

A New Pedagogy for Programming

Jan Dockx
Eric Steegmans

Report CW 339, Jun 2002



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A New Pedagogy for Programming

Jan Dockx

Eric Steegmans

Report CW339, Jun 2002

Department of Computer Science, K.U.Leuven

Abstract

This paper presents our experiences with teaching object oriented programming at the university and in the industry. We believe that the object-oriented paradigm shift makes it necessary to depart from the traditional pedagogy for programming in the small, based on sequence, selection and iteration, in favor of higher level programming based on the contract paradigm and behavioral subtyping. These experiences resulted in a Dutch book, *Objectgericht programmeren met Java*. We discuss why we think such a change is necessary, the pedagogy and structure of the courses and the book, the didactic form we use, and the results we got over the last 5 years. We conclude with some ideas for future change of the course and the curriculum.

Keywords : object oriented programming, pedagogy, IPS, Informatie- en programmastructuren, Data and Algorithms

CR Subject Classification : K.3.2, D.1.5.

A New Pedagogy for Programming

Jan Dockx, Eric Steegmans

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A
3001 Leuven, Belgium
{Jan.Dockx, Eric.Steegmans}@cs.kuleuven.ac.be
<http://www.cs.kuleuven.ac.be/>

Version 1, Monday, 8 April 2002

Position paper for the Sixth Workshop on Pedagogies and Tools for Learning Object Oriented Concepts, ECOOP 2002, June 11, 2002, Malaga, Spain.

Version 2, Tuesday, 28 May 2002

CW Rapport 339

Please refer to this report as published in the ECOOP Workshop Reader once it is published.

Abstract. This paper presents our experiences with teaching object oriented programming at the university and in the industry. We believe that the object-oriented paradigm shift makes it necessary to depart from the traditional pedagogy for programming in the small, based on *sequence*, *selection* and *iteration*, in favor of higher level programming based on the *contract paradigm* and *behavioral sub-typing*. These experiences resulted in a Dutch book, *Objectgericht programmeren met Java*. We discuss why we think such a change is necessary, the pedagogy and structure of the courses and the book, the didactic form we use, and the results we got over the last 5 years. We conclude with some ideas for future change of the course and the curriculum.

Introduction

Since 1997, we teach a course on object-oriented programming. The course is taught as the second programming course in different academic programs in computer science and electronic engineering, and under contract for the industry. Our experiences led to a book in Dutch, which is now in publication [Steegmans, Dockx 2002].

Traditionally in a university program in computer science, there is a first programming course that has a more intuitive approach. The second programming course is considered to teach the science of programming. This course traditionally is a variation on the pedagogy introduced by Dijkstra and Hoare [Dahl, Dijkstra, Hoare, Genuys 1972], and Wirth [Wirth, 1976]. The traditional second course focuses on techniques to develop and reason about algorithms, and introduces a number of well-known data structures and algorithms (which would be called low-level *patterns* in this day and age). The course mainly focuses on *programming-in-the-small*. Our course still is called *Informatie- en programmastructuren*, which is Dutch for *Data Structures and Algorithms*. It evolved out of the traditional second programming course, and still has to fill more or less the same space in the computer science program. However, the content does not really fit the title anymore. Object-oriented programming is significantly different from traditional, imperative procedural programming, and a different pedagogy is in order.

Below we will sketch the pedagogy we use for teaching object-oriented programming, and our experiences with it. Obviously, this is a work in progress.

Java is used in our course as the example language.

2 Kinds of Existing Pedagogies

We have recognized 2 existing general approaches to teaching object-oriented programming. Neither seemed suitable. We find books that focus on a language, and books that focus on programming.

Most handbooks on object-oriented programming limit themselves to a very detailed discussion of all concepts that are offered in a programming language such as C++ or Java. These books mostly completely pass over the art of *good programming*. These kinds of books are useful for persons that are al-

ready experienced in object-oriented programming, to quickly learn the technical details of a new object-oriented programming language. In our opinion it is not wise to use this approach as a first contact to object-oriented programming, or programming in general. The main reason for this is that the current generation of (object-oriented) programming languages allows for a large number of constructs that should not be used in *good object-oriented programs*. The concepts that form the basis of *good object-oriented programming* originated after the community gained experience with object-oriented programming languages: it has taken us a number of decades to find out which techniques do lead to better software, and which do not. A language such as C++ carries an enormous load of *old technology*. In this respect Eiffel as well as Java are a huge step forward, but still we find in even the best language oriented books examples that are not acceptable according to the principles of good object-oriented programming. Of course an intimate knowledge of the typical syntactic concepts of object-oriented languages is essential. But even more important than acquiring this rather technical background, is to know how these concepts should be handled to come to high-quality software. Inheritance, e.g., is 1 of the pivot concepts of object-oriented programming, and one cannot do without knowing the rules that govern the definition of a subclass. At least equally important though it is to know the principles that need to be applied when a class hierarchy is developed. Moreover, people should be aware of the advantages that inheritance brings to object-oriented programming languages over languages without inheritance.

