

Vrije Universiteit Brussel - Belgium

Faculty of Sciences

**In Collaboration with Ecole des Mines de Nantes - France
and**

UNLP - LIFIA - Argentina

1999



Teaching Object Technology with Intelligent Environments

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange
project funded by the European Community)

By: Isabel Michiels

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)

Co-Promoter: Dr. Gustavo Rossi (LIFIA Argentina)

Abstract

The objective of this thesis is to explore how to enhance environments for learning object technology with declarative knowledge about good object-oriented programming practices and about the learning process itself. This knowledge will allow us to infer what the student is learning and what he is not. In particular we will enrich an object-oriented learning environment, called LearningWorks, with pedagogical rules (cast as logic predicates) that lead the learning process. The learning knowledge of a particular student will then be used to individualize the interaction of the environment to that student and to guide his learning process in an intelligent way.

Acknowledgements

First of all I thank my co-promoter, Dr. Gustavo Rossi for leading me towards this very interesting subject and for helping me in any way during my stay in Argentina.

I would like to thank my advisor Alejandro Fernandez, for supporting me throughout this thesis, for proofreading my work and for answering all of my questions. He was always prepared to make some time for me whenever I needed it.

I would like to thank Roel Wuyts for assisting me with my thesis by proofreading and answering all those mails I sent him.

Also thanks to Bart Wouters for proofreading parts of my work.

I would also like to thank everybody at LIFIA Argentina for giving me a comfortable and enjoyable workspace and for assisting me with any kind of problems that occurred during my stay there, especially Gabriela. Guillermo and Annabella deserve a special thanks too for being there whenever I needed any kind of help, but most important of all, for being my friends.

My partner-in-crime Ilse also deserves more than a special mentioning here. She was there for me whenever I needed something. I want to thank her for being a friend and companion throughout these 5 adventurous months and for putting up with me. Thanks Ilse.

Also a lot of thanks to our computer suppliers Guillermo, Anabella and Maria-Laura for making it possible to work at my home too.

I also want to thank the other students of the EMOOSE program. The time we spent together in Nantes was great and it gave me the courage to keep on working when times were tough. Thanks to all of you.

I thank my mum, grandmother, my brother Kristof and Olga for their long-distance support throughout this year. I thank Frank also for being there,

for proofreading, for caring and for cheering me up those times when I really needed it. I realize that these last few months could not have been easy for them too, so also thanks for being patient and for believing in me.

Finally, I thank everyone who delivered a contribution to setting up this wonderful EMOOSE program. Special thanks to Annya Romanczuk at EMN for making my stay in Nantes as pleasant as possible. Also a lot of thanks to Dr. Carine Lucas, Wolfgang De Meuter and Prof. Dr. Theo D'Hondt for introducing to me the EMOOSE program in the first place and for helping me with all the administrative tasks that had to be accomplished.

This year of study has really been so incredible: first of all, the curriculum of this EMOOSE program made an enormous contribution to my knowledge on object orientation. Second, I got the chance to know other people, other cultures, another language and I could visit wonderful places I would never have had the chance to see otherwise. And, I learned a lot about life itself.

Isabel Michiels
August 10, 1999
La Plata, Argentina

Contents

Acknowledgements	1
List of Figures	6
List of Tables	7
1 Introduction	8
1.1 Objectives of this work	9
1.2 Object-Oriented Technology	10
1.3 Teaching about Object Orientation	10
1.4 Intelligent Learning Environments	11
1.5 Need for Representing Knowledge	12
1.6 Our Approach	12
2 Teaching Object Orientation	13
2.1 Main Problems	13
2.1.1 Languages	14
2.1.2 Programming Environments	15
2.1.3 Object Misconceptions	16
2.2 What, When and How should we teach?	17
2.3 Existing Approaches	20
2.3.1 Language-Independent	20
2.3.2 Language-Dependent	21
2.4 Our Teaching Approach	22
3 Learning Environments	24
3.1 Introduction	25
3.2 Why use a CBLE?	25
3.3 A Learning Environment: LearningWorks	26
3.3.1 Structure	27
3.3.2 The Curriculum	29
3.3.3 Implementation	30
3.4 Intelligent Tutoring Systems	30
3.4.1 Components of an Intelligent Tutoring System	31

3.4.2	Existing Intelligent Learning Systems	32
3.5	Conclusion	33
4	SOUL	35
4.1	Logic Programming	35
4.1.1	Characteristics	36
4.1.2	A Comparison with Imperative Programming	36
4.2	Logic Meta-Programming	37
4.2.1	Terminology	37
4.2.2	A logic meta-programming system	38
4.3	An Overview of SOUL	38
4.3.1	Comparing SOUL with PROLOG	38
4.3.2	Some Examples	39
4.3.3	The SOUL Declarative Framework	40
4.3.4	The SOUL System Repository	40
4.3.5	Other Application Domains	40
5	An Architecture for learning OOT	44
5.1	Preliminaries	44
5.1.1	Requirements	45
5.1.2	Limitations	45
5.2	Our System Architecture	46
5.2.1	An overview	46
5.2.2	The Student Model	47
5.2.3	The Pedagogical Model	48
5.2.4	The Domain Knowledge Model	49
5.2.5	The Communication Model	52
5.3	Remarks	53
6	A Case Study: LearningWorks	55
6.1	A Comparison with existing ILE	55
6.2	An overview	56
6.3	Event logging	57
6.4	Construction of the Student Model	58
6.5	Conclusion	61
6.5.1	Results	61
6.5.2	Remarks	62
6.5.3	General Conclusion	63
7	Future Work	64
7.1	A Framework for Teaching Object Technology	64
7.2	Structure of the knowledge	65
7.3	More detailed knowledge	66
7.4	Extending LearningWorks	66

<i>CONTENTS</i>	5
7.5 Pedagogical Patterns	67
8 Conclusions	69
8.1 Motivation and Initial Goal	69
8.2 Summary and Results	69
8.3 Final Conclusion	71
Bibliography	71
Index	76

List of Figures

3.1	The LearningWorks CourseBinder	27
3.2	Example of a Learning Book Page	28
3.3	Interaction of components in an Intelligent Tutoring System .	31
4.1	Composite Pattern Structure	42
5.1	Our Architecture for Learning OOT	47
5.2	OOT Learning Path	48
6.1	Our Architecture and LearningWorks	56
6.2	Schematic view of the Event Logging	58
6.3	Example of the Turtles book in LearningWorks	59
6.4	Example of a Rehearse Page in LearningWorks	60
6.5	Example of a page in the shapes book where multiple shapes can be selected	61

List of Tables

2.1	Requirements for an object-oriented teaching language	14
2.2	Requirements for a teaching environment	16
2.3	Example of an Object Misconception as in [HGW97]	17
7.1	Pedagogical Pattern format	68

Chapter 1

Introduction

During the past decade, the object-oriented paradigm gained increasing importance to the software community. The major benefits of object-oriented programming, like reuse for example, make it ideal for teaching novices the programming methodologies currently deemed important. However, the question arises of how, where and when object-orientation should be taught at colleges, universities and also in the software industry.

Learning Object-Oriented thinking [PR95] is a key issue for succeeding with objects. While this sounds relatively easy, this new way of thinking has proven to be very difficult to learn. Misconceptions about what learning object-oriented technology is all about further complicate the learning process, and make it harder than it should be.

One common misconception is to think that learning Object-Oriented Technology is just learning a certain methodology and that by *following the book* anybody will succeed with objects. But this definitely doesn't teach how to think in terms of objects. At most you will find a set of guidelines on how a software development process should look like or a set of visual formalisms, what is far from sufficient to have the right *way of thinking* about objects.

Training in general means bringing somebody to a desired standard of efficiency by means of *instruction* and *practice*. This definition is very general and it can be applied to anything, including training people in the technology of objects. Their behavior must be shaped by means of *instruction* and they must *practice* to learn how to think in an object oriented way.

The most important part of the training process is the *learn by doing* part. To learn object-oriented thinking a pure object-oriented approach is the best way to go, because the object-oriented paradigm consists in fact of a small and complete set of basic rules the learner must follow. A pure Object-

Oriented programming environment is also of a major importance, so that it can focus on all important object-oriented concepts in a consistent and easy way. *LearningWorks* is such an environment; it is built on top of Smalltalk. It also hides the minor drawbacks Smalltalk has as a teaching *instrument*, like the size of the Smalltalk libraries, or the language syntax. For a novice it takes too much time to get comfortable with them.

However, just letting students practice their object-oriented way of thinking by working with a learning environment sometimes isn't sufficient to let them learn the important concepts about object technology in a supervised way. A lot of guidance is needed, but more human experts are not always possible, for example in distance learning or homework activities.

Representing knowledge about a learner offers a solution to the guidance problem. Recording what the learner is doing in the learning environment can really be useful to draw conclusions about the learning level of the student. The environment can even act upon that gathered knowledge and advise to what step needs to be taken next. This way, creating a knowledge base on each individual learner leads to an intelligent learning system that provides a good practicing environment and at the same time well-founded instructions can be provided based on that knowledge.

1.1 Objectives of this work

The objective of this thesis is to show that learning environments for teaching object technology can profit enormously by using declarative knowledge about good object-oriented programming practices and in particular about the learning process itself for each student. This will be shown by presenting a general architecture for teaching about object technology. This architecture will then be validated by applying it to the learning environment *LearningWorks*. A knowledge base will be built as presented in the architecture, thereby making *LearningWorks* more intelligent.

The rest of this chapter will be a short overview of all the important topics that will be addressed in our work. We will start by mentioning object-oriented technology, because this is what this work is all about. The next topic is what we should teach in order to make people good object-oriented software engineers. Afterwards, we will discuss intelligent learning environments and what they are all about. The need to represent knowledge will then be highlighted and we will end this chapter by presenting our approach throughout this dissertation.

1.2 Object-Oriented Technology

Over the last years the object-oriented paradigm has gained a lot of importance. Traditional programming languages have been extended by providing object-oriented features, and newly developed languages support object orientation.

Also in software engineering object-orientation has brought on a true revolution. While polymorphism and inheritance are the cornerstones for reuse at the implementation level, software engineering was also affected and brought reuse at the design level (for example with frameworks). The adoption of reuse on such a large scale, is undoubtedly the most important advantage object-oriented languages have over procedural languages.

A language that is object oriented is characterized by three features:

- abstract data types,
- inheritance , and
- *polymorphism*, a particular kind of dynamic type binding.

Recall that an *abstract data type* is defined as a data structure in which the data and its operations are defined together in a single syntactic unit. This unit is called *encapsulated* if the data can ONLY be accessed and manipulated through the external interface the unit provides. *Inheritance* as second feature offers a solution to the modification problem posed by abstract data type reuse. If a new abstract data type can inherit all of the data and functionality of some existing type, and is also allowed to delete or modify some of those entities and add new ones, reuse is greatly facilitated. The third characteristic of object-oriented programming languages is a kind of polymorphism that is provided by dynamic type binding. It is one of the key positive features of object-oriented programming, because it allows abstract data types to be truly generic.

The unit control concept of object-oriented programming, the *object*, is modeled on the idea that programs simulate the real world. Because much of the real world *consists* of objects, a simulation of such a world should include simulated objects. In fact, a language based on the concepts of real-world simulation need only include a model of objects that can send and receive messages and react to the messages they receive [Seb96].

1.3 Teaching about Object Orientation

Since object-oriented languages and techniques already made their entrance in almost any company, we think it is necessary to focus on how to train

people and what to teach to them to make them good object-oriented software engineers.

In [Gol93] it is said that there continues to be a mismatch between what we teach and what we need to learn. A lot of teaching still puts emphasis on teaching data structures and algorithms instead of teaching ease of understanding and maintenance that comes from good architectural design [Gol93].

It is argued in [Dod99, PR95, LPR94] that the first step to take is to learn people how to think in terms of objects. The only way to really accomplish this is practicing: letting people try out things and let them be confronted with learning environments. LearningWorks [Gol97] is a learning environment that is aiming on making people think in terms of objects by using a representation of the world by means of Microworlds. An object-oriented Microworld [LPR94] is a computer-based representation of some portion of the real world built around objects and classes.

In [Dod99] it is also stated that *mentoring* is really a software development culture, and that it shouldn't be limited to an outside mentor giving lectures and helping during exercise classes. Good environments with a lot of guidance are an essential factor, and are called *intelligent environments*.

1.4 Intelligent Learning Environments

The term *Intelligent Learning Environments* (ILE) [Cos92, BPLR91] refers to a category of educational software in which the learner is *put* into a problem solving situation¹. A learning environment is quite different from traditional environments based on a sequence of questions, answers and feedback. To show what a learning environment is all about, consider a flight simulator: the learner does not answer questions about how to pilot an aircraft, he learns how to behave like a *real* pilot in a rich flying context.

Experience with learning environments (LOGO [Pap92] is one of those environments) showed that those systems gain efficiency if the learner is not left on his own but receives some assistance. This assistance may be provided by a human tutor or it may be implemented into the system.

To summarize, we use the word *intelligent learning environment* for learning environments which include a problem solving situation but also some kind of mechanism to assist the learner in achieving his goal and monitor his learning. We will show that this assistance can be provided by representing knowledge about the learner's learning process.

¹<http://tecfa.unige.ch/edu-comp/edu-ws94/contrib/schneider/advanced.fm.html>

1.5 Need for Representing Knowledge

Since the learning process is a fragile and difficult path to follow, it is not recommended to let the student work on its own and try things out without offering any kind of guidance besides lecture notes.

During these past few years of teaching object technology, some misconceptions that students pick up during their learning period have found to be very hard to *unlearn* after a while. In the same way it was found that teaching object technology to learners that already knew a procedural language was much harder than teaching it to novice learners. This is so, because the *unlearn* process means changing a person's *way of thinking*.

Representing knowledge about a learner offers a solution to this problem. Recording a user's activities inside a learning environment can lead to very interesting results. That information can be used to offer guidance to the students, what they should do next, or if they need more exercises if they didn't understand a concept very well. This way the learning process is more *supervised* and there is a bigger chance to intercept and guide wrong ideas about some concept.

1.6 Our Approach

In this thesis we will present a general architecture for teaching about object orientation. We will validate our architecture by extending a learning environment built in Smalltalk, called *LearningWorks*, thereby adding some intelligence to it by building a declarative knowledge base about the student's activities.

We will start in Chapter 2 by talking about teaching, more specifically how to teach object-oriented concepts and the main problems encountered. Chapter 3 will be about learning environments. What are they and how do existing environments look like? Chapter 4 presents a logic meta-language we will use in the following chapters to represent knowledge about learners in a learning environment. Our general architecture for teaching about object technology will be presented in chapter 5. We will identify and explain the different models we used for building this architecture. Then we will construct such an architecture for the learning environment *LearningWorks* in chapter 6. Chapter 7 talks about further interesting research that can be done in the context of this work and we end with chapter 8 by presenting what we achieved by doing this research project.

