the possibilities of the tool support in order to avoid an independent implementation phase. The process starts with an exploration of the problem space. After that the insights are structured using CRC cards. The learners/developers test the CRC-model playing an object game [Berg], a role play in which the learners act as objects. After that a FUJABA version of the OO-model is developed. DOBS is used to test the basic functionality. In later stages of the learning sequence GUI and event handling is added as the last step of the didactical software development process.

The learning sequence consists of three major phases to assure strategic *scaffolding* and *fading*. In the first phase, the learners become acquainted with basic concepts and the learning environment is introduced. The second phase increases technical knowledge of modelling and programming, and introduces a library to build graphical user interfaces and event handling. In the third phase, practicing autonomous modelling and programming is forced.

The first phase conceptualises an overview of object-oriented technology and introduces the expert language and tools. It *models* object-oriented modelling. The learners play a given object game that introduces the basic concepts of object-oriented models. The model then is presented and can be further explored in DOBS. Learners actively explore the execution of an object-oriented application; this helps to understand the relation between object structure and class structure. The following concepts and tools are introduced: class, object, association, method, attribute, object interactions, object structures, UML class- and object-diagrams, the tools FUJABA and DOBS, CRC cards, object game.

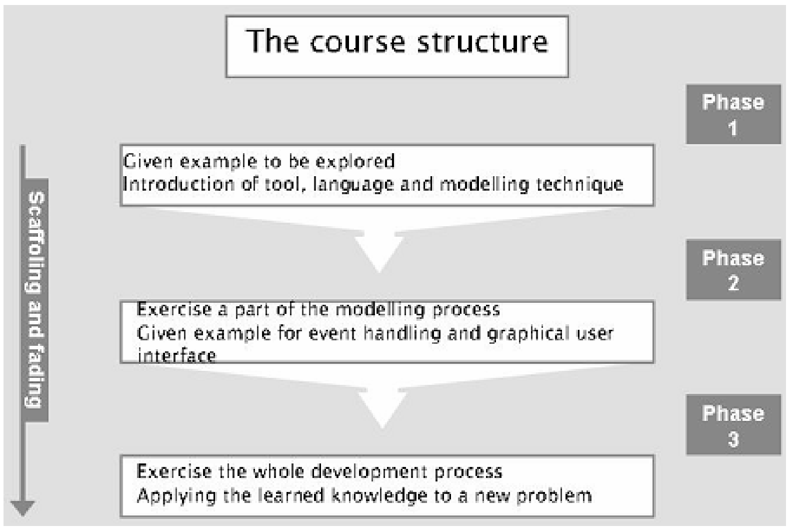

The course structure
Phase 1
Phase 2
Phase 3
Scaffoling and fading
Given example to be explored
Introduction of tool, language and modelling technique
Exercise a part of the modelling process
Given example for event handling and graphical user interface
Exercise the whole development process
Applying the learned knowledge to a new problem

Figure 5 Learning sequence

In the second phase, students gain practice in modelling another example. The students create a CRC-Model, play object game, and model it using UML and FUJABA guided by their teacher. The model should then be refined, thereby introducing more syntactical details of UML and activity diagrams. Once the model is complete, the teacher *models* how event handling and a GUI can be added. In this phase the key aspects of the software development process are introduced. Lessons learned in this phase are: The role and sequencing of different software development steps (Analysis, design, prototype, and test), graphical user interfaces and event handling, basic algorithmic aspects such as loops, conditions and if-statements, the role and use of a class-library, first impression of inheritance and interfaces. In addition, students learn how to recognize and handle errors.

In the third phase, students work in small groups to create a running application with graphical user interface. At this point, they finally work without the guidance of their teacher, allowing them to gain a deeper understanding of the development process and the way in which tools and modelling techniques fit together. In this phase different groups try to solve the same task, this enables competition between groups, but also cooperation within the groups. The need to agree to one model within the group and the expected changes to the original model during the specification in FUJABA gives many opportunities to articulate and reflect on the things learned. In this phase the *active learning* of the students is the main objective. They gain a deeper understanding of the interplay between the different steps and aspects of object-oriented software development, they learn to communicate, valuate and reflect on design ideas.

5. Case study

The teaching concept is currently evaluated in an empirical study at two secondary schools in Paderborn. We record the whole course on videotapes and use video capturing software to protocol the work at the different computers. First preliminary results show that nearly all students distinct clearly between classes and objects. The students were able to develop design ideas with CRC-cards. It does not seem that they were sidetracked by unnecessary details. In the specification phase with FUJABA they were able to find solutions for algorithmic problems, as they were able to reuse ideas observed in the examples of previous phases of the course. Sometimes students need help from the teacher or other groups.