For over 30 years, programming courses follow the same pedagogy. Dijkstra and Hoare are most cited for laying the foundation for this pedagogy in *Structured Programming* [Dahl, Dijkstra, Hoare, Genyuys 1972][Dijkstra 1969-1970], together with Wirth in *Algorithms + Data Structures = Programs* [Wirth, 1976]. According to this pedagogy, programming is based on 4 concepts — *sequence, selection, iteration* and *abstraction* — and programming is taught based on these 4 concepts. This approach has proved extremely successful to teach people the art of structured programming in imperative procedural languages. We believe however that these 4 basic concepts have becomes less important in object-oriented programming. They are relieved by new concepts: *encapsulation, object structure* and *object interaction*, and *inheritance, polymorphism* and *dynamic binding*. Methods in object-oriented programming are so small that the techniques to develop complex algorithms, such as stepwise refinement, have become less relevant. The complexity in most modern software has moved from the algorithms to the interaction between objects. Complex selection structures have shown to be the most vulnerable elements in software systems. In object-oriented programming we strive for as little explicit selection structures as possible, in favor of implicit selection through dynamic binding. The major complexity in algorithms has always been iteration structures. In object-oriented programming, by applying encapsulation and reuse, we are capable to implement this complexity once and for all, correctly and efficiently, and use it where we need it without headaches [Dockx, van Dooren, Steegmans 2001][Dockx, van Dooren, Steegmans 2001a]. Good structures that support weeding out iterations are very recent [Blakeley, Dockx, van Dooren 2001]. This surely is one of the areas where object-oriented programming will further evolve in the near future. Most books that have *learning how to program* as their primary goal are not adapted to object-orientation. The classic pedagogy is used, with Java syntax instead of, e.g., Pascal as example language. Object-oriented concepts are mostly introduced at the end of the book, almost as an afterthought. This gives rise to the impression that object-oriented programming is merely an extension of classic imperative procedural programming. We believe that object-orientation needs a new pedagogy. The old concepts are still important, but only for *programming-in-the-small*; they are no longer are the heart of the matter.

Overview of the Course

In the course most concepts of good object oriented software development are discussed based on the contract paradigm, as presented by Meyer [Meyer 1991][Meyer 1997], extended with behavioral subtyping [Liskov, Wing 1994]. These techniques lead to guidelines that a developer can use to decide whether or not his code satisfies logical rules of good code. We use a formal behavioral interface specification language (BISL). A concise table of contents is shown below.

In the first part of the course we focus on the definition of a class as a description of objects with common properties and behavior. A class is looked at as a contract between the developer of the class and the users. This leads to 2 complementary descriptions of a class, the specification on the one hand and the implementation on the other hand. We tell students that in the field they can decide whether they want to specify informally, structured or formally, but in the course we try to be as formal as possible. The part ends with a discussion of total methods and robustness, introducing exceptions as the modern way of doing things. Most basic Java features are introduced on the fly.

In the second part of the course the accent moves from the development of 1 single class in isolation to the structuring of software systems as a whole of related classes. To enable communication, first we need relationships between objects. In this part several strategies are discussed to realize different kinds of relationships. To be able to represent the relationships between classes in a software system in a concise way, a very limited graphical notation, based on UML, is introduced. This is the first small step for students on the way to object oriented design of software systems. During this discussion, the need to work with references and to manage the memory becomes apparent. Also, packages are discussed, because in Java this gives you the opportunity to create methods with limited accessibility. In the C++ version, friend declarations are used, and namespaces are only mentioned as a syntax feature. When discussing relationships, multiple associations show up, which need arrays or more advanced data structures and iterative algorithms to get implemented. This leads to a discussion of loop variants and invariants as key in the reasoning about the algorithm, in the traditional way.

