Experiences from Teaching Software Development in a Java Environment

Ruth Breu **Sybille Hellebrand** **Michael Welzl**

Institute of Computer Science
Leopold-Franzens-University of Innsbruck
Innsbruck, Austria
email {Ruth.Breu, Sybille.Hellebrand, Michael.Welzl}@uibk.ac.at

Abstract

In this paper, we present our ideas for basic education in software development integrating the engineering aspect right from the start. We discuss the aspects which are crucial for such an education and summarize our experience with a four semester beginner's course at our university. Our considerations are concluded by an analysis of the role of Java within the approach.

INTRODUCTION

The goals of a computer scientist's education in software development are quite clear: The prospective computer scientist should be able to produce high quality software, i.e. software which fulfills the client's need, is extensible, reliable and runs efficiently. Moreover, he or she should be able to work in a team and to work within the context of a project, e.g. considering the restrictions of a budget.

Our claim is that the engineering aspect of software development, i.e. the aspect that software is produced in teams with given targets and quality standards, should be taught right from the start of the education. Disciplined programming in the small is the most important prerequisite for successful programming in the large. However, like in any company, high software quality cannot be simply prescribed but requires a supporting climate. For a university this means that software engineering principles should be integral part of the whole education.

This paper is written out of the experiences with a four semester beginner's course "Software Development". This course has been held at the University of Innsbruck, Institute of Computer Science, as part of a new Bachelor education which started in winter semester 2001; it consists of a two hours lecture and three hours lab course in each of the four semesters. As programming languages we use Java and C++. Software engineering is taught as an integral part of programming from the outset.

In this paper we identify the key factors for this integrated approach, which we separate in three main aspects. The first aspect is programming in the small. We discuss the concepts, techniques and methods we consider to be crucial for disciplined team oriented programming. In particular, the students should be faced with the quality and team aspect from the first semester on. This means that testing, reflection about the code structure and documentation are integral part of lectures and exercises.

The second aspect is programming in the large. We identify the techniques and methods that should, in our opinion, be part of a basic education in analyzing and designing complex systems and in project planning and control. Part of the student's exercises is the execution of a one-semester project, which comprises the application of project management and modeling techniques as a central aspect. For this purpose, we defined our own simple process model which is adequate for the size of the project. Moreover, the students work in this project with a real development environment (e.g. comprising tools for modeling, testing and configuration management).

The third aspect concerns foundations of programming and software development. Our claim is that knowledge of the basic principles, system views and specification techniques is a prerequisite for developing problem solving and system structuring capabilities.

The structure of the paper is as follows: in sections 2, 3 and 4 we detail the three aspects identified above. In section 5 we summarize the role of Java for teaching the concepts presented. In section 6 we give a conclusion. In order to ease orientation we provide an overview of the contents of the course in Table 1.

PROGRAMMING IN THE SMALL

We organize this chapter as a collection of topics discussed subsequently.

Principles of Programming Languages

In the first semester we teach the main principles of programming languages. Following the book of Broy [1] we start with functional elements (functional-style programming with expressions, conditional expressions and recursion). Subsequently we go over to procedural programming constructs such as variables, statements, procedures and data structures.

Table 1: Contents of the Software Development Course

Semester 1:

- Basic Concepts of Programming
- Syntax and Semantics
- Functional and Procedural Programming Paradigms
- Verification and Testing of Simple Programs

Semester 2:

- Object Oriented Programming
- Inheritance
- Polymorphism (Templates, Operator Overloading, Dynamic Binding)
- Standard Template Library
- Error Handling

Semester 3:

- Event Oriented Programming and Graphical User Interfaces
- Streams
- Concurrent Processes
- Introduction to Client/Server Programming

Semester 4:

- Modeling Techniques
- Requirements Elicitation
- Designing Component Structures
- Quality Management
- Process Models

Even though, in our opinion, Java is a language that is very appropriate to be used for teaching software development skills, it is not capable of covering the basic principles of the programming language landscape. Therefore, we use an abstract notation (based on [1]) in the lectures to present core programming principles. In the exercises we either use Java or C++ (e.g. for programming with pointers).

In another part we focus on object oriented programming principles introducing the notions of objects, classes, inheritance, polymorphism, generic classes, interfaces and other well-known topics.

Documentation

Documentation is a topic which we deal with in many parts of the lectures and which is an important issue in the exercises. We use several techniques for documentation:

- Style guides which introduce programming conventions for naming classes, variables and methods (in order to ease understanding code itself)
- Documenting statements in the code
- The Javadoc tool for documenting the class structure of programs (including the Javadoc-specific specifications: authors, textual description of classes, specification of methods with textual pre- and postconditions, parameters, exceptions, etc.)
- UML class diagrams and sequence diagrams for higher level documentation of the static system structure and object interactions

In our experience the early use of documentation techniques (starting with the first semester) improves the quality of programming in advanced stages, since the students become accustomed with a careful coding style and experience the benefits of documented code.