Chapter 2

Teaching Object Orientation

In recent years, object-oriented programming became the most important development paradigm. It is nowadays widely used in industry and education, and almost every university teaches object orientation in its computer science curriculum.

These days, everybody involved in software engineering agrees that teaching object orientation is a good thing, because it really supports the concepts that educators have been trying to teach for many years, like well-structured programming and software design. It also has proven very useful for team programming, that deals with issues like reuse and in particular maintenance of large software systems.

But how should a good introductory course on object orientation look like? Some decisions have to be made regarding the programming language and the environment that will be used. This chapter focuses on the essential concepts important for teaching object orientation. We discuss what the main problems are and how they should be dealt with. We end by presenting the existing teaching approaches and we discuss our approach for this work.

2.1 Main Problems

Many reports that discuss teaching object orientation include a long list of problems encountered in the teaching process. A lot of them are caused by the language or environment used for teaching. The language should only aim for the concepts that need to be learned and any kind of distractions should be avoided. However, some languages have inherent deficiencies by which distractions are impossible to avoid.

This section will present the main problems that object-oriented languages and development environments face in order to serve as a teaching language. We will formulate the necessary requirements a good teaching language should certainly have. Object misconceptions are also briefly discussed in

this section.

2.1.1 Languages

Choosing a suitable programming language to expose students to object oriented concepts is essential when using a language dependent teaching approach. A teaching language should be able to focus only to the important concepts that have to be taught to the learner and not on distractions like a difficult syntax or low-level constructs. Table 2.1 shows the requirements for an ideal teaching language [KKR99, Kol99a].

Requirements
Clean, simple and well-defined concepts
A <i>pure</i> object-oriented language
No constructs without semantic value
Easily readable and consistent syntax
Well-defined, easily understandable execution model
No redundant language constructs
Support for correctness assurance
An easy-to-use development environment

Table 2.1: Requirements for an object-oriented teaching language

When we try to map existing programming languages onto this list of requirements, not one of them meets them all. This is mainly due to the fact that existing languages are built to serve industrial applications. Other, not teaching related factors like efficiency, are considered important in that context.

We will now evaluate briefly the object-oriented programming languages C++ and Smalltalk in the context of using them as a teaching language. We've chosen to put a brief evaluation of C++, because it is frequently used as a teaching language. An evaluation of Smalltalk is relevant within the context of this work, which we will see in a later chapter.

C++ Although C++ is one of the most widely used object-oriented languages, it is the worst candidate for being a teaching language. First, it is not a *pure* object-oriented language. It can support object-oriented programming as well as non object-oriented programming, leading to the temptation to develop applications that are not at all based on object technology. To state it otherwise: the language does not encourage the use of object-oriented concepts.

C++ has also many redundant features. Many different language constructs exist for the same semantic concept. Another well-known source of problems

is the explicit static storage allocation, that really forces the learner to think at an unnecessary low level. We conclude by saying that C++ is very badly suited to serve as a teaching language.

Smalltalk Smalltalk is an absolutely pure object-oriented language, that supports the concepts of object-oriented programming in a clean and especially a consistent manner. It enforces the user to program and to think in an object-oriented way. Smalltalk also has an integrated graphical programming environment. So most of the above requirements are met.

Unfortunately, there are also some problems for using this language to teach object technology. It can be argued that the system is too big. Smalltalk offers a very large class library and it is known from experience that students have difficulties coping with that library. It is also found that the environment is sometimes too complex and too confusing for beginning learners. So it usually takes some time before a learner can start to concentrate on the object-oriented features, and this is not what is intended.

2.1.2 Programming Environments

Better programming environments have become a necessity, for example because a lot of introductory courses are meant to teach novices the principles of object-orientation. These first-year students need a good environment to be able to cope with this additional complexity, being that object-oriented programs can sometimes be a complicated web of communicating classes. For earlier introductory courses it was not so crucial, because only attention was paid on the developments of algorithms, so an editor and a compiler could already be accepted.

As object-oriented programming became increasingly popular, there already have been lots of attempts to bring together developments environments and object-oriented technologies. Environments were developed that support the development of object-oriented programs, sometimes accompanied by object-design tools.

In [Kol99b] a study is presented on the requirements that should be met for programming environments to be suitable for teaching. They are depicted in table 2.2. Two facets are thereby kept in mind: suitability for teaching and special object-orientation support. According to these requirements we will discuss the Smalltalk environment and one of the most widely used C++ environments, Visual C++.

The Smalltalk environment One of the most interesting environments with object-oriented support is the Smalltalk environment. A browser is provided to support the use of a class library. Programming is done in a consistent manner, i.e. defining your own classes or reusing the classes from the class library is presented in a uniform way. They also provide a high level

Requirements
Ease of use
Integrated tools
Object support
Support for code reuse
Learning support
Group support
Availability

Table 2.2: Requirements for a teaching environment

of interactivity and object support that is absolutely necessary for having a good teaching environment.

Despite all of these qualities, this environment lacks some things too, like visualization tools for class relations. This is not so easy to do because Smalltalk is not statically typed. Extracting usage relations can be done, but not for all cases.

Another problem pointed out by teaching experience is its size. Teachers saw problems that some students are unable to cope with the environment. Learning by experimenting did not succeed because students were overwhelmed by the system.

Visual C++ Visual C++ is an integrated development environment from Microsoft. It exist also for other languages, like Java (J^{++}). It is a mature environment with well-thought out features, but it fails in providing specific support for object orientation. If we go through the requirements in table 2.2, ease of use, object support, class visualization, collaborations between objects, are not supported very well. The environment includes a lot of functions and options that are only suitable for professional development and these could scare novices. You could say that this programming environment is aimed at a more expert user who needs the flexibility of a professional software development environment. For teaching purposes however, this environment fails the requirements.

As a conclusion, in [Kol99b] it is stated that many of the existing environments are not object-oriented at all. They support object-oriented languages, but they do not exploit object-orientation at the environment level.

2.1.3 Object Misconceptions

Another issue that can have a major impact on the learning curve of a novice learner are misconceptions. When object concepts are taught, especially in the first stages of learning, they are usually accompanied by lots of practical demonstrations and with experts' help. However, it is very unrealistic to

assume that teaching conditions are always ideal. Distance learning is an example of a teaching method becoming more and more popular, but it doesn't always offer enough guidance for students. Expert teachers are also very scarce and hard to find. When demonstrations are limited and guidance is minimal, it might happen that students acquire *object misconceptions*, which can be very hard to *unlearn* later.

In [HGW97] six of these misconceptions are identified. Table 2.3 shows an example of such a misconception to make clear what we mean.

Object/Class Conflation	
Description	When presenting a series of examples in the early stages of learning, it is easy to find oneself by coincidence using examples of classes in which only a single instance of each class is used
Problem	This can lead to a confusion between classes and their instances
Solution	It is good practice to always choose examples very carefully; try to work with several instances of each class in any given teaching example

Table 2.3: Example of an Object Misconception as in [HGW97]

To avoid misconceptions as much as possible, mentoring or guidance is needed. In [Dod99] mentoring is discussed in the context of OO training and education. They stress that mentoring is a software development culture and is of a great importance. In section 2.3 we will explain our approach of teaching and how we will deal with the mentoring problem.

2.2 What, When and How should we teach?

Because of the increasing interest in Object Technology, and consequently also in teaching this paradigm, answers need to be found on certain questions: what should be taught to transform programmers into good software engineers? The next question that keeps popping up is *when* we should teach the concepts of this paradigm: in a first-year programming course, or in a more advanced course that is taught later in the curriculum. The pedagogical side of the problem to be thought about is *how* we will teach these concepts.

What?

The biggest issue to overcome in teaching is seeing the difference between what we *teach* and what we *need to learn*. Up until now, in some curricula, students are only taught algorithms and data structures, and how to perform clever coding. However, this bottom-up teaching approach doesn't teach to design well-structured and reusable programs, and does not focus on how to see and manage large software programs. As a result, the students become professionals who create systems that keep growing, and often get out of control, which often results in a legacy system [Gol95].

The pure *top-down approach* is the opposite approach, that aims at first getting the complete structure of a piece of software right, before getting into implementation issues. This approach is based on the foundations that system building should be done by first learning how to manage the big picture, and then zooming in until the implementation is reached. However, this approach also raises some questions. First of all it has to be considered that programming topics are important for a novice to develop some way of thinking about objects, a state of mind of how to think in an *abstract* way. Just teaching a modeling technique (like UML) to novices to make them think in terms of objects does not suffice. The student will have difficulties in defining what objects to use, because he has no way to picture an object in his mind. As stated in [Gol95], programming topics are important, especially when taught as a way to think abstractly, but together with the concepts and ideas of system building and *not* in isolation. That way a more iterative approach is used, where the student learns to handle a problem in a top-down, structured manner, while not neglecting the implementation.

Within this work and within the learning environment that will be presented in chapter 3 we will focus on how to teach software building to novice learners by using a special developed learning environment. This approach is mainly top-down and focuses on learning people to think in an abstract way by visualizing objects and letting the learner experiment by sending messages to it. This approach therefore ensures that they learn and understand both the general ideas and principles of object-oriented programming, and the implementation of their design.

When?

For a long time, object-oriented programming was considered to be an advanced subject that should be taught late in the computer science curriculum, preferably after first learning a *classical* programming language that deals with procedures and functions. However, education experience has led experts to change their mind, because it was found that people with a procedural background had greater difficulties in learning object-oriented concepts than learners having no programming experience at all.

Two reasons support the thought of teaching object orientation as soon as possible in the Computer Science curriculum:

- The object-oriented method is very general and it provides an excellent intellectual discipline, it can only prepare students for the later introduction of other paradigms such as logic and functional programming, and learning a traditional programming language afterwards, the knowledge of the object-oriented method will make it possible to use those languages in a safer and more reasoned way [Mey93],
- Since object-oriented programming requires a better but completely different way of thinking, teaching OO after the procedural way makes it then necessary to change a person's way of reasoning, which is very hard to do. This problem is referred to by the *paradigm shift*.

The teaching approach we will use in this work will be focused on teaching object-oriented concepts to novice learners, students that have no programming experience at all.

How?

Besides the fact that object orientation affects *what* can be taught to people, it also suggests thinking about new pedagogical techniques, *how* we will teach this paradigm.

Up until now, most pedagogical approaches were based on first teaching elementary stuff about programming in general. Examples to demonstrate the concepts were limited to calculating the factorial of a number, or the Fibonacci numbers, which is not very exciting for the student.

When Teaching object technology in a good environment (see section 2.2) and especially by using good libraries, a less conservative approach is possible by letting students access the library soon in the teaching process. This way, students already get the hang of reusing components. At first they can view the components as a *black box*, they don't have to worry about what's inside. This pedagogical approach supports the idea behind object technology, being a way of starting to think *top-down*, first seeing the global picture and worry later about implementation details. This deals with the problem addressed in the previous section, that we should teach *what we need to learn*. With this technique, students can already start building meaningful applications early in the learning process; they just put together parts of the library in some way to accomplish their goal. The next step is then to make students curious about what's inside those components. In [Mey93] this is referred to as *regressively opening of the black boxes* approach.

The above discussed problems and questions present a global picture of what the object-oriented approach is all about and what the underlying principle

really is. The following section will give an overview of the existing pedagogical techniques of how the concepts of object technology like objects, classes, inheritance and polymorphism are being taught to learners.

2.3 Existing Approaches

Because there has always been a need to adjust curricula based on the needs of the industry or some new trends evolving, the increasing popularity of object technology made a lot of computer science departments think about their core programming courses. As a result, a lot of experimental courses followed which varied in languages, environments and tools.

In this section we will discuss some of today's existing approaches used for teaching object orientation. We will not discuss techniques used for acquiring better programming practices based on heuristics [GLH96, Mic98].

The approaches presented follow two main directions, being language dependent and independent approaches. We will start by presenting the latter first. Then we will conclude this chapter with the approach we follow to prove our thesis.

2.3.1 Language-Independent

Some researchers strongly believe that learning object-oriented concepts is all about a way of thinking and students shouldn't have to learn syntax before understanding those concepts. This idea can be supported by choosing a language-independent approach for teaching.

CRC-Cards According to what was said in the previous section on *what* we should teach and *when* we should identify an introductory learning stage, being a stage in which you confront a novice learner with objects. But students often find it difficult to find good objects when they first start. A modeling technique, called *CRC-cards* [Bec89, WBWW90] is used to let students think this way. It is based on three concepts:

- **Classes**,
- **Responsibilities**,
- **Collaborations (CRC)**.

First different classes have to be identified by students and also the responsibilities they will have in the required application. The next step is then to describe how and with which objects they have to collaborate. A card is then physically made per class and all the information is written on that card. This approach has already proven useful [Bec89] for creating the right

way of thinking. It will already lead to good design, but it is not complete with regard to implementation, since it is only a modeling technique.

The Demeter System Another language independent approach for learning about object technology is presented in [LR89]. They use a Meta-Tool, called *Demeter System* and learning is done by gradually confronting the learner with objectives.

The first knowledge that the student must develop is a conceptual meaning for a class, an object and the relationships between these entities. The system uses class dictionaries to do so. A class description language is used that intends to make the task of defining classes an easier design step and then code is automatically generated in the object-oriented programming language of your choice. Once the use of classes is well understood, the next steps they focus on is writing methods, generic programming and multiple inheritance. Advantages of this language independent approach is that first, by using objectives, it provides a metric by which the user can measure his/her progress. And second, this method provides a facility through which users can begin their studies at the level of their experience. A deeper explanation of this system can be found in [LR89].

Learning Environments Learning Environments can also be used as language-independent approach. They can also be used to combine them with language-based approaches. They are built to cover up deficiencies of using a language as a teaching tool and they try to *filter* out and *emphasize* those concepts that we *need to learn*. With *filtering* we mean taking out the obstacles for learners like language syntax. And they *Emphasize* on the important issues of what should be taught by for example visualizing them.

LearningWorks is such a learning environment. It is based on Microworlds. An object-oriented Microworld [LPR94, AGMPR95] is a computer-based representation of some part of the real world built around objects and classes. We will not elaborate on this approach here since we will discuss this environment in detail in the following chapter.

Cognitive apprenticeship approaches for education purposes are also used. In [CTC93], an approach is presented for teaching Smalltalk programming in a Computer-based Learning Environment (Smalltalker), but we will not discuss this here.

2.3.2 Language-Dependent

These approaches of learning are based on describing the concepts to learn by using an object-oriented language. Since the problems with existing languages presented in sections 2.1.1 and 2.1.2 cannot be ignored by any

expert, some ways are found to overcome some of those disadvantages. A solution that a lot of education scientists would agree on is designing an object-oriented programming language specific for teaching purposes.

A Special Teaching Language In [KR99] the *Blue* teaching language is presented. It is specifically designed and developed for teaching purposes and it is based on a set of principles reflecting the requirements discussed in sections 2.1 and 2.2.

We refer to [KR99], we will not discuss this language here.