We noticed that tool support helps; the dialogs of FUJABA showed how to create classes, and therefore minimized syntactic problems and reduced the amount of memorized knowledge needed. But tool support alone is not sufficient for a teaching method that supports *articulation* and *reflection*, which are necessary to help students to integrate the different concepts and tasks into overview knowledge of object-oriented concepts and software development processes. We derive this observation from the following: In the first course the teacher encourages discussions and abstractions from the tool, in the second course the teacher encouraged the learners to experiment with the tool and worked more on the visual FUJABA-level then on a representational level. The initial effect was that the first group made slower progress in the first two phases of the course, but had a better understanding of the steps of the software development process. Consequently, in the third phase, the first group created a running application more quickly, because the students made a more thorough analysis and design.

We were impressed of the abstract views students used to communicate their design ideas, especially in the first group. In traditional approaches where a language is introduced first, students often rely on language details to express their ideas – and hence are not able to communicate and evaluate design ideas in early phases of the development process. In such courses one often observes one leader deciding the design without communicating it. In our approach the students are actively involved in developing design ideas, communicating and comparing them between the groups, thereby focussing on finding suitable object structures, so students developed what we call thinking in object structures.

6. Conclusions

In this paper we present a top-down approach for teaching object-oriented concepts in secondary schools. Our approach is based on the concepts of cognitive apprenticeship and *active learning*. First evaluation results show that we have achieved our goal to teach concepts and not details of a programming language. We use the FUJABA environment as graphical specification tool, which produces also executable applications and therefore supports *active learning*.

The major advantage of the FUJABA environment is the absence of source code. The developer uses the high abstraction language, which is UML like, to specify a complete application. The developer is not confronted with syntactical details of a (textual) object-oriented programming language, because the diagram editors are restrictive and syntax driven. Showing methods as collaboration diagrams also helps learners to conceptualize methods as a means to operate on object structures. In addition, the graphical diagram language provided by FUJABA is easy to learn and our experiences with students, developers and researchers from other sciences have shown that the language is also a good basis for discussions: The graphical representation encourages collaboration among the learners. These discussions initiate *articulation* and *reflection* processes that learners need.

In some groups it might be necessary to slow down first hand progress focused on handling FUJABA and give time for building up representational knowledge. Throughout the three phases of the described introductory course the teacher finds many occasions to stimu late this process: For example by group presentations of the current project status, by presenting common problems that occur in different groups, by asking the students to describe what they have done, etc. Another key point of our approach is the concentration of the model and the introduction of graphical user interfaces as of lower importance and later stadium in the modelling process.

A Noteworthy element of the teaching strategy is the first phase of the course: Here the students gain an understanding of the aim of the modelling process: they investigate an executable object-oriented-model. This phase is an important element to achieve *active learning*. The given examples are not only a resource for finding own modeling ideas and for comparing one's own ideas with a given example, but give also a necessary insight for self-contained modeling. Working in small groups on a project with the ability to realize the own ideas also fosters *active learning*.

We are currently analyzing and correlating the recorded videotapes and recorded user interaction with the tools FUJABA and DOBS. Preliminary results underline our observations presented in the previous section. In addition to, in our opinion great success of our approach, we have to investigate how students, who have attended our course, have advantages or disadvantages in consequent courses in comparison to students with a traditional education.

References

[Berg] P. Bergin: The Object game. Online at: http://csis.pace.edu/~bergin/patterns/objectgame.html (last visited 04.04.02)

[Co89] A. Collins, J.S. Brown and S.E. Newman: Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. In: L.B. Resnick: Knowing, learning, and instruction: Essays in honor of Robert Glaser. Hillsdale, NJ, Lawrence Erlbaum Associates, 1989

[BC89] K. Beck and W. Cunningham: A Laboratory for Teaching Object-Oriented Thinking OOPSLA'89 Conference Proceedings, 1989.

[FNTZ98] 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. Springer Verlag, 1998.

[KNNZ00] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In Proc. of the 22nd International Conference on Software Engineering (ICSE), Limerick, Irland, pages 241–251. ACM Press, 2000.

[Roz97] G. Rozenberg (Editor): Handbook of Graph Grammars and Computing by Graph Transformations, World Scientific, Singapore, 1997

[Th99] J. Tholander, F. Rutz, K. Karlgren and R. Ramberg: Design and Evaluation of an Apprenticeship Setting for Learning Object-Oriented Modeling. In: G. Cumming, T. Okamoto and L. Gomez: Proceedings of the International Conference on Computers in Education. IOS Press, 1999