Testing

As with documentation, we start with disciplined testing in early stages of the education. In particular, regression testing with a tool like JUnit has shown to have fruitful effects on the quality of the student's programs. Moreover, in the student's project of the fourth semester we use system tests. Based on test scenarios the student teams have to test their systems on one another.

Design Patterns

Design patterns like the ones presented in [2] constitute good design practices. Albeit a presentation of design patterns in the form of a catalog is appropriate for books, this is not the case for lectures. In order to bring design patterns into use, we continuously include them in our lectures and the exercises within sample programs and problems. For example, we talk about the composite pattern when introducing the structure of Java Swing containers and use the iterator pattern when talking about container data structures.

Through this continuous stress on design patterns we aim to enhance recognition of specific design problems in real programming.

Architectural Patterns

With architectural patterns like the layering of systems we pursue a similar approach: rather than talking about constructing layers in general, we demonstrate this architectural principle in detail with the example separating the user interface layer from the application kernel. In particular, we discuss various Java techniques (e.g. inner classes and anonymous classes) for building elaborate MVC structures.

Reuse

Reuse is another topic which requires continuous training. We prompt students to reuse code in a number of different ways.

In our exercises we continuously work with the Java class libraries (e.g. students tend to use arrays everywhere, neglecting comfortable container data structures).

Moreover, we do exercises in which we force the reuse and extension of the student's own code. As an example we had an exercise at the beginning of the semester in which the students had to program a graphical user interface for tic-tac-toe (with the option of human or computer players). At the end of the semester the students had to extend this program by the facility of playing over a network. For other exercises we gave students part of the code (e.g. the graphical user interface) in order to train reuse.

Social Aspects

A crucial factor for achieving high software quality is the motivation of the programmer. The best techniques will fail if the programmer does not apply them in the right way and with no motivation. In our opinion the best way to convince students to use the learned techniques is success and failure.

As an example, the extension of the tic-tac-toe game to a network version posed great problems to students who did not document their code in an appropriate way – in contrast, the exercise was a success for other students who restructured and documented their code after the successful implementation of the first phase.

A few further aspects contribute to a proliferous climate; we want to mention two of them. The first is that students should learn to talk about their programs. If the programmer is able to present his or her program to an audience, then he or she has clearly understood both the problem and the solution. We therefore made a brief presentation part of all exercises.

A second aspect is to reach a common understanding among students that code produced does not belong to the programmer alone but to the team. Thus, pairwise programming as practiced in eXtreme Programming is a very valuable technique which can support this view.

PROGRAMMING IN THE LARGE

The concepts of programming in the large are part of the fourth semester, during which we teach basic modeling and development techniques for executing bigger projects. The main lecture is complemented by an accompanying lecture which deals with the aspects of team organization and project management. In the exercises the students have to realize a system in teams of three students. During the last term, students implemented a component of a project management tool which allows team members to book worked hours for given projects.

The goal of the student's project is to exercise the modeling techniques presented in the lecture and to work with a real development environment. Apart from the basic programming and testing environment the tool set basically consists of

- a UML modeling tool
- a cost estimation tool
- a CVS server

Process Models

In the lecture we introduce principles of process models as e.g. supported by the Unified Process [3] or Catalysis [4].

For the student's project we define a simple process model which is based on a waterfall model extended by the early definition of a user interface prototype and the definition of test scenarios. The process model is depicted in Figure 1 and detailed under the subsequent topics. The figure does not contain the activity of project management which supports and surveys the whole process. The process model is described in a project manual. Moreover, we support the process by a number of given templates (i.e. for the use case model and the test scenarios).

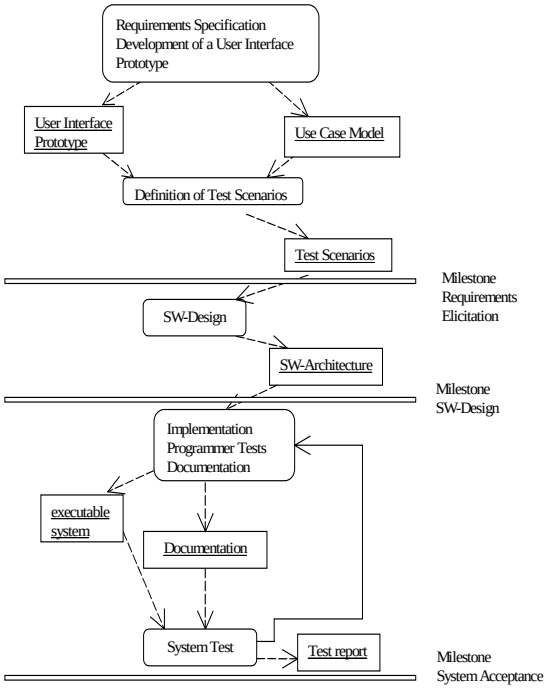

Figure 1: Process Model used in the Student's Project

Modeling Techniques

Apart from the basic UML diagrams (e.g. class diagrams, sequence diagrams, state diagrams, activity diagrams) we introduce predicative specification of object invariants and methods based on the main concepts of OCL.