Libraries Another language-dependent approach often followed is using as a teaching language an existing language that has its deficiencies, but hiding those deficiencies as much as possible by means of learning libraries. For example, In C++ for reading an integer from the screen you need to use the `scanf()` function, which is difficult to explain to a novice learner. This can be covered by defining an operation `getInteger()` that is easier to use and explain.

2.4 Our Teaching Approach

In this thesis we will show how to teach object technology by using the Learning Environment LearningWorks as presented above. We think it is important to *think* in an object-oriented way before a learner has to deal with the syntax of a programming language. So our teaching approach will be top-down to teach novice learners how system building should be done, in particular what the principles of object-technology are.

We will provide something extra though. We will add some intelligence to this learning environment by recording all the user's activities and put them into a knowledge base. We will use this knowledge to see how the learning curve of the student is evolving and we will develop a system that also acts as a *mentor*.

We will guide the student through a *learning path*. This learning path represents a path of concepts of object-oriented programming that the learner has to traverse in order to understand the basic features of object technology.

We will then use the knowledge of the learner to see if he mastered the learning topic of each node in the path and according to the learning level the student will be advised what to do next, i.e. making some extra exercises or proceeding to the next topic (a node in the learning path). This way we have a guided way of introducing a learner to new topic of the learning subject.

The following chapter will give more information about learning environments, and will introduce LearningWorks, the learning environment we will

use as a case study later on. We will also talk about Intelligent Tutoring Systems (ITS) and their main components.

Chapter 3

Learning Environments

During the last decade, computer-based learning has become a promising and interesting new area of application of computer science and computer technology. Several factors have contributed to the developments in this area. First of all, advances in electronics resulted in faster hardware and consequently much faster computers. This allowed for computational models for computer-based education, that were already developed but until then lacked the necessary computing power. Advances in cognitive¹ science were also an important factor; it moved thinking beyond the limits of the psychology of human behavior.

Besides these contributions to the development of learning environments, these environments themselves are contributing to new types of learning and to new approaches of instruction. This is the case because computer-based learning systems use technology which is completely changing the way classrooms exist in our educational systems, and the whole concept of a *teaching class* must be rediscovered again because of those changes. Since our goal is to promote learning and to improve the development of our learners it seems that learning environments stimulate us to think about those issues we *have* to focus on.

This chapter will present the main ideas about learning environments, and more specifically about the basic principles of Intelligent Tutoring Systems. Further on we will apply these principles to the learning environment we used, LearningWorks.

¹Cognitive means relating to the mental process involved in knowing, learning, and understanding things; a formal or technical word [Cob95]

3.1 Introduction

Computers have been used in education for several years now. Computer-based training (CBT) and computer-aided instruction (CAI) were the first systems developed to be used for teaching purposes:

- *Computer-Based Training* ² provides education in the form of an interactive training tool that allows you to learn a specific subject at your own PC. Information is delivered to your screen, questions are asked, and you receive feedback of the system on your responses. You navigate your way through the training process in a graphically enhanced environment finishing with a test that measures how much you've learned.
- *Computer-Aided Instruction* can be defined ³ as computer technologies that assist education, including guided practical exercises, visualization of complex objects, and computer-facilitated communication. CAI can increase access to information because it has some advantages over other types of instruction like interactive response, immediate feedback, infinite patience, animation, motivation, and the ability to keep accurate records of the progress of a user. With the advance of information technology, CAI is increasingly popular for providing health care information on the Internet.

The promise of computer-aided instruction has always been individualized instruction: providing a learner with an environment that is adjusted to his/her own learning needs and goals. However, those first systems based on CBT or on CAI could not take into account the needs of each individual learner. The architecture of CAI systems was found inadequate to provide robust and rich learning environments.

Therefore research was done in directions of Intelligent Systems, called ICAI. Such a system would at least need an explicit model of the domain and a model of an expert that can solve problems posed in that domain; also a student model and a pedagogical model [CS90]. This research together with cognitive science development gave this field of research its most common name, *Intelligent Tutoring systems*. In section 3.4 we will talk about these systems more in detail.

3.2 Why use a Computer-Based Learning Environment?

During the last two decades, computer-based learning environments (CBLE) became an emerging and challenging new area of application of computer

²<http://www.indiana.edu/~ucsdcas/cbt/whatis.html>

³<http://parsons.umaryland.edu/~kvolr001/paper.html#a3>

science and technology, and a very promising area of exploration for educational sciences. One of the reasons why it is becoming so popular is because you can provide teaching material that can be delivered with the computer itself as a teacher. But what about the quality of the education?

There are several reasons why educators make use of a computer-based learning environment. One of those is that learning environments make it possible to make education pleasant. Difficult concepts can be learned by hiding them behind visualized and animated displays, by using sound effects, so people have fun by learning new things. Because of the possible hardware facilities these days this is very realistic to pursue.

Educators also believe that learning environments provide a medium to enhance a student's general problem-solving ability [Lar97]. It seems that learning environments can provide a higher abstraction level of learning to a student, meaning that the student doesn't learn a specific task, but he develops an ability to easily learn that specific task when he's asked to do so.

Another reason why we would use CBLE's in education is expressed in [Gol93]. In this work, an experiment pointed out that computer teachers are better organized and more consistent. The students of the experiment claimed that the computer courses contained - finally - courses that started with clear statements of goals and objectives and followed through with material that was consistent with these introductory statements.

The next section discusses the learning environment LearningWorks more in detail. We will not discuss other learning environments here. However, in section 3.4 we will discuss existing intelligent tutoring systems.

3.3 A Learning Environment: LearningWorks

LearningWorks ⁴ is a learning environment in which learners can explore ideas about software systems architectures. The exploration is done making use of a programming language that supports dynamic object modeling and libraries of selected objects. The language used until now is Smalltalk, but any language that supports object modeling with dynamic binding would suffice. It is both a *toolset* in which the curriculum can be accessed and explored, and a specific curriculum about system building. The learning environment supports the ability to create, access and explore learning courses that emphasize individual, as well as group software construction activities.

The basic tools for LearningWorks are intended for novices who will either study on their own or with a group in a classroom. Figure 3.1 shows the opening window of the LearningWorks environment. It is called the course binder because a course is displayed and the available books for that

⁴<http://learningworks.neometron.com/>

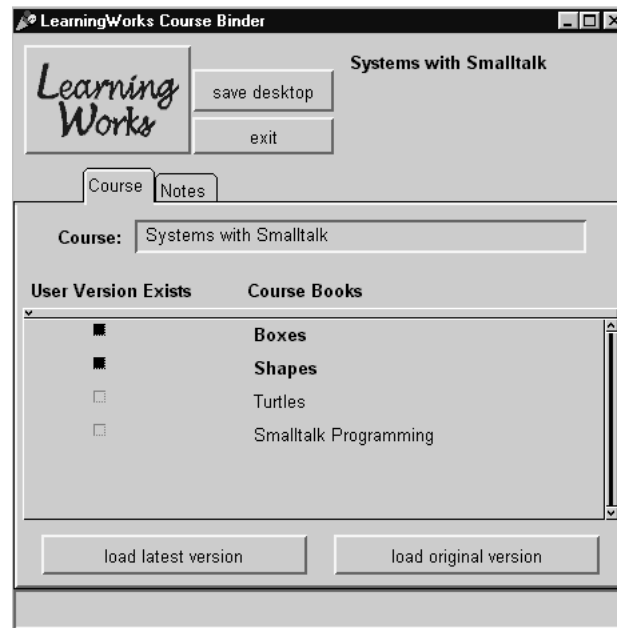


Figure 3.1: The LearningWorks CourseBinder

course are displayed there. For opening a book you select it first and then you click to load a personal version (if you worked with that book before and you saved your work) or the original one. The *boxes* book and the *shapes* book have already saved version, because the check box is marked.

3.3.1 Structure

We will first explain the structure of LearningWorks, to make clear how this environment will teach the principles of system building, and in particular the concepts of object-oriented programming.

Learning Books

All information and activities of LearningWorks are accessible from a *learning book*. You go through or *read* a book by selecting sections from it and in a section you select a particular page. Figure 3.2 shows an example of a page in LearningWorks.

A page in a book contains activities or applications that you interact with to explore various topics of a course of study. A set of pages can represent a simulated world that the learner explores as a way to understand basic concepts and techniques. We call these *rehearsal worlds*, because they provide a context in which learners can practice what they learned in their curriculum so far.

By selecting and loading the boxes book, you get the window as shown in

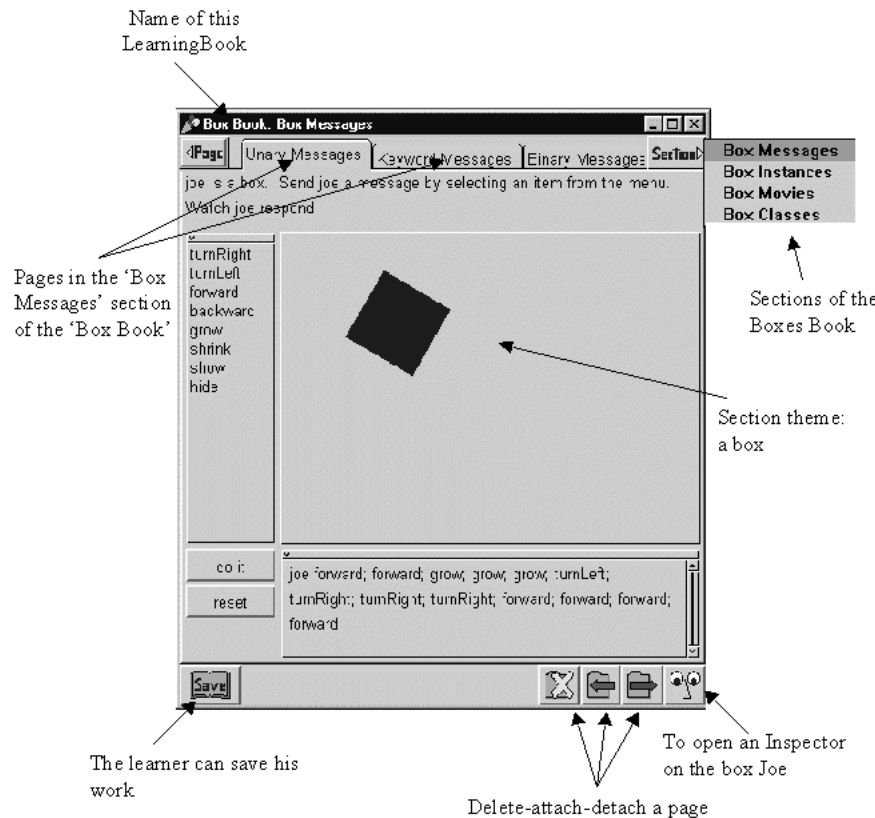


Figure 3.2: Example of a Learning Book Page

figure 3.2. You access the different sections of a book by selecting a section tab and then selecting the section you want to explore. Pages are labeled and by clicking on the label you can access the different pages for that section. An interesting feature of LearningWorks is that its book structure allows us to define name scope by something visible. We can define names within the context of a page by allowing the declaration of local page variables. This creates the possibility to extend or to close off access to definitions and you can also control what aspects of the language and library are accessible to the learner.

Pages

Three different kinds of pages are available in the environment: *tool* pages, *specification* pages or *activity* pages. The tool pages provide support for defining, browsing and inspecting the structure of Smalltalk objects. An inspector page is an example of a such a page, which is shown when in figure 3.2 you select the inspector button.

The information pages, are either specification or activity pages. A specifi-

cation page defines which objects will be available in the context of using a particular learning book. They are used to define software to be imported and made visible for the learner, to define software created specifically for use within the current book, and to define criteria for succesful completion of book activities. An activity page contains the rehearsal worlds.

Section Themes

Pages within a section can share a theme which displays as a side view that is the same for all the pages of that section, and also its state is remembered across the different pages of that section.

These themes are important for realizing our goal of simplifying access to aspects of software information, because the theme provides a context that can be shared among pages.

The Authoring Tool

One of the most important problems regarding computer-based learning environments is the cost of preparing a curriculum. Environments for teaching have been proven to be successful in the past, but it almost doesn't weigh up to the cost problem. At one point, it was estimated that it takes almost 25 hours of efforts for as many minutes of instructional interaction.

Newly developed learning environments try to overcome this problem by providing facilities to make your own curriculum. The LearningWorks toolset is designed together with an *authoring tool* so that others can create their own curriculum or examples. Their approach to creating the structure of a LearningBook is to give you a structure editor whose semantics is linked to that of the Book Definition Language (BDL) ⁵.

3.3.2 The Curriculum

After mentioning the learning environment itself, we will formulate what this environment will focus on. In section 2.2 we explained the general approaches of teaching and we pointed out some questions with regard to those approaches. We also mentioned what approach LearningWorks uses. Here we will go a little bit more into detail.

They want to teach novice learners the four essential aspects of software construction : systems, components, architecture and frameworks. They present the following definitions [Gol95] of these aspects:

- a *system* is a set of communicating parts or components that fit together according to some well-founded semantics for the purpose of a system,

⁵<http://learningworks.neometron.com/doc/authgde/bdlref.htm>

- A part or *component* is understood to be a systems building block that is well-defined by its external interface,
- An *architecture* is a specification of a set of components and a communications pattern or protocol among them to achieve certain behavior.
- a *framework* is a customizable system.

3.3.3 Implementation

LearningWorks is implemented as the composition of four frameworks [Gol97]: a LearningBook presentation and Interaction framework, hereby supporting the basic user model, a programming framework with access to a library of reusable software components, an authoring framework for creating LearningBooks and a team communications framework.

The first framework mentioned here captures the structure of LearningWorks as described above. The Programming framework provides those features that deal with the programming language LearningWorks is based on, being Smalltalk. Pages of a learningBook can contain tools for developing Smalltalk object definitions.

The next section will introduce the main ideas behind Intelligent Learning Environments. The main idea is that knowledge and reasoning that are used for CBLE's systems execution, it should be modeled in an explicit way by some specialized models.

3.4 Intelligent Tutoring Systems

Despite a lot of interest during the last thirty years in the development of intelligent tutoring systems, which were hoped to be as good and as effective as a human tutor, only very few systems are in practical use today. An intelligent learning system is a system that provides individualized and dedicated teaching, adapting the teaching rules to the students' particular abilities [WW98].

Two aspects must be dealt with for having an intelligent system that teaches effectively: interaction and intelligence. The system must *interact* with the learner to reinforce the content and it must be *intelligent* so it can adapt its teaching strategy out of the students success or failure [WW98].

In this section, we will provide an overview of the main components of intelligent tutoring systems (ITS). We will discuss the main problems that those systems have to deal with. We will also take a look at the existing ITS's.

3.4.1 Components of an Intelligent Tutoring System

Intelligent tutoring systems may appear to be systems that are built rather case-specific, because existing intelligent learning systems are all presented as learning systems for a specific application domain. However, we can distinguish some important components that occur in ITS's. Previous research [Bru95, BPLR91, WB92] has identified 4 major components:

- the domain knowledge,
- the student model,
- the pedagogical model,
- the communication or interaction model.

