Contract-Guided System Development

Isabel Nunes and Vasco Vasconcelos
Department of Informatics, Faculty of Sciences,
University of Lisbon
{in,vv}@di.fc.ul.pt

Abstract

We present the approach to teaching object-oriented concepts that we have adopted almost three years ago. Our integrated effort spreads over three one-semester courses, incorporated in a four year long undergraduate course, and is based on an objects-first approach coupled with contract-guided system development. Our aim is that students learn the fundamental skills to construct correct, robust, reuseable and extensible software systems. Some preliminary conclusions are presented.

1. Introduction

At the Department of Informatics, Faculty of Sciences, University of Lisbon, we have been following an objects-first approach centred in design by contract for almost three years, presenting to students a three-semester course that spreads over their first, third and fourth semesters. Prior to that, students were faced with an imperative-first approach in their first and third semesters, and only in the fourth semester were they confronted with object-oriented design and programming.

The methodology of design by contract, as introduced by Bertrand Meyer [5], for the specification and implementation of classes is present all the way through the three semesters under consideration. Students are introduced to the notion of program correctness in the very first semester; the emphasis throughout the course is on the use of contracts. Our experience tells us that the earlier students are initiated in this approach the more they get able to abstract what is really important and the smoother and effective is their acceptance and use of design by contract. Students with prior exposure to programming (imperative-first approach) tend to never really adopt the methodology.

We have chosen Java as the supporting programming language since it is used throughout the whole course for most of the programming assignments. The assertion language used to write contracts is a very important factor in motivating the students to this approach. It must be expressive enough to allow to write useful contracts, yet not difficult to master. We have picked the language of Java boolean expressions (extended with a few constructors), à lá iContract [3].

2. Course overview

The programme spreads over three semesters, each encompassing 22 to 26 hours lecturing, 22 to 26 hours tutorial, and aims at a population of over 300 students.

The first semester starts by presenting the basic concepts of any imperative programming language (types, variables, expressions, assignment). Classes and objects are then introduced, in this order. The bases laid, we introduce design by contract (assertions, pre and post-conditions, invariants, correctness of a class). Contracts accompany the rest of the three courses. The semester ends with the study of the remaining control structures, arrays and basic algorithms, streams, and exceptions. In particular, inheritance and polymorphism are dealt later, in the third semester.

Distinctive features of this first course are the emphasis on problem decomposition and design by contract. For example, several approaches to compute the distribution of deputies in a parliament following the method of Hondt are pursued. In doing so, students learn to decompose the problem (building and filling the auxiliary data structures, sorting and finding elements in arrays), and to specify the contract for each routine they write. A complete specification is obtained for the Hondt problem.

We stress three points in our approach: (i) Specification as a means to state the correctness of an implementation, while promoting non-defensive programming, (ii) Contracts as documentation for clients, and for implementors of descendant classes, (iii) Contracts can be monitored.

The second semester deals with techniques for data structuring and algorithm analysis. The OO programming methodology introduced in the first semester is followed and emphasis is put on the principles of abstraction, encapsulation and modularization. The fundamental data types – list, stack, queue, set, tree, table and graph – are studied and techniques for the evaluation of algorithms are introduced.

The abstract specification of each data type is built – sorts, functions, axioms and pre-conditions - in order to capture their essential properties without overspecifying. From these abstract data type specifications, interfaces are built: the applicative view of ADTs changes into a more operational view in which some of the data type functions are declared as commands. Contracts are written for every interface method defining the rules for any implementation of the data type. These contracts are built from the pre-conditions and axioms of ADT's in a systematized way. Specific classes are then built that implement (both statically and dynamically) the specifications. Proofs of correctness for implementations against interface contracts are discussed, both in the first and second semesters.

The third semester of the course presents a methodology [4] for the process of system development that covers the description of use cases, and an iterative OO development cycle of analysis, design and implementation: identification of concepts, contract writing for system operations, assignment of responsibilities to objects to fulfil system operation contracts (guided by general patterns of responsibility assignment) resulting in the design class model for the given iteration that can be implemented and tested before taking another iterative step in the process.

Here again the notion of contract (responsibilities, pre and post-conditions) is used, although in this case applied to system operations/events that emerge from use case descriptions. In this context contracts, together with a conceptual model that identifies and relates all the concepts that are of interest for the use case(s) in question, help us bridge the gap between an essentially process oriented view given by the use case approach, and the OO approach we pursue.

In this semester students have to deal with systems of many classes related both through clientship and inheritance. Inheritance brings a whole series of new and interesting problems like object polymorphism, dynamic binding, method redeclaration and others. The role of class contracts gains importance here: on the one hand they help us understand the nature of inheritance and on the other hand they can be used to control the change in the semantics of methods that redeclaration and dynamic binding can bring.

