A Formal Approach to the Teaching of Abstract Data Types

Laura Felice, Liliana Martinez, and Claudia Pereira
Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina

Lfelice@exa.unicen.edu.ar lmartine@exa.unicen.edu.ar
cpereira@exa.unicen.edu.ar

Abstract

In this paper we present a methodology for the teaching of programming applied to an elementary course of the System Engineering career at the Universidad Nacional del Centro de la Provincia de Buenos Aires. This methodology starts with the formal specifications of abstract data types and concludes with an implementation of an efficient algorithm in C++ language.

We describe the methodology, and a case of study showing the proposed methodology.

Keywords: algorithm design techniques; formal specifications; programming teaching.

Introduction

With the emergence of structured programming language in the 60s, the concept of data type, defined as a set of values serving the domain of some operations appears. In these languages (C, Pascal and others, all of them Algol derived) data types allow to classify the program objects i.e. variables, parameters and constants. This notion was insufficient to the large-scale software development, since the use of data into the programs ignores other restrictions than the compiler imposed, causing inconveniences in new types users defined. To solve this problem (in the middle of 70 s) several authors (like S.N Zilles, J.V Guttag, Gigyebm Thatcher, Wagner, Wright, etc) introduced the abstract data type (ADT) concept, considering that a data type is not only the set of values characterizing it but also the operations that handle it. All of these operations must verify the properties that will determine the unique behavior. These authors observed the need to employ a formal notation to describe the operations behavior, not only to avoid any ambiguous interpretation but also to identify the mathematics model denoted by the ADT (Franch Gutierrez, 1994).

A software curriculum should involve elements or concepts that reflect a trend towards distinguishing the true software professional from the occasional programmer. These trends have important consequences for universities. What matters is teaching the students fundamental ways of thought that will accompany them throughout their careers and help them grow in this ever-changing field.

As Bertrand Meyer analyses in (Meyer 2001), a software curriculum should involve five complementary elements:

- Principles: lasting concepts that underlie the whole field;
- Practices: problem-solving techniques that good professionals apply consciously and regularly;
- Applications: areas of expertise in which the principles and practices find their best expression;

- Tools: state-of-art products that facilitate the application of these principles and practices; and
- Mathematics: the formal basis that makes it possible to understand everything else.

In this paper we present a methodology, based on the above points, for the teaching of analysis and design algorithms at the elementary course of the System Engineering career. This methodology starts at the abstraction level defining the problem domain and identifying the ADT intervening. Then, they are formally specified and finally, at the implementation level, the algorithm and the formal specification of data types are implemented in C++ language.

Background

In our career, the computer science curriculum has several programming language used to teach at three levels. In the first course the principles of structured are taught and Pascal is the programming language used.

In the second course we center the matters on the Design and Analysis of the algorithms. The students identify the ADT that intervene in a problem and give a formal specification of them. The specifications allow us to describe the object classes' behavior in an abstract way independently of their implementation. Formal specifications allow us to define an ADT in a precise way, avoiding any ambiguity that can be present when using informal specifications.

In order to teach how to construct object classes hierarchy, concepts of client and inheritance relationships are introduced.

The class specifications are integrated with algorithms linked with different design techniques like Divide and Conquer, Greedy, Dynamic Programming and Backtracking (Cormen 1990).

Our goal is to teach the algorithms as simply and directly form as possible. The programming language C++ allows us to do this. In the same manner, GSBL (Clérici 1988) formal language supplies all of the formal concepts taught in this course. This language brings an approach for the incomplete specifications construction, that is, specifications that describe partial aspects of the problem to solve.

The advantage of using C++ is that it is widely used and for its hybrid object oriented language characteristics offers the possibility to work with object classes that belong to the problem domain and functions that manipulate them in an independent way.

At the third level, the course of object oriented programming introduces the basic and advanced concepts of the paradigm. They use Smalltalk and Java. When learning object oriented programming, students get the knowledge to represent problems in terms of objects interacting in client/server relationship; to classify the problem concepts according to different relationships, and to distinguish the object oriented language behaviors.

Also, it is possible the incremental construction of specifications reusing components early designed to solve other problems.