Some [BSH98] refer also to a fifth component, being the *expert model*. This knowledge could also be seen as part of the domain knowledge. Figure 3.3 provides a view of the interactions between the modules.

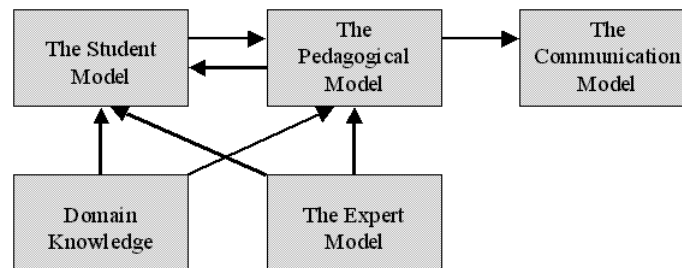


Figure 3.3: Interaction of components in an Intelligent Tutoring System

Domain Knowledge

This component contains information about *what* the tutor is teaching. It is undoubtedly one of the most important models, because without this information there would be nothing to teach. In general it requires significant knowledge engineering to represent a domain so that other parts of the tutor can access it.

The Student Model

The student model stores information that is specific to each individual learner. At least, such a model should track how well a student is performing on the material being taught. A possible addition to this is to also record misconceptions, but this is hard to describe. Since the purpose of the student model is to provide data for the pedagogical model of the system, all of the information gathered should be able to be used by the teacher of the course.

The Pedagogical Model

This component provides a model of the teaching process. For example, information about when to review, when to present a new topic, and which topic to present is controlled by this model. As mentioned earlier, the student model is used as input to this component, so the pedagogical decisions reflect the differing needs of each student. One pedagogical concern for an Intelligent Learning System is the selection of a meta-strategy for teaching the domain.

The Communications Model

Any form of interactions with the learner, including dialogues or windows where advice can be displayed are controlled by this component. How should the material be presented to the student in the most effective way? This part can be seen as the *advice* that results from the intelligent part of the system, how it is transferred to the student. Any kind of knowledge that deals with how this transfer is done is placed in this model.

The Expert Model

The expert model is somehow similar to the domain model in that it also must contain the information being taught to the learner. It should represent all knowledge a student should learn in the end to master completely the subject being taught. It must be a runnable model, one that is capable of solving all problems of that specific domain. By using an expert model, the tutor can compare the learner's solution to an expert's solution, pointing out the places where the learner had difficulties.

3.4.2 Existing Intelligent Learning Systems

We will show some existing learning systems here. Most of the existing systems have one thing in common: they are very case specific. We found lots of intelligent tutors for teaching programming languages like for example LISP and Pascal.

The Domain of Computer Programming Computer Programming is an interesting domain for research on intelligent tutoring systems. It is constrained enough to be traceable, but it is still interesting to present a real modeling challenge. If a domain is traceable, this means that the representation of domain knowledge can be seen as a set of problem solving rules, which is a frequently used strategy for ITS's. Computer programming is also a rich domain, because there are a variety of correct answers for a programming problem. A functional model of computer programming, including goals, problem states and problem solving operators can be specified precisely in

computer simulations that reflect the reasoning of human subjects.

In this section we will present one of those intelligent tutors for programming, named LISPITS. We will also mention one more general system that presents an architecture for a rapidly prototyped intelligent tutoring system, and it validates the architecture by building a tutor to teach Pascal loops to novices.

A LISP Tutor: LISPITS

In [CA92] they introduce LISPITS. This ITS is an intelligent tutor for LISP programming. The tutor provides instruction in the context of problem solving by monitoring the solution of the student and providing feedback when the student needs guidance or demonstrates a misconception. LISPITS uses the technique of model tracing to *understand* the student's reasoning as each step is taken. The tutor analyzes each step of the learner's solution to determine whether it is on the path toward a solution or indicates a misconception. Since LISP syntax is very consistent in that each LISP expression consists of a function call composed of a function and its arguments, this tracing of a possible solution or not is not very hard to do.

RAPITS

In [WW98] an architecture for a rapidly prototyped intelligent tutoring system, called RAPITS, is presented. They claim that their system teaches most effectively for teaching hierarchically organized procedural knowledge. The system consists of a set of lesson and assessment templates which are used to generate pages in an electronic book. The intelligence is driven by a meta-strategy entered by a teacher, which determines the next lesson, based on the student model. It has been evaluated by a group of students and it was found to be easy to use and to significantly improve learning. This approach supports also the authoring of learning courses.

Design of a lesson Just to have an idea of how this tutor works we will describe how a lesson looks like. A lesson is based on teaching-by-analogy to demonstrates examples. The lesson test involves choosing the answer out of multiple choice possibilities. So the pattern for a lesson *page* is to first teach a rule, then demonstrate it by example and in the end testing the student.

3.5 Conclusion

Although it is hard to show that today's learning environments make students learn faster and better, they have been shown to be very effective in increasing student motivation and learning itself.

In designing these systems, it is best to see them as architectures that need to have at least four components: the student model, the pedagogical model, the domain knowledge and the communication model. A fifth component, an expert model, is considered optional.

For this work we will use the learning environment LearningWorks as part of the architecture of an intelligent learning system. We will present our general architecture in chapter 5; it is based on the main components described here for Intelligent Tutoring Systems, but we will not consider an expert model because of the given time constraints for this work. That system will then be used for teaching object technology in an intelligent way.

The next chapter talks about the medium we will use to represent our knowledge with the main system components in the logic meta-language SOUL.

Chapter 4

The Smalltalk Open Unification Language (SOUL)

In this chapter we will give a general overview of the logic meta-language SOUL. It will be used in the next chapters for representing the knowledge for our tutoring system . Since its semantics rely on the nature of logic programming, we will start by giving a quick overview of logic programming in general. After presenting SOUL, some examples will be used that demonstrates the usefulness of this language.

4.1 Logic Programming

Logic Programming is the name of a programming paradigm which was developed in the seventies. Rather than viewing a computer program as a step-by-step description of an algorithm, a program is built as a logical theory, and a procedure call is viewed as a theorem of which the truth needs to be established. Thus, the execution of a program is in fact the search for a proof.

Logic Programming uses a collection of facts and rules in a database. Facts are used to hold static information that is always true in the application domain. Rules are used to derive new facts from existing ones. The body of a rule specifies under which conditions a new fact, with as 'name' the head of the rule, can be concluded. Queries are then used to access this data.

Logic programming languages are declarative languages, because programs consist only of data declarations, and lack the assignments and control flow statements that are needed in imperative languages. They are thus non-procedural, i.e. the characteristics of the solutions are given, but the complete process of getting the solution is not. Logic programs are compositions of statements, or propositions in symbolic logic. They use one general con-

trol structure, logic inference.

The syntax of logic programming languages is remarkably different from that of the imperative languages. The semantics also has little in common with that of imperative language programs. A more detailed description of the nature of logic programming can be found in [Fla94].

4.1.1 Characteristics

One of the essential characteristics of logic programming languages is their semantics, which is called *declarative semantics*. The basic concept of this semantics is that there is a simple way to determine the meaning of each statement. It does not depend on how the statement might be used to solve a problem. Therefore, declarative semantics are much easier than semantics of imperative languages. For example, the meaning of a given proposition in a logic programming language can be concisely determined from the statement itself. In an imperative language the semantics of a simple assignment statement requires examination of local declarations and knowledge of the scoping rules of the language just to determine the types of the variables in the assignment statement [Seb96].

A very important property of declarative languages is that they are very open, i.e. everybody can easily add or delete information used in the unification process by manipulating the database of facts and rules. Logic Programming Languages are therefore easier to change.

The major drawback of logic programming is its sometimes slow execution time, depending on the query that needs to be solved. In non-procedural programming, where you don't specify *how* something is accomplished, take for instance sorting a list, you can be dealing with a very slow process. Also for efficiency reasons and even to avoid infinite loops, logic programmers must sometimes state control flow information in their programs.

4.1.2 A Comparison with Imperative Programming

Logic programming is much closer to mathematical intuition than imperative programming:

- Take for instance the concept of a programming variable. In imperative languages, a variable is the name for a memory location which can store different types of data. While the contents may vary, the variable always points to the same location. On the other hand, a variable in a logic program is a *placeholder* that can take on any value, just like in a mathematical formulae.
- logic programming is multi-way, i.e. one predicate expresses multiple relationships.
Example: take the mathematical equation $A + B = C$. In a logic

programming language we would create the following predicate `add(A,B,C)`. However, this predicate is able to perform 3 different tasks:

1. adding two numbers,
2. subtracting two numbers, ($B = C - A$)
3. checking whether the equation $A + B = C$ holds for given A , B and C .

Imperative programming and Logic Programming also differ in the *machine model* they assume. A machine model is an abstraction of the computer on which programs are executed. Imperative programming languages assume a dynamic, state-based machine model, where the state of the computer is given by the contents of its memory. The effect of a program statement is then a change from one state to another.

In Logic Programming, computer and program represent a certain amount of knowledge about a world, which is used to answer queries [Fla94]. Therefore Logic Languages are especially suited for representing knowledge about almost anything. How we will use it within the scope of this thesis will become clear in the next chapters.

4.2 Logic Meta-Programming

To avoid confusion we want to make clear what we understand under the terms *meta programming*, *meta program*, *base program*, etc. Therefore this section starts by introducing some terminology. Afterwards, we will present the general idea behind Logic Meta Programming.

4.2.1 Terminology

A *program* is a specification of a computational system that manipulates representations of entities from some "universe of discourse". A program is expressed in a formalism that can be interpreted automatically in order to obtain the computational system it specifies. This formalism is called a *programming language*.

Programs can be constructed to reason about almost anything imaginable. It just boils down to defining representations of the entities or concepts one wants the program to reason about in terms of the data structures that are built into the programming language. Consequently, programs can be constructed to reason about other programs. Examples of such programs are compilers, type checkers, interpreters, code generators, etc.

A program, the "universe of discourse" of which contains programs, is called a *meta-program* or a *meta-level program*. The programs in the universe of discourse are called *base programs* or *base-level programs* [DV98].

4.2.2 A logic meta-programming system

The main idea of logic meta-programming is describing aspects of base-language programs by means of logic programs. The central concept around which everything revolves is a *mapping* which associates certain aspects of the base-language with a set of logic propositions. As we stated earlier in this chapter, a logic program is a sophisticated way to specify a set of logic propositions. Therefore it follows that logic programs can be used to specify aspects of base-language programs indirectly.

Within this dissertation we will work with a logic meta-programming system that has as base language Smalltalk and as meta language SOUL. The link between the two levels is that in SOUL you can use terms in the logic rules that are Smalltalkblocks, meaning that you can put Smalltalk code in the predicates. This approach is followed by SOUL, that allows to specify logic programs over Smalltalk

4.3 An Overview of SOUL

SOUL (Smalltalk Open Unification Language) is a logic programming language written in Smalltalk [Lal94, LP90]. It is based on PROLOG¹, but it has an extension that allows unification of user-defined elements expressed in Smalltalk [Wuy98]. This makes capturing any Smalltalk language concept in logic rules possible, without explicitly formulating this information into facts and rules.

4.3.1 Comparing SOUL with PROLOG

For comparing SOUL to Prolog, suppose that we want to write a logic program that reasons about classes and methods. In Prolog this means that we first of all have to create a database containing the information we want to reason about. Since we are interested in classes in methods, we need to fill the database with facts such as:

```
class('Object').
class('Collection').
....
```

and

```
method('Object', 'size').
method('Collection', 'add:').
...
```

¹PROLOG is the most widely used logic programming language

We have to do this for *EACH* class *AND* for *EACH* method in the system. In SOUL, you use the *generate* predicate to bind a variable to all classes in the system, but:

- it is a lot more *efficient*,
- you use two statements instead of thousands of statements,
- you have an *automatical update*, when a class is being added or removed, you don't have to add or remove facts.

In the next section, we will give some introductory examples to show how a SOUL program looks like.

4.3.2 Some Examples

Before giving any examples, first a note about basic SOUL syntax: logic variables are denoted with question marks, the comma is used for the Boolean *and*, and terms between square brackets are Smalltalk terms, terms that contain Smalltalk expressions which can refer to logic variables.

As a first example, we introduce the *class* predicate we talked about before. It consists of two rules:

```
Rule class(?Class) if
    constant(?Class),
    [Smalltalk includes: ?Class name].
```

```
Rule class(?Class) if
    variable(?Class),
    generate(?Class,[Smalltalk allClasses])
```

The first rule describes what happens if a query is launched with as head the **Class** predicate and the variable *?Class* is bound to a value (so if it is a constant). Then the Smalltalk term checks whether or not the constant *?Class* is included in Smalltalk.

The second rule is applied if *?Class* is just a variable. Then the *generate* predicate is used to bind that variable to all the classes present in the Smalltalk image you are running SOUL with by compiling the code between the square brackets.

With these two rules defined, we can ask SOUL to list all its classes or we can ask whether or not Array is a class:

```
Query class(?C)
```

```
Query class([Array])
```

Here is another example to demonstrate the usefulness of SOUL. This rule uses a predefined predicate *hierarchy* to look for a common superclass of 2 classes:

```
Rule commonSuperClass(?Class1,?Class2,?CommonClass) if
    hierarchy(?CommonClass,?Class1),
    hierarchy(?CommonClass,?Class2)
```

The predefined predicate *hierarchy* used here searches for *?CommonClass* for all the subclasses, i.e. all classes that belong to that class's hierarchy. Notice that the power of logic programming is reflected here: you can either use this predicate to find a common superclass or, given a superclass, you can find all pairs of classes that have that class as a common superclass.

4.3.3 The SOUL Declarative Framework

SOUL has a built-in framework of rules and facts that allow reasoning about Smalltalk code. This framework consists of two layers: the core rules and the basic structural rules. The core rules capture the structure of the base language, such as *class*, *superclass*, *method*, *methodSelector*, *instVars*, ... The second layer of rules uses the core layer rules. Rules like *hierarchy*, *isReceiver*, *isSendTo*, ... form this second layer. A detailed version of the SOUL framework can be found in [Wuy98].

4.3.4 The SOUL System Repository

Just as any other logic language, SOUL uses a database to store all gathered knowledge. The whole declarative framework is stored in the SOUL system repository.

An important extension for the context of this work has been made. A class *SOULCompositeRepository* has been created to be able to work with repositories inside the system repository. This is very useful when there is a need to structure the represented knowledge into *models* like in a tutoring system. This will be explained in detail in chapter 5.

4.3.5 Other Application Domains

SOUL has proven to be very useful for various application domains. Using the above defined declarative framework it is possible to use SOUL for several sophisticated development tools. We will shortly present a few of them in this section.

Advanced Structured Searching Object-Oriented systems are a very complicated and tangled web of interdependent classes. In order to be able

to extract the structure, people who want to get a closer look at those systems really need to dive into the code. So advanced structural search would make a major contribution.