Requirements Specification

We identify the basic steps of use case oriented development based on the approach of [5] and [6]. Moreover, we sketch informal modeling techniques like

mindmaps and CRC cards, and we present approaches to business process modeling.

The main goal in teaching techniques of requirements specification is to train students to think in terms of the application domain in early stages of design (contracts and clients instead of data base tables). Moreover, the students should be able to fix requirements in early stages of the design. We exercise requirements elicitation by simulating sessions with clients.

The requirements specification in our project (i.e. the use case model) basically consists of a use case diagram with use cases described in plain text together with a class diagram modeling the static concepts of the application domain.

The requirements specification phase is accompanied by the development of a user interface prototype. The user interface prototype serves to gain a common understanding with the team instructor about the system functionality and how users interact with the system. The user interface is prototypical in the sense that it presents dummy data and is not complete.

Constructing a Component Structure

The field of software architectures is only sketched in the basic lecture and left to advanced courses. The lecture touches on techniques for modeling software architectures (based on the concepts of components and interfaces and on different abstraction levels). Moreover, the most pregnant architecture schemes such as layered architectures and variants of client/server architectures are discussed. Details of these architectures are given in the respective lectures, e.g. about gui programming and client/server programming.

The development of the software architecture is an important milestone within the student's project. In the simple technical environment (consisting of a Java Swing interface, a Java kernel and a database), the software architecture mainly serves to elaborate the central ideas of the implementation. This comprises e.g. the cooperation of objects in different layers or message flows describing the execution of use cases.

A second task within the elaboration of the software architecture is the definition of a component structure which supports division of work among -team members. Here, the focus is on the definition of layers and component interfaces.

In our experience the definition of the software architecture poses a lot of problems to unexperienced developers. In particular, developing descriptions at a high level of abstraction requires much experience at the programming level. Nevertheless, we hope that we can convince our students to design systems before programming.

In order to support the learning process we go the other way round. As part of the documentation of the executable system we demand a high-level component description. That way, students are forced to think about the system architecture at least after the implementation.

Quality Management

In the lecture the standard techniques of quality management like reviews and code inspection and testing techniques are introduced. Moreover, we treat the interrelations between software development and quality measures in process models such as the Unified Process or the V-Model [7].

In the student's project we distinguish between developer tests and system tests. The developer tests are performed together with the implementation of classes supported by the JUnit tool. For the final system tests students define a set of test scenarios in the early stages of the design based on the use case model. Figure 3 shows an example for such a test scenario.

Test Case: incorrect login	
Name:	Incorrect login
Use Case:	*Login*
Initial State:	The application presents the login window.
Action:	The user enters the user name „smith“ and password “123456”
Expected Final State:	The application presents the login window with the error message „wrong password“, the field for the user name is filled with the user name “smith”, the password field is empty.
Observed Divergence:	
☐ OK ☐ minor divergence ☐ medium divergence ☐ great divergence ☐ system unusable	

Figure 2: Sample Test Scenario

The test itself is performed after the implementation phase. The teams test the system of each other, filling out the fields of the given test scenarios (resulting in the test report).

Project Management

Project management is an important part of our project. The project plan is initialized at the beginning of the project. In the project plan we fix the products to be developed and the activities according to the process model presented in Figure 1. The student's task is to plan the milestones and to keep track of the hours they work for the project.

The planning of schedules and efforts is another activity which poses great problems to a majority of the students. While the project is accompanied by a special lecture about team organization and project management in which the topics of cost estimates and schedule planning are treated, students simply lack experience.

Here we follow again the approach of self reflection. We demand a final project report in which the students are forced to analyze the course of the project. This comprises the following aspects:

- If and why the planned and the actual efforts differ?
- If and why the specification documents (use case model and software architecture) and the realized system differ?
- What could be done in a better way in the next project?

Social Aspects

We all know that the social capabilities of the team members are a key factor for the success of a project, may it be a real project or a student project. We will not do further reflection about that point at this place.