Methodology

This work presents a methodology for the teaching of the programming in the analysis and design area of algorithms, oriented to students of the initial levels of the career. Throughout this course the students will solve practical exercises applying the different techniques of algorithm design: greedy, divide and conquer, backtracking, dynamic programming, etc.

The proposed methodology starts with specification levels that formally describe a problem, independently of the representations of the data types and of a particular language. It concludes in an implementation level with efficient programs written in C++. We define two levels: the abstraction and implementation levels.

At the abstraction level, the problem domain is defined and the entities that intervene in the problem are identified. In the first stage, the classes of objects are identified and they are algebraically specified. This formalism allows defining objects, classes of objects as well as its operations in an abstract way and independently of its implementation, that is to say in a non-operational way. The functionality of the operations and their semantics through algebraic properties are defined. In the second stage, the algorithms that work on the classes of objects in the domain are defined.

In the implementation level, the formal specifications are translated to C++ code and the algorithms that intervene in the solution of the problem are implemented. The C++ language, for its hybrid object-oriented language characteristic, offers the possibility to work with object classes and functions independently. It allows modeling a problem defining the object classes that belong to the problem domain; and, on the other hand, the processes (algorithms, programs) that manipulate them.

According to our experience, we can say that applying this methodology, the students can:

- acquire a high abstraction level of thinking, distinguishing the essential from the auxiliary in a class specification,
- distinguish from specifications to implementations
- learn throughout the practices the ability to decide which information will be hiding or be visible (information hiding)
- increase the reuse of specifications, the primary motivation to reuse software specifications is to reduce the time and effort required to build specifications of software systems (Krueger, C. 1992).
- use the recursion as a powerful mechanism to understand the behavior of the data type functions. When the students have learned to use recursion properly, they have gained a powerful intellectual tool.

The GSBL language

GSBL offers an approach for the construction of incomplete algebraic specifications, this is, specifications that only describe aspects of the problem to solve partially. This focus is based on adding to the design of specifications a new dimension calls vertical that expresses the process of completing specifications. With this refinement type, a new structuring relationship appears that is conceptually bound to the subclass notion in the object-oriented languages.

The goal of this language, apart from the possibility of working with incomplete specifications, is to foster the incremental construction of specifications reusing previously designed component for the solution of other problems.

The fundamental design principles of this specification language are the following: incomplete specifications, genericity, inheritance and mechanisms of powerful binding. The objects of the specification environment in GSBL are denominated classes. The syntax of a GSBL class is defined by means of the scheme showed in the figure 1.

In GSBL strictly generic components can be distinguished by means of explicit parameterization. The explicit generic parameters of the class are presented in the <*parameter_list*> list and it is also indicated if some of them is restricted to a specific class, or to a subclass of this.

The syntax of a complete class can include the *basic constructors* clause that refers to generator operations.

The *over* and *subclass* clauses define the imported and inherited specifications respectively. The over relationship corresponds to both the enrichment construction and the relationship "based_on" in the object-oriented languages. The *subclass* clause builds the new specification starting from each of the specifications of the <*superclass_list*> list. The *subclass* relationship involves an inheritance mechanism that can be multiple, in which case the resulting class is the fusion of its superclass. The *subclass* relationship corresponds with the "is-a" relationship in the object-oriented languages.

The *with* and *defines* clauses add new sorts, operations or equations. Whereas in the clause *with* this enrichment is incomplete, there are not enough equations to define the behavior of the new operations or there are not enough operations to generate all the values of a given sort, in *define* clause they are totally defined.

Ops define the functionality of type operations, and *Eqs* express the operations semantic through a set of axioms that are well formed formula over terms of first order predicate calculus.

A complete and detailed description can be found in (Clérici 1988).

CLASS class_name [parameter_list]
OVER < over_list >
SUBCLASS OF < superclass_list >
BASIC CONSTRUCTORS < constructor_list >
WITH
SORTS <sort_list>
OPS < operations_list >
EQS < variable_list > < equation_list >
DEFINE
SORTS < sort_list >
OPS < operations_list >
EQS < variable_list > < equation_list >
END_CLASS

Figure 1. GSBL syntax.

A Case Study

A simple example that shows the proposed methodology is presented below.

The Abstraction Level

The power of ADT specifications comes from their ability to capture their essential properties without overspecifying. For this reason, formal specifications are introduced in an early stage of our career to develop the student's abstract and formal reasoning.