Standard Smalltalk Code Browsers offer find tools like *senders* and *implementors* that are quite useful, but too general. Sometimes we need to zoom in on a certain part of a program. Our developed declarative framework makes this possible by combining primitive find tools.

Let us show two examples:

- You want to find classes that implement a method A *and* a method B. This is not possible in a current code browser, but using SOUL we could make the following query:

```
QUERY implementors([#terms],?Impl),
      implementors([#allUnifiers:Repository],?Impl).
```

All classes that implement `#terms` and `#allUnifiers:Repository` are then returned.

- Suppose we want to find the sender of a frequently occurring method (i.e. a method that has a lot of senders), but we know the one we're looking for lies within a hierarchy of classes (our developing domain). What we would need is a way to minimize the search space. Consider the following query in SOUL:

```
QUERY findAll(?SubCl,hierarchy([LwNode],?SubCl),?Lst),
      member(?SenderClass,?Lst),
      senders([#new],?SenderClass,?SenderMethod).
```

This query returns all sender classes and sender methods of the method `#new` within all LearningWorks nodes (books, pages, sections,...). If you perform a search on all senders of `#new` (this is the only thing you can do in today's browsers), you get a very long list, so the sender you're looking for is still out of your reach!

Programming Style Rules Programming style rules define a programming style that is used throughout an object-oriented system. A good example of a programming convention could be to *never access instance variables directly, always define accessor methods to access them*. A former version of SOUL was used [Mic98] to express programming conventions. A declarative framework was developed in LPS with several layers. The first layer included the core rules that express the structure of the base language Smalltalk, like classes, methods and protocols. The outermost layer used the core rules them to express conventions about programming style rules in general and specific rules for Smalltalk programming.

Design Pattern Structure Rules Another application of SOUL is expressing structures such as those described by Design Patterns [GHJV95]. Design Patterns capture solutions to common design problems encountered by designing software. They represent recorded experience from over several years in designing object-oriented software in a format so that people can use it effectively.

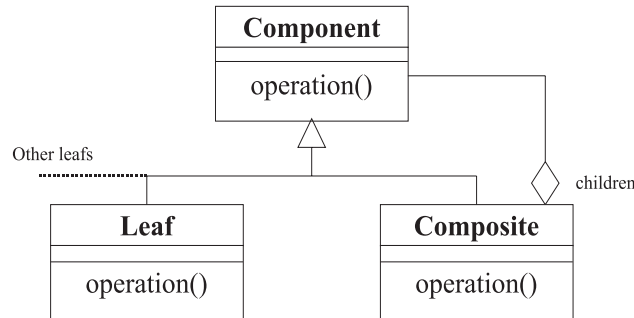


Figure 4.1: Composite Pattern Structure

Since Design Patterns are represented in a template shape, we can detect that shape if it is unambiguously defined. We will demonstrate this by expressing the Composite Pattern (as described in [GHJV95] and as depicted in figure 4.1) in SOUL:

```

Rule compositePattern(?Component,?Composite,?Msg) if
    compositeStructure(?Component,?Composite),
    compositeAggregation(?Component,?Composite,?Msg).

```

```

Rule compositeStructure(?Component,?Composite) if
    class(?Component),
    hierarchy(?Component,?Composite).

```

```

Rule compositeAggregation(?Component,?Composite,?Msg) if
    commonMethods(...),
    ...

```

The first rule says that a composite pattern consists of some kind of structural relationship between a component and a composite component and that there is an aggregation relationship between those two. The second rule expresses that you have a composite structure between the composite and the components if the component is a class and the composite is a subclass (direct or indirect) of the component class.

The aggregation relationship is not so easy to express, but we won't elaborate on this, it is outside the scope of this thesis. Basically you have to

express that the composite should at least override one method of the component, and in this overridden method an enumeration should be done over the instance variables that hold these composites and recursively apply the method to each of the composites.

Codifying Software Architectures In [MW98] SOUL is being used as a formalism to describe software architectures at a sufficiently high level of abstraction. A tool is proposed to automatically verify whether source code of a program conforms to the software architecture of that program. The paper illustrates how SOUL and virtual classifications can be used to elegantly express software architectures in such a way that they can still be checked against the implementation.

The next chapter presents our general architecture for learning about Object Oriented Technology (OOT) in an intelligent environment. Chapter 6 then describes our case study, i.e. applying our architecture for the Learning Environment Learningworks.

Chapter 5

An Architecture for learning OOT in an Intelligent Environment

This chapter presents the general architecture for learning about Object Oriented Technology (OOT) in an intelligent environment. We will make a clear distinction between the learning environment used for teaching object technology and the knowledge base, representing the intelligent part of the system, and used to guide and help the student in any possible way.

In section 5.1 we will present some preliminary ideas we have about our general architecture. This section is divided into two main parts: the system requirements, and the limitations of the presented architecture.

Then we will gradually introduce our general architecture. First, we will handle the main components of the architecture of our Intelligent Learning System. We will explain the role that each of those components will have within the system. We will also show examples of what kind of knowledge these components contain and how they collaborate with each other.

The validation of our work consists of demonstrating this architecture for the learning environment LearningWorks. We presented this environment in section 3.3 and we will add some intelligence to it.

5.1 Preliminaries

This section presents some important preliminaries. We will handle the system requirements first. The limitations we had to face during this development are also important to mention.

5.1.1 Requirements

What aspects are important for making an architecture that provides support to learn object technology? What requirements have to be met in order to have an acceptable intelligent teaching environment?

- we want to focus on teaching the *concepts* of object technology, with as much *guidance* as possible,
- a good *separation of knowledge*, having clearly definable subparts, e.g. models.
- *flexibility*, i.e. it should be easy to plug in/out different components/-parts,
- *reusability* of the system components,

With the first requirement we want to state that it should be possible to reuse the defined models of any learning environment that teaches OOT. If we look at the domain model for example, it should only contain knowledge on concepts of object technology. Of course it uses knowledge of the student model, but when given a clearly defined interface (what the domain model needs from the student model) it should be perfectly possible to plug it into another ITS on teaching object technology.

Our second requirement needs a good strategy for teaching the concepts on OOT, and consequently also provide a good guidance throughout this learning process.

Flexibility is an important demand since it must be possible to extend or modify the models. For example in our pedagogical model we could put extra guidance to see that the student is not mislead by some event. The communication model should also be very open, because the way in which the intelligence is shown to the environment has to be extendible.

The last requirement rather speaks for itself. We do not want to have one large database of rules, because it is not at all reusable, readable and extendible.

5.1.2 Limitations

This research area was too big within the line of this thesis to address and take into account any topic of this domain. We restricted ourselves to the following issues:

- we did not take into account an Expert Model,
- we did not detect object misconceptions (see section 2.1.3),
- we aimed only for novice learners in a university curriculum and not to learners that have to be reskilled,

An expert model is a model that contains perfect knowledge. It means that a learner who is gradually learning is in fact trying to transform his student model into the expert model. A lot of conclusions can be drawn from an expert model, for example you can calculate a *degree* of knowledge the learner has acquired, being the *distance* from the student model to the expert model. Although this can lead to very interesting results, it is not possible to accomplish this within the given time constraints.

Object Misconceptions are very important in the learning process. Since it is very hard to *unlearn* an idea a user can have, it is necessary to detect *wrong ideas* or misconceptions in an early stage of learning. The longer it takes to detect them, the harder it will be to unlearn them. We will give an example of how we can deal with these misconceptions in our case study but because of lack of time we will not fully exploit this interesting topic.

We will also constrain ourselves to teaching object technology to *novices*. If we would consider dealing with any kind of learner, including people that already know a procedural programming language, we would have to take into account the *paradigm shift* (see section 2.2). And this requires a lot more research.

5.2 Our System Architecture

This section will present the architecture we designed for teaching object-oriented technology. It consists of four main parts. As explained in section 3.4 for ITS's, the most important part is the domain knowledge, which represents the material that has to be taught to the learners. There is also the Student Model, the Pedagogical model and the Communications model.

5.2.1 An overview

Figure 5.1 shows a schematic overview of our architecture. The schema shows the connection between a learning environment that teaches OOT and our system architecture. How this connection is realized will be explained in more detail in chapter 6 where we will present our architecture for the learning environment LearningWorks. In this chapter we will focus on the architecture, being the relation between the knowledge models.

The *student model* holds knowledge about the abilities of the learner. It gets its information from the learning environment (how this is done is not important here). The *domain model* on its turn uses the knowledge of the student model to see in what degree the concepts of the domain - being the concepts of OOT like inheritance and polymorphism - were explored and understood by the learner.

The next step is the *pedagogical model*: how can we connect the concepts in our domain? We need a guidance path through all of these concepts to help the learner to learn them in an efficient way?

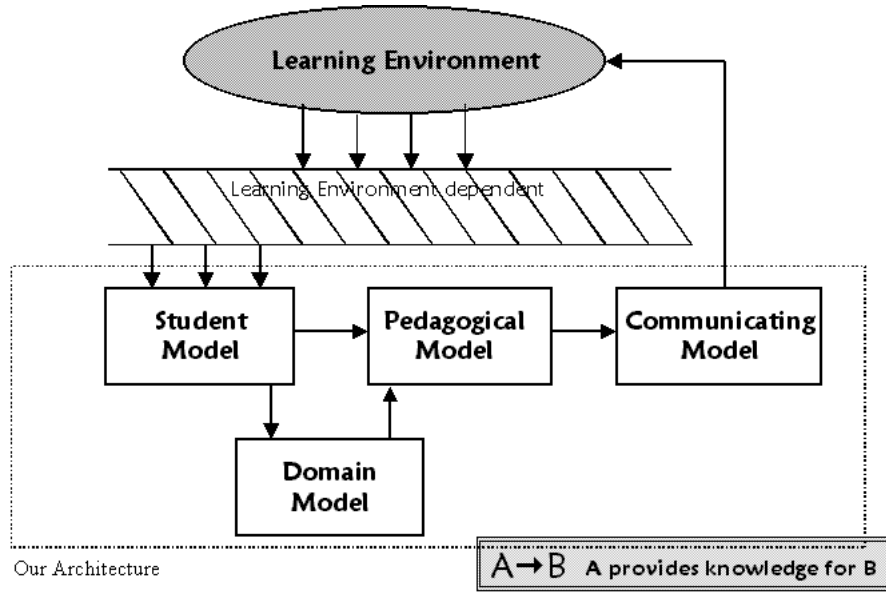


Figure 5.1: Our Architecture for Learning OOT

In what way can we offer advice and guidance to the student? This knowledge will be gathered in our *communication model*, which is important because its output is the only thing our user gets to see.

We will start by introducing the Student Model, because that is where the connection starts between the learning environment and our created architecture.

5.2.2 The Student Model

In existing ITS, as discussed in section 3.4.1, the student model contains some kind of profile of the learner. It is usually extracted out of tests the learner had to undergo in the learning environment. In RAPITS and LISPITS (see section 3.4.2) the same procedure is used for creating a student model.

We will create a student model out of the events we logged. You can view this like some kind of filtering of events. We will demonstrate how this is done in our case study in the next chapter for the learning environment LearningWorks.

Let's show an example of knowledge that is kept in the student model:

```

Rule succeedsInObjectCreation(?aStudent) if
    usedInstanceCreationMethod(?aStudent)

```

We want to express a rule that says whether or not the learner is capable of creating an object. We conclude that he is able to make objects if he

succeeded in using an instance creation method. We will show in our LearningWorks case study what this can mean for a particular environment.

Another example is shown below:

```
Rule capableOfObjectInspection(?aStudent) if
    usedInspectorThoroughly(?aStudent)
```

A student is supposed to be capable of inspecting an object if he already used an inspector a few times ¹.

5.2.3 The Pedagogical Model

The pedagogical model, as explained in chapter 3, contains all knowledge about the *way of teaching* the domain of the learning environment. The teaching approach we will adopt here will be following a *path of OOT concepts*. We will call this path the *learning path*. This will offer more guidance to the learner and this way he will not be confronted with topics above his learning level.

The Learning Path

As said before, this path reflects the kind of guidance we will offer to the student: we will gradually introduce to them all concepts of object technology by starting with the basics and then building further on those basics. This way the learner is not confronted with anything above his level of expertise and he can start at the level he wants too.

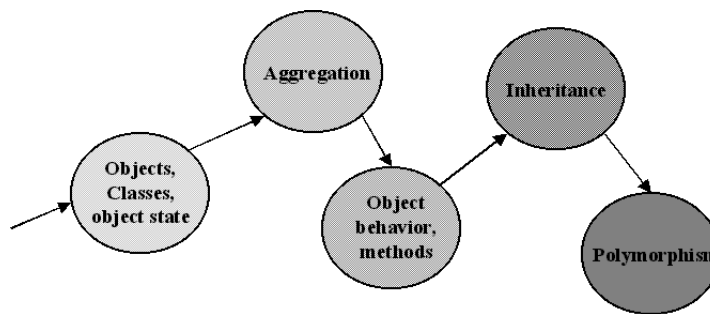


Figure 5.2: OOT Learning Path

Figure 5.2 shows the main concepts that have to be taught to a student. This path is based on a solution for a pedagogical pattern, called *Big Picture on a Small Scale (BPSS)* ², see [SMM⁺96, MSPM98]. It proposes a curriculum for teaching object oriented concepts to novice learners by introducing

¹This is knowledge that is base-language dependent, because it is based on using an object inspector in Smalltalk

²<http://www-lifia.info.unlp.edu.ar/ppp/pp46.htm>

the concepts shown in our learning path, and in that order.

We will put this learning path into our pedagogical model as follows. The first clause is a fact *learningPath* that makes up a list of the object-oriented concepts of our path. Each student should have such a path. The first concept in the list denotes the concept that the student is working on at that time. A full learning path list as presented below means that the learner is working on objects and classes, and he doesn't have enough knowledge (in the student model) to proceed with the *aggregation* concept.

```
Fact learningPath(?aStudent,<[objectsAndClasses],
                             [aggregation],
                             [behaviorAndMethods],
                             [inheritance],
                             [polymorphism]>)
```

The concept to learn is then the first in the learning path.

```
Rule conceptToLearn(?aStudent,?aConcept) if
    learningPath(?aStudent,?path),
    first(?path,?aConcept).
```

The next rule *proceedToNext* checks if the learner is mature enough, meaning that he controls the current learning topic sufficiently, to proceed to the next topic in the learning path. It checks the concept that the learner is doing now and if that same concept is known (this is a rule in the domain knowledge model).

```
Rule proceedToNext(?aStudent) if
    conceptToLearn(?aStudent,?aConcept),
    knowsAbout(?aStudent,?aConcept),
    setToNext(?aStudent)
```

We will first introduce all the concepts of the learning path, which are all part of the domain knowledge. Figure 5.2 shows the main path. These 5 concepts in the figure can however be more fine grained. We will discuss each node in more detail.

5.2.4 The Domain Knowledge Model

Our aim is teaching object technology to novice learners. The domain knowledge model will capture all knowledge about object technology, everything a learner needs to know about the domain.