The topic of the third part of the course is inheritance. First the basics and the syntax are introduced through examples of extension inheritance, focusing on reuse. Polymorphism, overwriting methods and dynamic binding are discussed in the context of specialization inheritance. The concepts of static and dynamic type play a pivotal role here. The third chapter brings the underlying theory that shows which inheritance hierarchies are correct and which are not: behavioral subtyping. This is considered a natural extension of the contract paradigm. Interfaces are discussed as reverse contracts, with the event pattern (a.k.a. Observer/Observable [Gamma, Helm, et al. 1994], or Model-View-Controller, or MVC) as example. The introduction of nested classes closes this part.

In the fourth part, recursive and iterative algorithms are compared, using the agenda mechanism, and backtracking is discussed. Basic list forms are shown, exposing e.g. the concept of sentinels. The last chapter studies trees, binary trees, binary search trees and AVL trees in short. The Java Collections API is discussed also. Obviously, this is part of the traditional content of the *Algorithms and Data Structures* predecessor course. We have tried to bring this content in a more object-oriented way, e.g., using encapsulated instance methods instead of static methods using data classes without behavior, as most books do. Time is very limited though. This part is not used when we give the course in an industrial context. We presume that participants have had a traditional *Algorithms and Data Structures* course.

At several points in the course object-oriented code is compared to equivalent implementations in traditional procedural languages, like Pascal and C. This can be important to help persons that already have a background in procedural programming. Persons that do not have such background can skip these passages. We also discuss how to validate object-oriented programs. Testing is introduced in the first chapter, and static verification at the end of the chapter on the contract paradigm. After that, with every new concept that is introduced, also how to verify it is discussed.

The principles of object-oriented programming are concretized in the course in a number of rules, which are repeated at the end of each chapter. These principles are not aimed specifically at the development of object-oriented programs in Java. We are convinced that most of the rules that are put forward are usable when developing code in other object-oriented languages, such as C++, Ada '95, Object Pascal, Eiffel or Smalltalk.

Throughout the course, students are exposed to design patterns [Gamma, Helm, et al. 1994], Java idioms and the foundations of object oriented frameworks. This provides a basis for a deeper study of these subjects later, when students have more practical experience and view these techniques in a proper context.

The book does not address interfaces, nested classes or the 4th part of the course.

Part 1: Classes in Isolation

- Introduction to classes and objects, covering encapsulation and valued semantics versus reference semantics.
- Principles of contractual programming by means of preconditions, postconditions and type invariants.
- Principles of robust programming, with a thorough discussion of exceptions.

Part 2: Object Interaction and Relationships

- Realization of single and multiple associations between objects, using package accessibility in Java.
- Study of constructors and termination and garbage collection in Java, with strategies for avoiding memory leaks.

Part 3: Inheritance

- Introduction to several extension and specialization inheritance, linked to the notions of polymorphism, static and dynamic type, and static and dynamic binding.
- Development of class hierarchies using behavioral subtyping as guideline.
- Study of Java interfaces as reverse contracts, using the vent pattern as an example.
- Nested classes.

Part 4: Algorithms and Data Structures

- The Java Collections API.
- Recursive implementations & Backtracking.
- Lists.
- Trees.

Java as the Example Language

In the first place, we teach *object oriented programming*. That the programming language Java is used is of less importance. We do believe that Java is the best programming language for most programming tasks demanded of developers in this day and age, but basic concepts of object-oriented programming are more important than, and are independent of, any given language. The concepts we study can be applied equally well to C++, Eiffel, Smalltalk, Ada '95 or Delphi. We have a C++ version, which is sometimes used when we teach the course for companies.

This approach is rather important in our opinion. Often people are trained in applying techniques or idioms, and the underlying rationale of these techniques is passed over. In our experience, it is more efficient to train people, of any level, in the academia as well as in the industry, with a deep insight in the matter. This demands for a higher initial investment of all people concerned because such a course requires more time. Our experience shows however that the return on this investment is almost immediate. The quality of the software that is produced by programmers that are well versed in the underlying concepts is much higher than that of software that is produced by programmers that merely replicate programming techniques. This results almost immediately after the training in more stable software, which needs to be revised less often, is easier to revise when it is needed, and in a shorter time-to-market. People trained this way also become less dependent of a specific programming language or environment. A person trained in object-oriented programming should be able to switch from Java to C++, e.g., quite easily. This is much more difficult for people only trained in programming techniques and idioms, because these are by nature extremely dependent on the language. The investment the training represents is consequently more save and can return longer. It is clear by now that this deep knowledge of object object-orientation will not become irrelevant soon. All new evolutions in programming, such as component-based programming and aspect-oriented programming, have their basis in object-oriented programming.