Instead, we want to point out the fact that a key target of the project is to convince students about the benefits of the specification and management techniques they are applying during the project. In particular, use case description and software architecture description should not be experienced as bureaucracy but as helpful techniques for fixing requirements and should be used as communication basis between clients, architects and programmers. Therefore we elaborated "lightweight" models which only contain the core concepts and do not overstrain unexperienced users.

BASIC FOUNDATIONS

Obviously, a central task of any software development course is to teach the foundations of programming and specification. In our opinion, the use of formal methods is of great importance for achieving a deep understanding of concepts – but, on the other hand, the approaches presented must have strong connections with the practical parts.

In what follows, we shortly describe the basic topics that are part of our curriculum.

Syntax and Semantics of Programming Languages

In the first semester we follow the approach of [1] and introduce the basic concepts of functional and procedural programming (such as sorts, terms, recursion, states and variables, constructs of control flow etc.). These concepts are accompanied by denotational semantics in an algebraic model theory.

Predicative Specification of Sequential Systems

In the next step we introduce the Hoare calculus for specifying sequential procedural programs based on the semantics defined earlier. With the Hoare calculus we demonstrate the principles of specification and implementation together with the notion of correctness. Moreover, the principle of abstraction can be studied in a foundational way in this setting.

In later parts of the lecture the basic ideas of Hoare's calculus are picked up and connected with pragmatic specification approaches. In particular, we discuss the specification of class invariants and pre- and postconditions of methods in object oriented modeling.

Programming and Specifying Concurrent Processes

In order to give students the opportunity to gain experience with programming their own examples, we start with the synchronizing concepts of Java threads. Afterwards we study general approaches to synchronization and communication (such as semaphores, monitors and message passing) and the related problems (such as deadlock and starvation).

Due to the fact that semantics and specification play a special role for distributed systems we discuss both a semantic approach to the modeling of concurrent processes (the framework of action structures [1]) and a bundle of specification techniques (e.g. petri nets).

The section about concurrency is continued with a practical part about client server programming. Here we start with basic socket programming in Java followed by concepts of Java RMI and a sketch of CORBA.

ABOUT THE ROLE OF JAVA

In our opinion the central goal of an introductory lecture such as ours should be to teach concepts rather than specific languages. On the other hand, students need an appropriate basis for exercising the concepts. Java together with the available libraries and tools turned out to be such an appropriate basis in our lecture for a number of reasons – one being well-designed support of basic object oriented programming techniques. This functionality allowed us to teach inheritance as well as polymorphism; a downside for teachers is the lack of generic data types.

Java Swing, Java concurrent programming and Java RMI enabled us to develop lectures and exercises which integrate general concepts and application oriented programming. In particular, the powerful combination of Java threads and Java Swing provided us with a wide range of interesting programming problems. The large number of classes available in the standard JDK API – uniform handling of heterogeneous data streams and containers in particular – enabled us to create small exercises which are solely concerned with the relevant programming aspects while being effective enough to motivate students. Some of our exercises were:

- Deadlock prevention of two cars on a bridge (we provided the classes that realized the graphical objects)
- A "Pretty Printer" that turns java code into simple HTML in order to realize syntax highlighting
- A graphical pocket calculator
- A graphical tic-tac-toe game which can be played over the Internet
- A RMI server that receives arbitrary java code for execution

A further advantage is the variety of (free) tools supporting software engineering aspects. This comprises e.g. documentation in Javadoc, testing with JUnit, programming environments and modeling tools supporting Java.

Java should of course not be the only language tackled in a basic lecture – other approaches should be covered as well. Among them are functional programming and classic procedural programming. Moreover, the specification aspect is not part of Java and should be taught based on a variety of approaches (such as UML, predicative specification of invariants, petri nets, etc.).

CONCLUSION

In the preceding sections we presented our ideas for basic education in software development integrating the engineering aspect right from the start. Our claim was that the key capabilities for developing high quality software can and should be trained in the small. This concerns for example adequate code structuring, documentation and testing.

Moreover, we hope that we can sensitize our students for the problems that arise when developing complex real systems. Among these problems are the communication between clients and developers, requirements elicitation and the careful design of a software architecture.

We feel that we are on a good way but much work has yet to be done and many further ideas await realization.

REFERENCES

[1] M. Broy, *Informatik. Volume 1 and 2*, 2nd edition, Springer, 1998.

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

[3] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[4] D. D'Souza, A. Wills, *Objects, Components and Frameworks with UML*, Addison-Wesley, 1999.

[5] R. Breu, *Objektorientierter Softwareentwurf – Integration mit UML*, Springer, 2001.

[6] R. Breu, *An Integrated Approach to Use Case Based Development*, In preparation.

[7] http://www.v-modell.iabg.de