Our teaching strategy will be built around a learning path of concepts. But what concepts will we focus on and how can we conclude that the novice *controls* the concepts? We need to define those in terms of the student model.

Objects - classes - object state This node of concept provides the basis for a novice learner. In this phase, a learner's mind needs to be set to *thinking in terms of objects*, as explained in chapter 2. This node puts as threshold that the learner should know about *classes*, *instantiation*, *encapsulation* and *information hiding* and *initialization*.

As an example of this knowledge, how will we define instantiation?

```
Rule knowsAbout(?aStudent,[instantiation]) if
    succeedsInObjectCreation(?aStudent).
```

If a learner succeeds in object creation (knowledge of the student model we defined earlier) then we conclude that he knows about instantiation.

It is of course possible to make the preconditions stronger before making this conclusion. This can be changed for example by a teacher who has his own ideas about instantiation, but all within the limits of being possible to detect in the used learning environment where this architecture will be based on.

To master the concept of a *class*, the learner must fulfill 3 conditions. First he must know about instantiation, what we defined above. Second, he must have some experience with exploring multiple instances of one class and third, he must have mixed multiple instances of different classes ³.

```
Rule knowsAbout(?aStudent,[classes]) if
    knowsAbout(?aStudent,[instantiation]),
    exploredSeveralObjectsOfSameClass(?aStudent),
    useInstancesDiffClasses(?aStudent).
```

Aggregation Aggregation refers to a collaboration between classes which is based on an *Is-part-of* relationship [WBWW90]. How can this concept be described declaratively? We could say that the learner should have created a class of his own with some instance variables he communicates with. The communication is the difficult part because we chose to introduce methods after the *aggregation* concept. We want to concentrate also on the communication with the aggregated object for trying to show the difference between the different types of relationships:

- *is-part-of* relationship or *aggregation*,
- *is-a* relationship or *inheritance* (is subclass of),
- *is-acquainted-with*, a collaboration

We will use this rule to express aggregation:

³This architecture is described completely independent of the used learning environment, so we will not mention any knowledge here of *how* these subconditions are defined. In the next chapter we will apply our architecture to LearningWorks

```
Rule knowsAbout(?aStudent,[aggregation]) if
    definedAClassWithInstVar(?aStudent)
```

It should be noted for this example that this is not a very strong expression. We should make it more constrained in the future.

Object behavior - methods When an object receives a message, it performs the requested operation by executing a method.

Definition 5.1 *A method is the step-by-step algorithm executed in response to receiving a message whose name matches the name of the method [WBWW90].*

According to definition 5.1, we should focus on sending messages and see what effect it has on the object. We defined a rule in our Domain Knowledge model like this:

```
Rule knowsAbout(?aStudent,[methods]) if
    methodsBrowsedAndModified(?aStudent),
    trivialMethodDefinitions(?aStudent),
    returnValuesStudied(?aStudent).
```

We conclude that the learner understands how communication between objects is done if in the first place he has browsed a class looking at the different methods and also modified some of them. Trivial method definitions also have to be studied. Note however again that this is hard to do in some programming languages. In Smalltalk it is easy to do, because you have a consistent syntax and everything is an object. For example: let the learner know that a number is also an object where you can send a '+' message to with as argument another number. This is best done by creating an exercise in the learning environment itself.

Return values should also be studied to look at how the object responds when a message is sent to it.

Inheritance Object-oriented programming languages support another abstraction mechanism called inheritance.

Definition 5.2 *Inheritance is the ability of one class to define the behavior and data structure of its instances as a superset of the definition of another class or classes [WBWW90].*

Definition 5.2 can also be formulated that one class is just like another class except that the new class includes something extra.

```
Rule knowsAbout(?aStudent,[Inheritance]) if
    createdSubClass(?aClass),
```

```

usedBehaviorOfSubclass(?aClass,?ListOfUsedMethods),
createdNewBehavior(?aClass,?ListOfNewMethods),
/* Overriding of methods */
modifiesBehavior(?aClass,?ListOfModMethods)

```

To detect this, we thought of letting the learner define a subclass of an already existing class and that he *uses* the behavior of his parent class, defines some behavior of its own and overrides some methods.

Polymorphism Limiting object access to a strictly defined user interface such as the message send allows another use of abstraction known as *polymorphism*.

Definition 5.3 *Polymorphism is the ability of two or more classes of objects to respond to the same message, each in its own way [WBWW90].*

The best thing to do for the learner to understand polymorphism is to send the same message to different objects and see how each objects responds differently. This can be done in different ways and it depends on the learning environment. It will become clear what we mean here in our next chapter on applying our architecture to LearningWorks.

```

Rule knowsAbout(?aStudent,[polymorphism]) if
    sendsSameMessagesToDiffObjects(?aStudent).

```

It is possible that the learning environment provides some extra possibilities to test some concepts we mentioned here. Although it is not possible to put them in our architecture, it would be a good idea to also integrate them if possible.

5.2.5 The Communication Model

The communication model holds the knowledge that presents the results of our knowledge base to the learning environment. This model captures the *output*, the *purpose* of our built system architecture. It is rather important, because this model is the only model that holds information that the student will actually get to see.

Our communication model consists of *guidance* knowledge. We are able to guide the learner through his learning path and give him advice on what to do next. An example:

```

Rule adviseNext(?aStudent) if
    doNotProceed(?aStudent),
    conceptToLearn(?aStudent,?aConcept),
    newline(_),
    write(['You don't master the concept of ']),

```

```

write(?aConcept).
write(['Try making exercise']),
adviseExercise(?aConcept).

```

If we execute the following query in our meta-language SOUL:

```
Query adviseNext([Anabella])
```

then a message is written on a transcript for the learner what he/she best undertakes as a next step. And this is what we wanted to accomplish.

5.3 Remarks

After presenting our architecture for teaching object technology by means of extending a learning environment by adding some intelligence to it, we want to point out some remarks we had during our development process.

First of all, our presented architecture was intended to lay the foundations for showing that learning environments that teach object-oriented technology can benefit significantly from representing declarative knowledge about the learning process itself. Also we showed in this chapter the different models answer to our predefined requirements. But this only shows the foundations, and it is the intention to extend it first before putting it into practice. But we must point out that extending this architecture is very easy to do because of the openness of logic languages we talked about in section 4.1.1.

Another remark handles about the variable `?aStudent` we use in the examples. We used that notation here to express that we were talking about knowledge for a specific student. However, it would be better in practice to load the student model of a particular student whenever that student *logs into* LearningWorks. Then your complete repository is not filled with data on other students. You only need the knowledge of one user at a time, when that particular user is logged in. We refer to section 7.2 in our future work chapter for more information.

The rules we demonstrated in the student model are meant to give an idea of *what* knowledge it contains. It would be better to have facts, i.e. *data* on the student instead of rules that infer something out facts. The reason for using rules instead of facts lies in the fact that we filter events by declaring rules, so if a rule fails, it is considered that the knowledge it expresses is not applicable for that student. It would perhaps be better if we could summarize the events into facts: instead of making a rule *A if B* we would like to have *if B is an event then create a fact A* and put it in the student model.

The next chapter presents a case study. We will apply our system architecture onto the learning environment LearningWorks. We will describe what needs to be done in order to

Chapter 6

A Case Study: LearningWorks

After presenting our general architecture for learning object technology in an intelligent environment, we will apply it on our learning environment LearningWorks (see section 3.3), thereby making it somewhat intelligent. We will describe all the steps needed to apply our architecture on this learning environment.

First, we will briefly discuss the difference with other intelligent learning environments. Afterwards we will give an overview of the complete architecture specified on the learning environment LearningWorks. We will end by formulating some results and we will summarize our conclusions.

6.1 A Comparison with existing Intelligent Learning Systems

In section 3.3 we gave an overview of what LearningWorks looks like. We discussed among other things its education approach, and explained its structure.

One issue that we left out was comparing it with the environments presented in section 3.4.2. We think it is important to stress the difference between LearningWorks and other existing learning environments. One thing they have in common is a hierarchical structure in both systems, but that's where the comparison ends. In contradiction to existing environments, LearningWorks is a *browsing* environment. The learner can *explore* ideas about system building at its own pace. In LISPITS and RAPITS (and in most ITS's) you have a proposed problem, the learner has to solve it and then a test is performed. Based on these tests, a student model is then extracted. LearningWorks is much more open and less constrained, but at the same time guidance is missing.

Another important difference is that this environment is not teaching any

routines or helping in mastering a programming language. It aims at teaching *concepts*, which is a domain that is much harder to capture.

What we will do here to make this environment intelligent requires a slightly different approach than with other ITS's. Since we don't have test results or correction of presented exercises the learner made, we will log all events that occur in LearningWorks and we will draw our conclusions out of those events.

6.2 An overview

Our architecture as presented in figure 5.1 applied on the learning environment LearningWorks is depicted in figure 6.1.

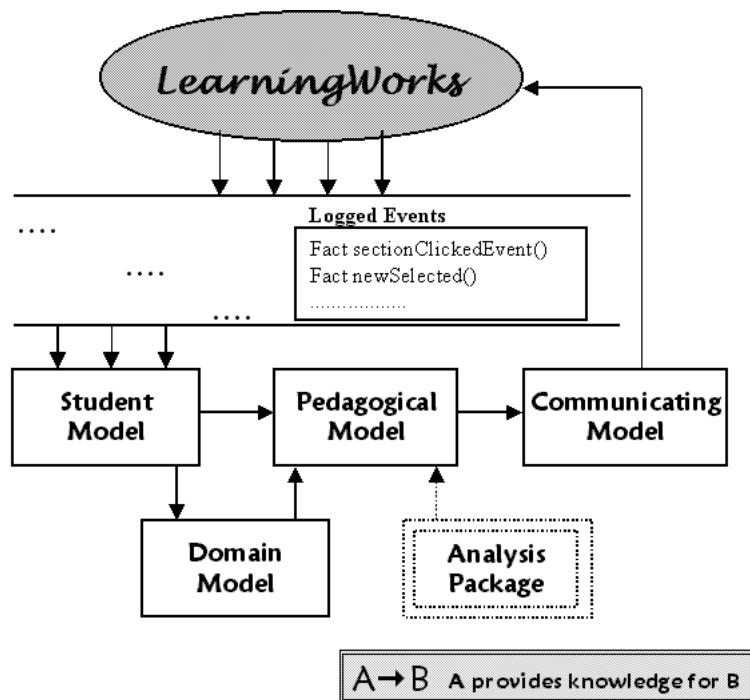


Figure 6.1: Our Architecture and LearningWorks

In the layer that was denoted to be learning environment dependent, LearningWorks events are put. There are also some rules to *summarize* those events and which helped in building the student model that is completely independent of the environment, which was one of the most important requirements of our architecture.

Another new feature is denoted as the *analysis package*. This will hold clauses that will help in analyzing the clauses out of the other modules. These clauses however are completely independent of our architecture. They

help in further analyzing of the acquired data. As an example of what's inside this package, take for instance a rule where you deduce that a learner created a class and some behavior of that class:

```
Fact createdClass(?timeStamp,?aStudent,[Circle])
Fact createdMethod(?timestamp,?aStudent,[Circle],[#create])
```

To analyze further *how* the learner created these items and *what* he created, we have a rule in the analysis package to see in what protocol this method was put:

```
Rule methodInProtocol(?aClass,?aMethod,?aProtocol) if
    generate(?aProtocol,[?aClass organization
        categoryOfElement: ?aMethod)
    == ?aProtocol])
```

Then conclusions can be drawn and questions can be asked to the learner regarding using good protocols and good method names, like always using an *instance creation* protocol for *#new* methods, etc....

This package also offers a way in plugging in easily user defined clauses. A teacher might want to add analysis rules to check *how* the learner programs. What we want to emphasize on here is that in the learning environment LearningWorks there is for example also a book used for teaching introductory Smalltalk programming and you could plug in rules about programming conventions easily, as explained in [Mic98].

Figure 6.1 shows what has to be done. Our goal is to provide the learner with some guidance in his learning process, which can be reached if our communication model is able to create some output that can help the student. We start by logging events that occur in LearningWorks. Afterwards, we try to *summarize* those events into (still) learning environment dependent rules to be able to construct a well-formed independent student model. Then the domain model uses this knowledge to conclude/deduce which concepts are known and which concepts are not known at all. The pedagogical model decides what should happen next and the communications model explains to the student what he should do. We will describe how the event logging is done by means of an example. Afterwards we will show to what feedback for the student this can lead.

6.3 Event logging

Event logging requires some way to create a logic fact when an event occurs, i.e. when a method is executed a fact should be created. In practice, this means using a message passing control technique. We chose here for the fastest technique, i.e. putting logging code directly into the methods.

Figure 6.2 shows a schematic view of how the event logging is done. In the LearningWorks object model, each method of a class that needs to be logged in a fact has an extra piece of code: a message is sent to an event monitor *LwEventManager* to log that this event has occurred. Consequently, the monitor creates a clause and puts the clause into a log repository.

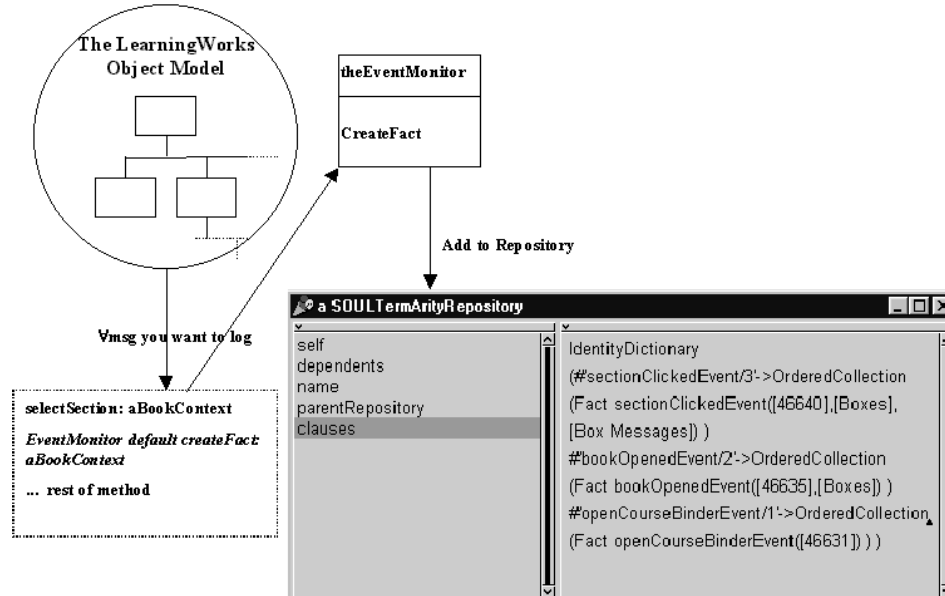


Figure 6.2: Schematic view of the Event Logging

Another technique that could have been used, given some more time, are the MethodWrappers of John Brant¹. Other message passing control techniques can be found in [Duc99]. We realize that our approach is not the best way to go, but it is not important for achieving our goal.