Let us suppose we have the hierarchy showed in the figure 2 that was specified to solve a previous problem, based on the following considerations:

- *Collection* is a group of elements of the same nature,
- *Set* is a Collection of elements without copies,

- *Sequence* is a finite Collection of 0 (empty sequence) or more elements, which are ordered lineally. A sequence can be defined as the added of an element (for right or left) to a sequence already existent. A sequence ***s*** can be written:

  $$(s_0, s_1, s_2, \ldots, s_{n-1}, s_n)$$

  where s_0 is the element to the furthest more left of the sequence and s_n is the furthest one of more right, and s_{i+1} is the following one of s_i and s_i is the previous of s_{i+1}.
- *Queue* is a sequence that serves to pile up and retrieve elements in a first-in, first-out (FIFO) manner.

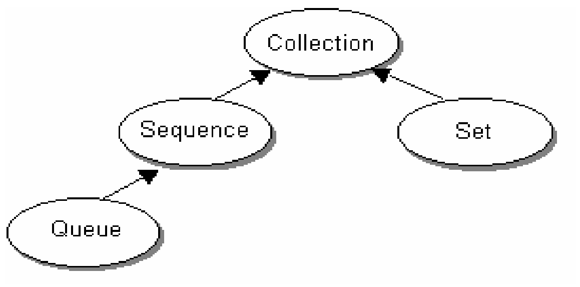


Figure 2. Collection hierarchy

Part of this hierarchy specified in GSBL language is shown in the following figure 4.

Now, an object stack is specified to solve a new problem, which can be achieved reusing existent components. Using the above hierarchy the Stack data type could be specified as subclass of Sequence (see figure 3), considering that a stack implements the LIFO political (last-in, first-out) where the last element entered in the stack is the first one to be removed. The stack specified in GSBL is shown in the Figure 5.a.

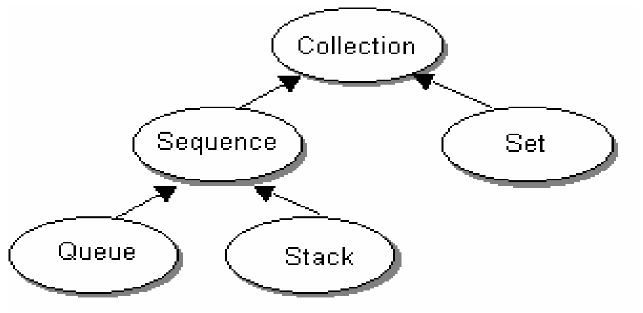


Figure 3. Hierarchy specified in GSBL

This stack formal specification expresses all there is to know about the notion of stack in general, excluding anything that only applies to some particular representations of stacks. A method relying on the physical representations of data structures to guide analysis and design would not be likely to yield flexible software.