The change in the semantics of redeclared methods is controlled through the imposition of the "Don't ask for more; don't offer less" rule in contract writing and, consequently in method implementation, in order to be consistent with the substitution principle. Pre (post)-conditions of redeclared methods must be implied by (imply) pre (post)-conditions of the corresponding method in the super-type. Students are asked to specify contracts for classes in a hierarchy that verify the above rule and understand the importance this has for reuse and extension. They get aware that there are different approaches to contract inheritance on what concerns monitoring code generation tools: ContractJava [2] and Jass [1], for example, generate monitoring code that checks contract hierarchy by generating an exception if the above rule is violated; iContract [3] and Eiffel [5], for example, generate monitoring code that results from or-ing the pre-condition and and-ing the post-condition with ascendant pre and post-conditions of corresponding redeclared methods.

In order to complete each course students are supposed to develop one group project (usually spread over two or more assignments) and take a final exam. The group projects are intended to integrate their modelling and programming skills in the team development of a real world system.

3. Evaluation

It is still too early to have enough feedback on the programming and design skills of our students from lecturers of upper division courses where the focus is on application areas such as databases, artificial intelligence, operating systems and others. The major drawback that was pointed out by the average student was his serious difficulty in the analysis and design of a solution to any problem involving more than a couple of entities.

We are able to compare, however, present with past results in what concerns the evolution of students' skills in the three semesters that compose our object-oriented programming and design course. We feel that the change from a procedural approach to an object-oriented contract-guided one at early stages brought positive results insofar as students are not only able to reason about algorithms as before, but they do it

within the context of classes as modules of some complex system. They are constantly encouraged to decide who should be responsible for doing what, in a way that promotes software quality (correctness, robustness, reuse, extension) before deciding how to do it. We strongly feel that objects-first contract-guided approach promotes the abstraction indispensable to system development.

Students learn the limits of the language we picked for contracts in the very first semester. The main difficulty seems to be the inability to use procedures in contracts: to express that the number of days that have elapsed since a given date, we would like to write "if we advance the given date that number of days, we get the current date". Only that advancing a date is usually a procedure, hence cannot be incorporated in contracts. This problem becomes acute in the second semester of the course when students study ADT and try to find a systematic method to convert them into Java interfaces.

The lack of expressiveness of the assertion language is also felt, for example, when we want to write contracts for classes whose instances are composed of other objects (that, in turn, themselves may be composite objects) which is typically the case in the third semester of the course where students deal with systems of many classes. The task reveals itself difficult with the languages that are at our disposal: it either asks for the creation of a series of otherwise useless queries, or it brings undesired class coupling. We developed the concept of meta-assertion [6] as a possible means to surpass this difficulty.

It is important to be able to monitor assertions while running programs, to see contracts at work. In 1999 we tried a few freely available tools and decided on iContract [3]. The experience did not last the whole semester since the tool was found bug ridden, making it unusable. It must be emphasized that the target user is a first year, first semester undergraduate student, with no experience whatsoever on using any tool at all. Since then we have been using no tool. Tools may have matured now and may be it is about time to have a second look at the market (we have no intention of developing our own tool, for different reasons). The absence of a tool that students may use is really distressing, for students have no means to check their assertions, not even the syntax. We have witnessed pre-conditions written in plain Portuguese, and since these are incorporated in the header (** comments) of routines and classes, the usual javac ignores them.

References:

[1] Bartezko, D., Fischer, C., Moller, M. and Wehrheim, H., Jass-Java with assertions, Workshop on RunTime Verification, 2001. Held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01.

[2] Findler, R.B. and Felleisen, M., "Contract Soundness for Object-Oriented Languages", OOPSLA 2001.

[3] iContract HomePage: http://www.reliable-systems.com/tools/iContract/iContract.htm.

[4] Larman, C., "Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design", Prentice-Hall PTR, 1998, ISBN 0-13-748880-7.

[5] Meyer, B., "Object-Oriented Software Construction", Prentice-Hall PTR, 1997, ISBN 0-13-629155-4.

[6] Nunes, I., "Design by Contract Using Meta-Assertions", submitted.

Biography:

Isabel Nunes is an assistant professor at the Department of Informatics, Faculty of Sciences, University of Lisbon since 1998 where she teaches in the area of programming languages since 1988. Lectured courses include OO Programming, Imperative Programming, Semantics of Programming Languages, Computer Graphics. She cooperated with Vasco Vasconcelos and José Luiz Fiadeiro in the definition of the curriculum of this three semester OO course.

Vasco Vasconcelos is an associate professor at the Department of Informatics, Faculty of Sciences, University of Lisbon where he has been teaching since 1996. He has lectured several courses in the area of programming languages, including OO Programming, Functional Programming, Algorithms and Data Structures, Compiler Construction, Concurrent Programming. He has participated in the restructuring of the three-semester course on OO design here described, for the academic year of 1999/2000.