6.4 Construction of a learning environment independent student model

To be able to make a student model independent of the learning environment we need clauses that put a barrier between the learning environment dependent layer depicted in figure 6.1 and the Student Model. Some examples of clauses occurring in this in-between layer (we will take those that we used in earlier introduced examples):

```
Rule usedInstanceCreationMethod(?aStudent) if
    turtlesPressNewEvent(?aTimeStamp),
```

¹<http://st-www.cs.uiuc.edu/~brant/Applications/MethodWrappers.html>

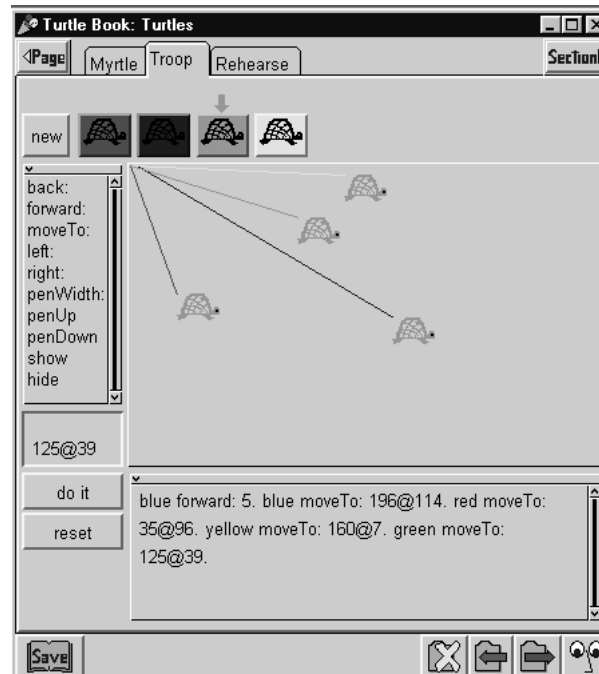


Figure 6.3: Example of the Turtles book in LearningWorks

```

evaluateRehearseEvent(?aTStamp,?aCompiledString),
findNew(?aCompiledString,?index),
not(equal(?index,[0]))

```

```

Rule findNew(?aString,?indexOfNew) if
    generate(?indexOfNew,[(?aString findString:'new'
        startingAt:1) equals: ?indexOfNew])

```

The first rule is used to express if the learner already created an instance of a class. The first event we set as a condition is the `turtlesPressNewEvent`. You have a turtles book with as section theme turtles and one page in a section has a new button to create new turtles. Figure 6.3 shows that page in a section of the turtles book.

The rule `findNew` is a rule from the analysis package we mentioned in the beginning of this chapter. It is used to search for the string `New` that a learner evaluated in a rehearse page. Figure 6.4 shows such a rehearse page. In that page, a learner just evaluated `joe := Box new`.

We can also analyze the compiled method. Lots of information can be retrieved that way, for example the instance variables of the class where the method was compiled in, etc. . . .

Another example of the knowledge used to create our independent student model looks like this:



Figure 6.4: Example of a Rehearse Page in LearningWorks

```

Rule sendsSameMessagesToDiff0bjects(?aStudent) if
  countDifferent(?aShapeName,
    shapeSelectedEvent(?timeStamp,
      [Shape Book],[Messages],
      [Shape Interface],?aShapeName),
    [2]),
  countDifferent(?aTimeSt,
    messageSelectEvent(?aTimeSt,
      [Shape Book],[Messages],
      [Shape Interface],?aMsg),
    [3])

```

This knowledge was used in clauses of the domain model to conclude whether or not the learner knows about polymorphism. The rule `countDifferent` is also in the *analysis package*; it counts the different facts that are equal on all variables except one (the other variable in the clause). So we check that at least 2 different shapes are selected and at least 3 times the same message is sent. Figure 6.5 shows such a page in the shapes book.


```

writeln(?aConcept),
write(['The next concept to learn is ']),
writeln(?newConcept)

```

```

Rule adviseNext(?aStudent) if
  doNotProceed(?aStudent)),
  conceptToLearn(?aStudent,?aConcept),
  newline(_),
  write(['You don't master the concept of ']),
  write(?aConcept).
  write(['Try making exercise']),
  adviseExercise(?aConcept).

```

```

Rule adviseExercise(?aConcept) if
  exercise (?aConcept,?anExercise),
  write(?anExercise).

```

Our presented system clearly succeeds in creating some kind of output for the learner. Stronger even, it offers guidance by giving advice to the learner what step should be taken next. Problems can be presented specifically for the learner's needs.

In section 3.4 we formulated a definition about an ILS: "An intelligent learning system is a system that provides individualized and dedicated teaching, adapting the teaching rules to the students' particular abilities" [WW98]. We can say that we provide individualized teaching and that teaching rules are adapted to the needs of the learner, so we can say that we developed some intelligence by presenting the learning process in a declarative way.

6.5.2 Remarks

For presenting a clear and thorough evaluation of this architecture tested on the learning environment LearningWorks, we should do an experiment with two groups of novice learners. We would have to let one group use LearningWorks with our presented intelligence and the other group of novices should work with LearningWorks without any extensions. We should study their learning curve and ask questions to both groups if they found the system to be helpful. It is obvious that this is not realizable within our time constraints.

In our communications model we presented what kind of advice we can offer to a student. It is rather obvious that this is of course too primitive for using this into a learning environment. We could enhance these communication features of our system architecture by making graphical extensions, so that it looks more appealing. We could use queries for example that open up an application window for presenting a suggested exercise to tackle a specific problem the user has.

6.5.3 General Conclusion

Although we stated some remarks and we pointed out some minor drawbacks throughout this chapter, these issues do not obstruct our goal of this thesis, i.e. demonstrating that we *can* enhance learning environment by making them somewhat intelligent to them by expressing knowledge about the learning process.

Before formulating our thesis conclusion we will first point out some interesting future work.

Chapter 7

Future Work

The last chapters validated our claim that learning environments for teaching object technology can be enhanced significantly by expressing declarative knowledge about good programming practices and in particular about the learning process itself. We will now discuss some interesting future research topics with our eye on the future.

7.1 A Framework for Teaching Object Technology

Our general system architecture presented in chapter 5 could be further generalized and put on a higher level. At this moment, a fixed strategy is implemented in the pedagogical model, i.e. a learning path of concepts that should be understood by the learner. Other strategies are possible, and it would be useful to make this variable, meaning that a teacher can plug in his own strategy of learning.

Another interesting research topic could be that one could plug in other student models. This way, we would even be able to specify which learners we are dealing with. This architecture could then even be used for teaching object orientation to people that have had experience already with procedural languages, thus taking the paradigm shift into account (see section 2.2). In this case the student model should be adapted, because this type of learner thinks and acts differently and draws conclusions in another way.

This research would lead to creating a framework for teaching about object technology. The hot spots would then be the *type* of learner that will use the intelligent learning environment, as well as the *pedagogical techniques* that will be used to teach the concepts about object technology. Even the communication model we talked about can vary according to which type of learner will use the learning environment. We saw in earlier chapters that this model represents knowledge about *how* the inferred intelligence will be used to inform and help the learner, so how it will be presented to him/her. To summarize this topic, we would want to create a framework for teaching

object technology by making all models expressed earlier the hot spots of this framework.

7.2 Structure of the knowledge

Until now, we described different kinds of knowledge and we classified them in the appropriate model (for example: the knowledge about the order in which the topics of the learning path should be learned are put in the pedagogical model). However, a lot of knowledge in such a model is strongly related and even has similar parts. Even if we just consider the events that have to be logged. Events that can occur within a certain page in a section of a book have to log also the place where they occur, like:

```
Fact msgSelectEvent(timestamp,aBookName,aSectionName,
                    aPageName,aMsgSelector)
```

This event is logged in LearningWorks whenever a message is selected and sent to an object. In figure 3.2 for example when you would select the message 'forward' for sending the box *joe* a message you log the following event:

```
Fact msgSelectEvent([104],[Boxes Book],[Box Instances],
                    [Unary Messages],[#forward])
```

but you must put the bookname there, the name of the section, etc. . . .

However, if you consider this knowledge to belong in the *scope* of the general knowledge about the section *Box Instances* in the *Boxes Book*, you wouldn't have to repeat the bookname and the section name every time, so our clauses would look simpler and shorter. So we could benefit enormously if we could structure our knowledge *hierarchically*.

Not only LearningWorks lends itself to *hierarchically structured knowledge*, most intelligent tutoring systems support this kind of structure, because they are divided into lessons. The existing systems we discussed in section 3.4.2 also have this structure.

Another issue regarding the structure of the knowledge, in order to have a true declarative framework in SOUL, as mentioned in section 4.3.3, it must be possible to override rules and inherit them.

To realize these features in SOUL, mechanisms that are used in Object Oriented Logic Programming (OOLP) can probably be used. In languages that support OOLP, you can define objects in a declarative way and the behavior of the objects, the *methods* are logic rules and the instance variables are facts to hold them in a variable term of that fact. Let's show an example of a NUOO-Prolog program ¹, an object-oriented extension of Prolog:

¹<http://www.cs.mu.oz.au/%7Elee/src/oolp/>

```

def_object point(x,y) isa object.
  xval(x).
  yval(y).

  move(DX, DY, point(X, Y)) :-
    plus(x, DX, X),
    plus(y, DY, Y).
end_def point(x,y).

```

The example shows a definition of a point object. Mechanisms you could reuse for the above stated purposes of having a *scope* of rules, and *overriding* of rules are those used for information hiding (the instance variables *x* and *y*, that they cannot be accessed from outside the object) here and when a subclass of point would exist and a method has to be looked up. To summarize: what we could use for our purpose are the information hiding and method lookup mechanisms as used in an OOLP language.

7.3 More detailed knowledge

Other kinds of knowledge that could be included in the knowledge base are misconceptions, to avoid some kind of misleading thoughts with students. To intercept misconceptions, we could extend our knowledge base with rules that declaratively describe the wrong thinking patterns described in section 2.1.3. If it is possible to detect these wrong conceptions of learners, the tutoring system can also set the student straight if he starts thinking in a wrong direction. This could be very important, because the longer a misconception is in a person's mind, the harder it is to 'unlearn' it.

An expert model could also be developed, representing a knowledge base of the so called 'perfect learner'. This would provide a way to express more fuzzy knowledge of the learner. Sometimes you can say a student knows a concept *sufficiently* to proceed to a next stage in the learning process. This is hard to express in the student model as it is now.

We could also try to express already previously gathered knowledge about the domain, so that the learner is not confronted in the beginning with learning topics he already knows.

7.4 Extending LearningWorks

In order to be able to use our extension of LearningWorks in a classroom of novice learners, some extensions should be made to the LearningWorks environment first.

To start with, a user of LearningWorks should identify himself when using the system, so the correct knowledge on that learner can be loaded into the

system repository. This way, the learner always starts at the appropriate knowledge level.

Exercises should also be made for each different concept of object technology. They could be used to create stronger barriers to let the student advance to the next concept in the learning path or not. These exercises can also be used to check for object misconceptions (see section 2.1.3) the student could have developed until then. We could for example create an exercise to test whether or not the learner is having difficulties with distinguishing an object from a class (the misconception depicted in table 2.3). Then we would have to present an exercise where many instances of a class are used.

7.5 Pedagogical Patterns

Pedagogical ideas that were presented at OO conferences and published in proceedings and journals have been collated into *pedagogical patterns* [SMM⁺96, MSPM98]. They represent the effective practices of many OO educators into one publication. The format of such a pattern is depicted in table 7.1 ².

We could integrate these patterns in our intelligent environment by detecting within the student's behavior which concepts are not well-understood and letting the environment act upon them.

As an example, let's look at the *mistake* pedagogical pattern ³. In the very early stages of programming, learners will certainly be confronted with errors in their program, but they don't always know how to deal with them. The solution that this pattern offers is that students have to be asked to produce an artifact with certain specific errors (usually a single error). Then, the effect of the error should be explored. For example, students are given an assignment in which they are instructed to create and run a program with certain specific errors. They are then asked to comment on the diagnostics produced and/or why no diagnostics were produced for the error.

Related with this thesis, this means that we could easily trace the *problem* that this pattern formulates and our system could offer the pattern's *solution*. We could have something like this in SOUL: first we trace the following events

```
Fact messageNotUnderstoodEvent(aTimeStamp,anObject,aMsgSel).
Fact responseToMNUEvent(aTimeStamp,#terminate).
```

In Smalltalk when a student is trying to rehearse a piece of code. If we trace these events more than two or three times, we can conclude that he's having problems with debugging, because he never selects the *debug* method.

²<http://www-lifia.info.unlp.edu.ar/ppp/format.htm>

³<http://www-lifia.info.unlp.edu.ar/ppp/pp33.htm>

Name	pattern name
Date	date of last update
Author	name of person submitting the pattern
Thumbnail	short description (abstract) of the pattern
Problem	problem, challenge, or issue that the pattern is addressing
Audience	For what type of learners, in what context, is this pattern appropriate?
Forces	What makes the problem a problem?
Solution	the solution this pattern proposes
Discussion	resulting content/consequences and implementation issues
Special Resources	resources needed to use this pattern (things that are not ordinarily available to the person using the pattern)
Contraindications	when not to use the pattern, including any cultural dependencies
Related Patterns	The author may want to browse the web page (and other sources) and comment on any existing patterns related to this one.
Example instances	specific uses of the pattern
References	any citations and/or individuals who should be acknowledged as contributor

Table 7.1: Pedagogical Pattern format

```

Rule mistakePatternDetected if
    count(messageNotUnderstoodEvent(...),[3]),
    count(responseToMNUEvent(...),[3]).

```

If this rule is true, then an exercise should be presented in LearningWorks where a piece of code should be run and in advance it is mentioned what is wrong with the code. Then the student should choose the *debug* option and make a report, even correct the mistakes.

We will end this thesis by stating our final conclusion and providing a summary of our presented work.

Chapter 8

Conclusions

We will now discuss the results we obtained in the previous chapters and we will draw the conclusions with regard to our initial goal of this thesis. Therefore it is important to formulate again our goal and motivation. Afterwards, we will summarize our approach, point out to what extent we achieved our original aim and what limitations we had to deal with and why.

8.1 Motivation and Initial Goal

The objective of this thesis was to show that learning environments for teaching object technology can profit significantly by using declarative knowledge about good object-oriented programming practices and in particular about the learning process itself for every student. This would be shown by presenting a general architecture for teaching about object technology. This architecture would then be validated by applying it to the learning environment *LearningWorks*, thereby making *LearningWorks* more intelligent.

Our motivation to do so relied in the fact that just letting students practice their object-oriented way of thinking by working with a learning environment sometimes isn't sufficient to let them learn the important concepts about object technology in a supervised way. A lot of guidance is needed, but more human experts are not always available, like in distance education for example.

8.2 Summary and Results

To achieve our goal, we started with an introductory chapter about teaching object-oriented concepts. We also gave a brief introduction on intelligent learning environments and we expressed the need for representing knowledge about the learning process of a novice learner. We ended in that chapter by presenting our approach throughout this work.