The Implementation Level

A class is an ADT equipped with a possibly partial implementation (Meyer 1997). The definition states that the implementation may be partial, a class which is fully implemented is said to be effective, and

```
CLASS Collection [item:ANY]
OVER Natural, Boolean
BASIC CONSTRUCTORS make, add
WITH
        SORTS Collection
        OPS
        make: -> Collection
        add: Collection x item -> Collection
        size: Collection -> Natural
        count: Collection x item -> Natural
DEFINE
        OPS
        empty: Collection -> Boolean
        belong: Collection x item -> Boolean
        EQS {c: Collection; e, e1: item }
        empty(make) = TRUE
        empty(add(c,e)) = FALSE
        belong(make, e) = FALSE
        belong(add(c,e), e1) = IF (e = e1) THEN TRUE
                                    ELSE belong(c, e1)
END_CLASS


CLASS Sequence[item:ANY]
SUBCLASS OF Collection [length  : size]
BASIC CONSTRUCTORS make, add
WITH
        SORTS Sequence
        OPS
        make: -> Sequence
        add: Sequencex item -> Sequence
        first: Sequence(s) -> item
                pre: NOT empty(s)
        rest: Sequence(s) -> Sequence
                pre: NOT empty(s)
DEFINE
        length : Sequence-> Natural
        count: Sequencex item -> Natural
        insert:Sequence(s)xitemxNatural(n) -> Sequence
                pre: length (s) >= n-1
        delete: Sequence(s) x Natural(n)  -> Sequence
                pre: length (s) >= n
        get: Sequence(s) x Natural(n)  -> item
                pre: length (s) >= n
        EQS {s: Sequence; e, e1: item; p: Natural}
        length  (make) = 0
        length  (add(s,e)) = 1 + length  (s)
        count(make, e) = 0
        count(add(s,e), e1) = IF (e = e1)
    THEN 1 + count(s,e1)  ELSE count(s, e1)
        insert(make,e1,1) = add(make,e1)
        insert(add(s,e),e1,p)=IF(p = length (add(s,e))+1)
           THEN add(add(s,e),e1)
           ELSE add(insert(s,e1,p),e)
        delete(add(s,e),p) = IF (p = length (add(s,e)))
   THEN s      ELSE add(delete(s,p),e)
        get(add(s,e),p) = IF (p = length (add(s,e)))
   THEN e      ELSE get(s,p)
END_CLASS


CLASS Queue [item:ANY]
SUBCLASS OF Sequence[undefine:count, un-
define:insert, undefine:delete, undefine:get, put : add,
remove: rest]
BASIC CONSTRUCTORS make, put
DEFINE
        SORTS Queue
        make, add
        top : Queue -> item
        remove: Queue -> Queue
        EQS
        {c: Queue; e:item}
        top (put(make,e)) = e
        top (put(c,e)) = top (c)
        remove (put(make,e)) = make
        remove (put(c,e)) = put (remove (c),e)
END_CLASS
```

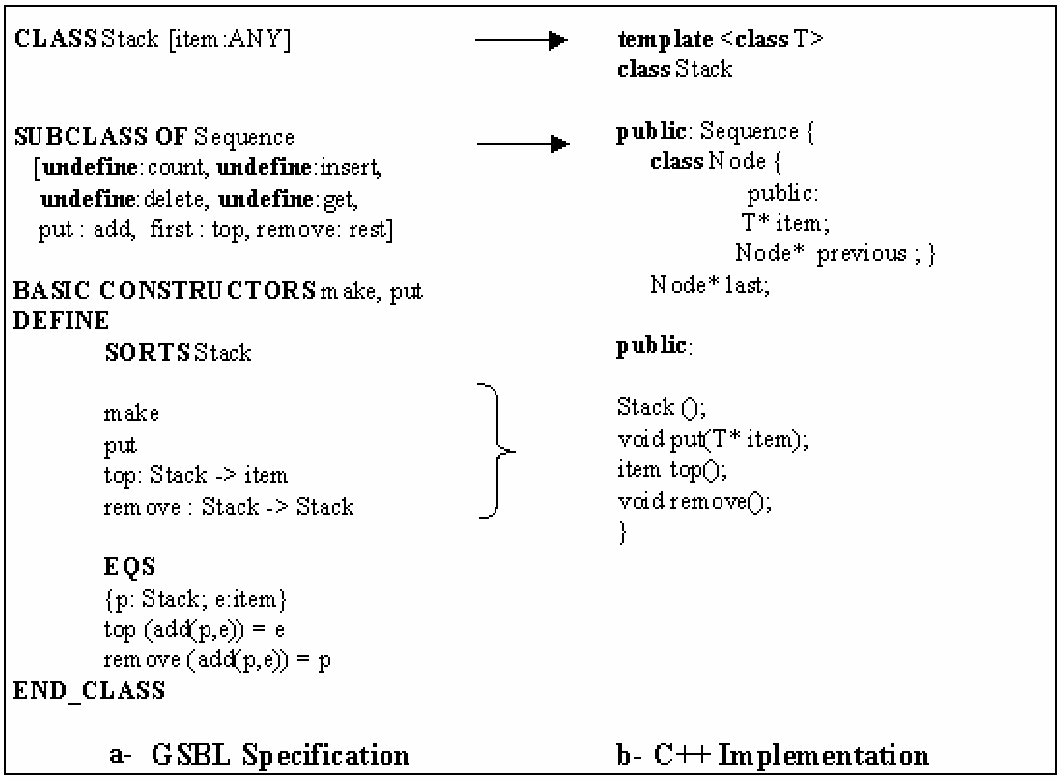
Figure 4. GSBL Specifications

those implemented only partially is said to be deferred. In the abstraction level they correspond with the complete and incomplete specifications, respectively.