Java is most interesting because the language is particularly clean, it comes with the standard Java class library and code written in Java has a high degree of portability. Older languages such as C++, and to a lesser extend, Eiffel, do not bring these advantages. With current prices of processors and RAM, their somewhat faster execution does not outweigh these advantages.

Didactic Form

University

Originally, the course was taught in the university ex cathedra using transparencies. An early version of the book was available for the students. Every teaching session was followed by an exercise session under the guidance of teaching assistants. Each part was concluded with a larger project. The exam was traditional, i.e., a 4-hour session with a number of small questions and programming problems, evaluated in written form. 3 years ago we switched to a didactic form based on self-study, and a project based approach. Students are required to study the course text in advance. Contact sessions with smaller groups (in practice less than 30 students) are held for each chapter. Here the subject matter of the course is spitted out interactively, based on some examples. There are no more exercise sessions, but there are a number of self-test-exercises available on-line. At the end of each part there is a larger project, which students have 2 to 3 weeks to finish. There is a personal feedback session about the student's solution to the project. The exam now is another larger project, which is evaluated orally.

This change proved a great success. We have far less failures, and we feel that the students have a deeper understanding than they had before. The projects give the students the opportunity and incentive to discuss the subject matter amongst each other, and we believe that this the main reason for the higher success rate. The contact sessions make it possible for us to highlight what are the more and what are the less important issues in the course. They are intended in the first place to give students the opportunity to ask questions about the chapter they studied in advance, but frankly, this does not happen often. Whether or not the contact session has success depends on the level of interaction we can entice, and that depends heavily on group dynamics. We observe that after the first lessons, students forget to prepare for the session more and more often, but that the level of interaction rises.

Industry

When the course is taught in an industrial context, most often there is no time to take this approach. The course is taught ex cathedra using transparencies, for 10 to 20 participants. Typically, we agree to do the course in five 6-hour days. We have done the course several times on consecutive days in one week, and spread out over a period of 3 to 10 weeks. 2 to 3 hours each are spent on a larger exercise, concluding the first part, and at the end of the 4th day. There is no time in the 30 hours contracted for more exercises, and we never succeeded to convince contractors of the need for more time. Typically, we agree that smaller exercises are given as homework, which is discussed the next morning. Typically, participants do not do their homework.

These courses have been very successful, in every respect. We have taught the course for new, young employees, and for seasoned employees.

The most positive feedback we get from people with a lot of programming experience. While they are skeptic initially about the contract paradigm and the stress we put on specifications, especially using a formal syntax, they become very fascinated by the results it gives by the third day, and are raving by the end of the course. Every time they tell us that they are going to use this approach, because that's exactly what was missing in the project they are working on currently. Sadly, we don't know whether this actually happens or not. We think the training would benefit highly from an extra half day after 2 or 3 months, where trainees can ask questions and talk about the course once the dust has settled. Although the contractors usually agree to such a contact, it never actually happened due to particular circumstances. Young employees typically tend to be cockier and less receptive to our message. Still, the course is evaluated highly by the participants and the contractors. The best results we got with heterogeneous groups. Young participants are more timid when older colleagues are around, and are surprised seeing them interested. As a result they are more open to the subject matter. More experienced participants get an incentive to really think through what we bring them when they try to rephrase the subject matter to younger colleagues in response to questions, or when applying them to actual problems and projects of the company they work for. With heterogeneous groups, the dynamic is at its peak.

Most examples in the course are chosen so that the complexity is not in the example domain. Most example classes describe objects from the banking sector, like checking accounts, credit cards and savings accounts, in a simplified fashion. Because most readers will undoubtedly be acquainted with such objects, they can concentrate fully on the way in which the properties and the behavior is described. To our surprise, this causes problem with a company we did the course for a number of times. The company works primarily in the banking sector itself, and especially the seasoned employees had trouble dealing with the simplified examples we used. It was very difficult for them to pass over the "mistakes" that they found in the examples.

Future Evolution

Possible Changes in the Content

The authors have different opinions whether and how the structure of the course should evolve in the future. In the current structure, inheritance is only introduced in the last part, which is rather late in the course. In the 5-day industry version, inheritance is discussed on the 4th and 5th day. Both authors do agree that the contract paradigm needs to be introduced as soon as possible, so the earliest we can talk about inheritance is after the introduction of the contract paradigm. This means that the discussion of total methods and robustness and defensive programming, in short, exceptions, needs to move further down in the course. On the other hand we want to avoid that robustness or memory management, or bi-directional associations for that matter, appear afterthoughts.