In our second chapter we dealt with the main topics on teaching object-oriented concepts. We started by highlighting some problems encountered when teaching an object oriented curriculum. We also discussed when, what and how object orientation should be taught and on each of those questions we clearly stated the answers with regard to this work: we will focus here on a *top-down* teaching strategy to learn to *novice learners* the concepts of object technology by providing them with an *intelligent learning environment*.

Learning environments was our topic in chapter 3. We pointed out why learning environments are used for education purposes and what their educational value is. The conclusions we formulated were that learning environments make learning enjoyable, they stimulate the development of a student's general problem solving ability and they provide well-organized and well-structured courses with clearly stated objectives.

Afterwards, we introduced LearningWorks, the learning environment we used to validate this work. We explained its structure and gave an overview of its main features. We presented the system's curriculum outline, that it aims at teaching software building to novice learners.

Intelligent tutoring systems were also highlighted. We presented the main components of such a system and we discussed two existing ITS's: LISPITS and RAPITS. We mentioned the components we would use in our architecture and we ended by presenting a schematic preview of our architecture for teaching object technology to novices.

Afterwards, we introduced logic programming and SOUL (the Smalltalk Open Unification Language), the logic meta-language we used for expressing our declarative knowledge about the learning process. We first introduced Logic Programming because the semantics of SOUL are based on that. We concluded that a logic meta-program is a very powerful and open medium that suits very well in our goal of expressing knowledge about the learning process. We pointed out what advantages logic programming has over procedural programming, like its multi-way property (being able to use one query for several applications) and that logic languages are very open, i.e. you can easily add or delete knowledge from the knowledge base. We also gave some introductory examples to show how SOUL works and we pointed out some other application domains of this logic language.

Our architecture for teaching object technology in an intelligent way is presented in chapter 5. In a preliminary section we started by formulating the system requirements and we expressed the boundaries of this work.

Afterwards we gave a brief overview of the complete system and we zoomed in on the main system components in detail. We identified four main models: a *student model*, a *domain knowledge model*, a *pedagogical model* and

a *communications model*. We demonstrated the kind of knowledge for each model and we ended by formulating some remarks.

After presenting our system architecture, in chapter 6 we validated our thesis statement by using SOUL to develop our presented architecture on top of the learning environment LearningWorks, thereby adding some intelligence to LearningWorks. We gave a clear overview here of how all the subparts connect to each other and especially how they connect with our architecture of chapter 5.

Then we explained event logging, and how the student model is filled with data. We formulated the results we achieved with our architecture by focusing on the *communication model* because this model is the model the student gets to see. We pointed at that to really test this architecture on LearningWorks for example we would need a group of novice learners and let them work with the system. This however absorbs too much time. We concluded despite all of those little remarks we achieved our goal, i.e. that learning environments can significantly be enhanced by expressing declarative knowledge about the learning process.

The last chapter provided an overview of future work regarding our developed architecture. We pointed out there that LearningWorks should be extended for putting our expressed intelligence into practice. We have to be able to recognize learners and some exercises have to be added to provide a way of testing on some recently learned concept. We could also improve the structure of our knowledge to be able to reuse facts and rules or override them, as well as providing more detailed knowledge. And we discussed what contribution pedagogical patterns could make with our eye on the future.

From all of this we can certainly state that our research, because it opens up several interesting possibilities, delivered an important contribution to the further research on object technology education with an intelligent tutoring environment.

8.3 Final Conclusion

To conclude this thesis, we can certainly claim that our initial goal is hereby achieved, despite some limitations we had to deal with. These limitations however do not alter the fact that we showed that learning object-oriented concepts with a learning environment can be significantly improved by expressing knowledge about the learning process itself.

Bibliography

- [AGMPR95] X. Alvarez, R. González Maciel, M. Prieto, and G. Rossi. Customising learning environments for teaching object-oriented technology to different communities. *Proceedings of International Conference on Teaching and Training Object-Oriented Technology, TATTOO'95, Leicester, UK*, January 1995.
- [Bec89] Kent Beck. A laboratory for teaching object-oriented thinking. *OOPSLA '89 Proceedings*, pages 1–6, October 1989.
- [BPLR91] Hugh Burns, James W. Parlett, and Carol Luckhardt Redfield, editors. *Intelligent Tutoring Systems - Evolutions in Design*. Lawrence Erlbaum Associates, 1991.
- [Bru95] Peter Brusilovsky. Intelligent learning environments for programming: The case for integration and adaptation. In J. Greer, editor, *Proceedings of AI-ED'95, 7th World Conference on Artificial Intelligence in Education*, pages 1–8, August 1995.
- [BSH98] Joseph Beck, Mia Stern, and Erik Haugsjaa. Applications of ai in education. <http://www.acm.org/crossroads/xrds3-1/aied.html>, 1998. Crossroads, the ACM's First Electronic Publication.
- [CA92] Albert T. Corbett and John R. Anderson. Lisp intelligent tutoring system: Research in skill acquisition. In Jill H. Larkin and Ruth W. Chabay, editors, *Computer-Assisted Instruction and Intelligent Tutoring Systems*, chapter 3, pages 73–109. Lawrence Erlbaum Associates, 1992. Advanced Computer Tutoring Project, Carnegie Mellon University.
- [Cob95] Collins cobuild english dictionary, 1995. The Cobuild series from the bank of English.
- [Cos92] Ernesto Costa, editor. *New Directions for Intelligent Tutoring Systems*, volume 91 of *F: Computer and Systems Sciences*.

- Springer-Verlag Berlin Heidelberg, 1992. NATO ASI series, result of the NATO Advanced Research Workshop on New Directions for Intelligent Tutoring Systems, held in Sintra, Portugal, October 6-10, 1990.
- [CS90] William J. Clancey and Elliott Soloway, editors. *Artificial Intelligence and Learning Environments*. MIT press, 1990. Preface of the book written by the editors.
- [CTC93] Y. S. Chee, J. T. Tan, and T. Chan. Applying cognitive apprenticeship to the teaching of Smalltalk in a computer-based learning environment. *Proceedings of 7-th International PEG Conference*, pages 569–588, 1993.
- [Dod99] Mahesh Dodani. Cultivating a software-development mentoring culture. *JOOP - Journal of Object-Oriented Programming*, 12(1):68–69 and 74 and 78, march/april 1999.
- [DPKV94] Wim De Pauw, Doug Kimelman, and John Vlissides. Modeling object-oriented program execution. *Proceedings of the 8th European Conference, ECOOP*, pages 163–182, July 1994.
- [Duc99] Stéphane Ducasse. Evaluating message passing control techniques in smalltalk. *JOOP-Journal of Object-Oriented Programming*, 12(3):39–50, June 1999.
- [DV98] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, September 1998.
- [Fla94] Peter Flach. *Simply Logical - Intelligent Reasoning by Example*. John Wiley and Sons Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 1994. Reprinted December 1994.
- [GHJV95] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts 01867, 1995. Eleventh print, May 1997.
- [GLH96] Cleveland Gibbon, Gillian Lovegrove, and Colin Higgins. Tools, heuristics and techniques to assist oo education. OOPSLA '96 Educators' Symposium Notes, October 1996.
- [Gol93] Adele Goldberg. Wishful thinking. *JOOP: Journal of Object-Oriented Programming*, 3(4):87–88, December 1993.
- [Gol95] Adele Goldberg. What should we teach? OOPSLA '95 - Addendum to the proceedings, 1995.

- [Gol96] Adele Goldberg. *Learning About Building Software Systems*, 1996. Companion Book for the learning environment LearningWorks.
- [Gol97] Adele Goldberg. The learningworks development and delivery frameworks. *Communications of the ACM*, 40(10):78–81, October 1997.
- [HGW97] Simon Holland, Robert Griffiths, and Mark Woodman. Avoiding object misconceptions. *SIGCSE Bulletin*, 29(1):131–134, March 1997. Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education.
- [Joh90] Lewis W. Johnson. Understanding and debugging novice programs. In William J. Clancey and Elliott Soloway, editors, *Artificial Intelligence and Learning Environments*, pages 51–97. MIT press, 1990.
- [KKR99] Michael Kolling, Bert Koch, and John Rosenberg. Requirements for a first year object-oriented teaching language. 1999.
- [Kol99a] Michael Kolling. The problem of teaching object-oriented programming, part 1: Languages. *JOOP: Journal of Object-Oriented Programming*, 11(8):9–15, January 1999.
- [Kol99b] Michael Kolling. The problem of teaching object-oriented programming, part 2: Environments. *JOOP: Journal of Object-Oriented Programming*, 11(9):6–12, February 1999.
- [KR99] Michael Kolling and John Rosenberg. Blue - a language for teaching object-oriented programming. 1999.
- [Lal94] Wilf Lalonde. *Discovering Smalltalk*. Addison Wesley Publishing Company, Reading, Massachusetts 01867, 1994.
- [Lar97] Gilles Larin. Computer-based learning environments. <http://calvin.stemnet.nf.ca/elmurphy/emurphy/-computers.html>, 1997. This page was produced by Elizabeth Murphy in the context of courses TEN-61937 & TEN-61938, Universite Laval, Quebec City, Quebec, Canada, Summer, 1997.
- [LP90] Wilf R. Lalonde and John R. Pugh. *Inside Smalltalk*, volume 1. Prentice Hall International Editions, EngleWood Cliffs, N.J. 07632, 1990. A Division of Simon & Schuster.

- [LPR94] C. Leonardi, M. Prieto, and G. Rossi. Micro-worlds: A tool for learning object-oriented modeling and problem solving. *Proceedings of the Educator's Symposium. ACM Conference on Object Oriented Programming Systems Languages and Applications OOPSLA '94, Portland, Oregon, USA*, October 1994.
- [LR89] Karl J. Lieberherr and Arthur J. Riel. Contributions to teaching object-oriented design and programming. *OOPSLA '89 Proceedings*, pages 11–22, October 1989.
- [Mey93] Bertrand Meyer. Towards an object-oriented curriculum. *Journal of Object-Oriented Programming*, 6(2):76–81, May 1993.
- [Mic98] Isabel Michiels. Using logic meta-programming for building sophisticated development tools. Thesis submitted for obtaining the degree of Licentiate in Computer Science, Vrije Universiteit Brussel (VUB), Belgium, May 1998.
- [MSPM98] Mary Linn Manns, Helen Sharp, Maximo Prieto, and Phil McLaughlin. Capturing successful practices in ot education and training. *JOOP - The Journal of Object-Oriented Programming*, 11(1):29–34, march/april 1998.
- [MW98] Kim Mens and Roel Wuyts. Declaratively codifying software architectures using virtual software classifications. 1998.
- [Pai95] A.M. Paiva. About user and learner modeling - an overview. Technical report, INESC, IST, Technical University of Lisbon, Portugal and Department of Computing, Lancaster University UK, December 1995.
- [Pai99] A. Paiva. Computer-based learning environments. <http://www.rnl.ist.utl.pt/ic-eac/POR/eac-doc-por.html>, 1999. Course Lecture Notes.
- [Pap92] S. Papert. *The Children Machine*. Basic Books, NY, 1992.
- [PR95] Maximo Prieto and Gustavo Rossi. The importance of learning object-oriented thinking. *Workshop on Learning, training and teaching in Object Technology, European Conference of Object-Oriented Programming (ECOOP '95), Aarhus, Denmark*, August 1995.
- [RDW98] T. Richner, S. Ducasse, and R. Wuyts. Understanding object-oriented programs with declarative event analysis. Ecoop'98 Workshop on OO Reengineering, ECOOP'98, Brussels, Belgium, July 20-24th, 1998.

- [Seb96] Robert W. Sebesta. *Concepts of Programming Languages*. Addison Wesley Publishing Company, third edition, 1996.
- [SMM⁺96] Helen Sharp, Mary Lynn Manns, Phil McLaughlin, Maximo Prieto, and Mahesh Dodani. Pedagogical patterns - successes in teaching object technology. *ACM Sigplan*, 31(12):18–21, December 1996.
- [WB92] Radboud Winkels and Joost Breuker. What's in an ITS? a functional decomposition. In Ernesto Costa, editor, *New Directions for Intelligent Tutoring Systems*, volume 91 of *F: Computer and Systems Sciences*. Springer-Verlag Berlin Heidelberg, 1992.
- [WBWW90] Rebecca Wirfs-Brock, Brian Wilerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.
- [Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. *Proceedings of TOOLS USA '98*, March 1998.
- [WW98] Pam Woods and James Warren. An architecture for a rapidly prototyped intelligent tutoring system with a sound pedagogical design. 1998.

Index

- The Demeter System , 21
- abstract data type (ADT), 10
- Aggregation, 50
- architecture, 30
- Architecture for learning OOT, 44
 - Requirements, 45
- Blue teaching environment, 22
- CBLE
 - Why?, 25
- Codifying Software Architectures, 43
- cognitive, 24
- Cognitive Apprenticeship, 21
- Collaboration, 50
- Collaborations, 20
- Communication Model, 52
- component, 30
- Computer-Aided Instruction (CAI), 25
- Computer-Based Training (CBT), 25
- CRC-Cards, 20
- Declarative
 - semantics, 36
- Design Pattern Structure Rules, 42
- Domain Knowledge Model, 49
- dynamic type binding, 10
- Event Logging, 57
- framework, 30
- framework for teaching OOT, 64
- guidance, 52
- Imperative programming, 37
- Inheritance, 51
- inheritance, 10
- intelligent learning environments, 11
- Intelligent Tutoring Systems, **30**
 - Components, 31
 - LISPITS, 33
 - RAPITS, 33
- is-a relationship, 50
- is-part-of relationship, 50
- Learning Environments, 24
- Learning Path, 48
- LearningWorks, **26**, 55
 - CourseBinder, 27
 - Curriculum, 29
 - possible extensions, 66
 - Structure, 27
 - The Authoring Tool, 29
- Logic Meta-Programming, 37
- Logic Programming, **35**, 36
 - Characteristics of, 36
- mentoring, 11, 17
- Message Passing Control Techniques, 58
- meta-programming, 37
- Methods, 51
- Microworlds, 21
- misconceptions, 17
- multi-way, 36
- Object Misconceptions, 46
- object misconceptions, 17
- Object-Oriented Logic Programming, 66

Object-Oriented Programming, **10**
Objects/Classes, 50

paradigm shift, 19

Pedagogical Model, 48

pedagogical patterns, **67**
 format of, 67

Polymorphism, 52

polymorphism, 10, 60

programming environments, 15

Programming Style Rules, 41

PROLOG, 38

rehearsal worlds, 27

Responsibilities, 20

Smalltalk, 15

software construction
 aspects of, 29

SOUL, 35, **38**
 advanced searching, 40
 Application Domains, 40
 Declarative Framework, 40

Student Model, 31, 47

system, 29

Teaching OO, **13**, 17
 existing approaches, 20
 How?, 19
 Problems, 13
 What?, 10, 18
 When?, 18

The Analysis Package, 56

The Student Model
 Construction, 58