Therefore, to obtain a C++ class the student must provide an ADT and decide on an implementation. The ADT is a mathematical concept; the implementation is its computer-oriented version.

Several possible physical representations exist for stacks. A linked representation was selected. The GSBL specification of the Stack type and a possible implementation in the object-oriented language C++, reflecting the correspondence between some of the clauses in the abstraction and implementation levels, is shown in the figure 5.b.

As a last step, we have the integration of the ADT implementation with the algorithmic schema implemented in C++ language. The schema is linked to different techniques applied in this course.



```
CLASS Stack [item:ANY]
SUBCLASS OF Sequence
  [undefine:count, undefine:insert,
   undefine:delete, undefine:get,
   put : add,  first : top, remove: rest]
BASIC CONSTRUCTORS make, put
DEFINE
        SORTS Stack

        make
        put
        top: Stack -> item
        remove : Stack -> Stack

        EQS
        {p: Stack; e:item}
        top (add(p,e)) = e
        remove (add(p,e)) = p
END_CLASS
```

```
template <class T>
class Stack

public: Sequence {
   class Node {
              public:
             T* item;
            Node* previous ; }
   Node* last;

public:

Stack ();
void put(T* item);
item top();
void remove();
}
```

a- GSBL Specification b- C++ Implementation

Figure 5. Stack Specification and Implementation

Conclusions

The teaching methodology is based on a design discipline through the formal specification of ADT and a discipline of development through rigorous programming techniques.

It allows us to construct algorithms in an independent way of a particular language, to guide the implementations choices for object classes involved. In this way it is possible to introduce proper topics to a basic course of Analysis and Design of Algorithms (formal specification, algorithm design techniques, and a vast classic algorithms) in a framework that allows us to include object oriented notions, formal specifications and components reusability.

According to our experience, we can say that applying this methodology, the students will achieve:

- the abstraction reasoning,
- the mechanisms of specify formally, make reuse and the use of recursion
- the principles of object oriented programming.

References

Clérici, S and F. Orejas (1988). *"GSBL: an Algebraic Specification Language Based on Inheritance"* Proc. of the European Conference on Object-oriented Programming (ECOOP 88) pp 78-92.

Cormen, T., Lierserson, C. and Rivest, R. (1990) *Introduction to Algorithms*. MIT Press.

Ellis,M.A. y Stroustrup, B. (1990). *The Annotated C++*. Reference Manual. AT&T Bell Laboratories, Murray Hill, New Jersey.

Franch Gutierrez, Xavier. (1994). *Estructuras de Datos: Especificación, diseño e implementación*. Ediciones UPC, Universidad Politécnica de Catalunya.

Hennicker, R. , Wirsing, M.(1992) *A Formal Method for the Systematic Reuse of Specification Components*. Lecture Notes in Computer Science 544, Springer-Verlag, Berlin.

Krueger, C. (1992) , *Software Reuse*. ACM Computing Surveys : 24 (2), 131-183.

Meyer, Bertrand (1997). *Object-oriented Software Construction*. Prentice Hall.

Meyer, Bertrand (2001). *Software Engineering in the Academy*. IEEE Computer Society. Vol. 34 Number 5. pp: 28-35.

Biographies

Laura Felice is an assistant professor at Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires (UNCPBA), Tandil, Argentina. She is currently doing her MSc. thesis and her research interests are focused on Formal Software Development and Reuse. Actually she is assistant of Design and Analysis of Algorithms course. She is a member of the Technology Software Group of the INTIA at the UNCPBA.

Liliana Martinez is an assistant professor at Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires (UNCPBA), Tandil, Argentina. Her research interests are focused on Formal Specifications. She is a member of the Technology Software Group of the INTIA at the UNCPBA. She is assistant of Design and Analysis of Algorithms course.

Claudia Pereira is an assistant professor at Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires (UNCPBA), Tandil, Argentina. Her research interests are focused on Formal Specifications. She is a member of the Technology Software Group of the INTIA at the UNCPBA. She is assistant of Design and Analysis of Algorithms course.