The chapters on interfaces and nested classes actually are glued on as an afterthought at this time. Recently we changed the chapter on interfaces, and the chapter on nested classes is entirely new. The event pattern (a.k.a., Observer/Observable [Gamma, Helm, et al. 1994], or Model/View/Controller) is used as the narrative in these new chapters. These chapters are used 3 times now, and where not entirely satisfactory. The position of the chapter on interfaces, as an extension of inheritance, seems logical. Inner classes may be better discussed as a form of associations, although the technique makes little sense without inheritance. Interfaces and nested classes are not discussed in the first edition of the book.

Possible Changes in the Academic Program

A change has happened since the traditional pedagogy for programming has become general. Currently, complex algorithms and data structures to gain speed and memory efficiency, are less important than reusable, understandable code which is stable under change. The cost of a software developer is much higher than the cost of a faster or extra processor, or extra RAM. There are circumstances where speed and memory efficiency still are important, e.g., in embedded systems, or with complex, possible NP, problems, but this is no longer the mainstream of IT. To us it seems that the fourth part of the course, which is closest to the original Algorithms and Data Structures course, should be split of and offered in the last year of the university program as an optional specialization. The general program should make students aware of issues of performance and memory use, but they should not all be versed in it. They should know when to call the experts. The extra time we get in this course would be used gratefully.

On the other side of the spectrum, it might be possible to use the course as the first programming course. More time is needed to bring the same subject matter to freshmen, and an extensive introduction would be needed, but still we wonder whether the more intuitive first programming course is not a waste of time.

Such changes in the academic program are a responsibility of the faculty of course, and we do not foresee them in the near future.

References

[Blakeley, Dockx, van Dooren 2001]

Peter Blakeley, Jan Dockx & Marko van Dooren: **jutil.org**; SourceForge; Labrador, Leuven; 2001; <http://org-jutil.sourceforge.net/>, <http://www.sourceforge.net/projects/org-jutil/>

[Dijkstra 1969-1970]

Edsger W. Dijkstra: **Notes on Structured Programming; In Pursuit of Simplicity; the manuscripts of Edsger W. Dijkstra**, Ed. Texas; 1969-1970; <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF>

[Dahl, Dijkstra, Hoare, Genuys 1972]

Ed. O. J. Dahl, Edsger W. Dijkstra, C. A. R. Hoare, F. Genuys; **Structured Programming**; Vol. Academic Press; New York; 1972

[Dockx, van Dooren, Steegmans 2001]

Jan Dockx, Marko van Dooren & Eric Steegmans: **Dijkstra's Dream; Internal Iterators as Software Theorems**; Katholieke Universiteit Leuven, Dept. of Computer Science; Leuven; 2001; CW340; <http://www.cs.kuleuven.ac.be/publicaties/rapporten/>

[Dockx, van Dooren, Steegmans 2001a]

Jan Dockx, Marko van Dooren & Eric Steegmans: **Different Implementations in Java of a Nested Loop and Their Proofs**; Katholieke Universiteit Leuven, Dept. of Computer Science; Leuven; 2001; CW324; <http://www.cs.kuleuven.ac.be/publicaties/rapporten/>

[Gamma, Helm, et al. 1994]

Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides: **Design Patterns: Elements of Reusable Object-Oriented Software**; Addison-Wesley Pub. Co.; Wokingham, England; Reading, Mass.; 1994; ISBN 0-201-63361-2

[Liskov, Wing 1994]

Barbara H. Liskov & Jeannette M. Wing: **A Behavioral Notion of Subtyping**; *ACM Transactions on Programming Languages and Systems*, Vol. 16, Nr. 6; November 1994; p. 1811-1841

[Meyer 1991]

Bertrand Meyer: **Design by Contract**; *Advances in Object-Oriented Software Engineering*, Ed. D. Mandrioli, Bertrand Meyer; Prentice Hall; Englewood Cliffs, N.J.; 1991; p. 1-50

[Meyer 1997]

Bertrand Meyer: **Object Oriented Software Construction**; 2nd Edition; Prentice Hall; Upper Saddle River, NJ; 1997; 1254 pages; ISBN 0-13-629155-4

[Steegmans, Dockx 2002]

Jan Dockx, Marko van Dooren & Eric Steegmans & Jan Dockx: **Objectgericht programmeren met Java**; Acco; Leuven; 2002?; ISBN 90-334-4535-2; in publication

[Wirth, 1976]

Niklaus Wirth, **Algorithms + Data Structures = Programs**; Prentice Hall; 1976