Computación y Sistemas Vol. 4 No.3 pp. 230 - 241
© 2001, CIC - IPN. ISSN 1405-5546 Impreso en México

An Object-Oriented Approach for Learning of Algorithm Design with Sequential Devices and Schemas

Rafael M. Gasca, Juan A. Ortega and **Miguel Toro**
Departament of Languages and Computer Systems
Seville University, Avda. Reina Mercedes s/n 41012
Sevilla, Spain.
e-mail: {gasca, ortega, mtoro}@lsi.us.es

Article received on October 03, 2000; accepted on January 15, 2001

Abstract

This article proposes the learning of algorithm design using a metaphoric model. This model allows us to structure the knowledge of this domain by mapping onto it concepts and relations from an existing and already familiar domain, input/output devices and schemas.

Abstract design techniques are used to develop a constructivist view of learning of the students to solve problems algorithmically. These techniques have a common problem solving strategy that can be applied to many problems. The aim is the identification of structural similarities among problems and the application of design patterns.

The single most important design technique is modeling, the strategy of abstracting a messy real-world application into a clean problem suitable for algorithmic attack. This article presents a construction medium to algorithm design by means of an object-oriented pattern. The constructs are input/output sequential devices and a well-constructed library of sequential schemas. The application of the sequential schemas or combination of schemas to these devices allows the learning of a robust methodology in order to solve a broad range of diverse problems.

Keywords: Object-oriented model, Constructionist Learning, Algorithm design, Sequential schemas.

1 Introduction and Motivation

A good algorithmic designer, rather than starting from scratch to produce a new algorithm for every problem, knows how to look for known patterns that serve as a starting point to enable the use of an existing algorithmic design.

The teaching of problem solving by means of algorithmic design, actually considers every problem as a different problem and then the student learns to develop and to implement different algorithms. The results of this teaching and learning shows serious shortcomings. In order to improve these results and produce the conceptual change in the students that learn it, we have reduced the design process to a creative sequence of steps and templates that the students know, what provides a path to take from the initial problem statement to a reasonable solution. The student then looks for the set of schemas and devices that can be used to solve the problems, rather than the individual instructions of a particular language.

The constructionist approach to learning is proposed in the bibliography (Papert, 1991). The name constructionism derives as a variant of the related psychological school of *constructivism*. The processes of learning are considered as both active and creative. The learner must discover structures of knowledge, which are original ideas, at least relative to him. The active and creative role of the learning process is based on the idea that students should be able to use input/output devices and schemas as a construction medium. These previous ideas of the students can allow a significant learning in algorithm design. Constructivism can serve for discussion of issues in computer science education(Ben-Ari, 1998) and

software engineering education. In the last years, several articles have considered the constructivism in teaching of the development and estimation of software projects (Ramos *et al.*, 2000), study of the difficulties that students encounter when learning Java (Fleury, 2000), the implementation of a constructivist model for learning programming (Gibss, 2000) and the investigation of the cognitive processes in students dealing Data Structures (Aharoni, 2000).

Our main idea accepts that knowledge is not "learned", rather, it is constructed. Then the proposition of concrete problems allows us:

- To facilitate the students the production of relevant knowledge.
- To provide a diverse set of devices and schemas as alternative conceptions.
- To facilitate conceptual change in algorithm design problems.

During software development there are reusable solutions to recurring problems, it is named *software patterns* (Gamma *et al.*, 1994). In their catalog of patterns, *Iterator* pattern is defined as an interface that declares methods for sequential accessing the objects in a collection. We use this pattern in the creative process. In more intuitive way we have named it as *input sequential device(ISD)*. Also, we have defined a device to contain the output information, that is named *output sequential device(OSD)*. The sequential treatment of an *ISD* through a schema or corresponding combination of schemas, permits the ease of learning of problem solving for many families of problems by means of algorithmic designs. The solution of the problem is put in an *OSD*.

This article assumes that the students already know methods of modeling, specifying and implementing data structures, in a basic pseudocode. Also, we consider that the foundations of the object-oriented paradigm are known. This paradigm is a useful way to produce quality software and "the method which leads to software architectures based on the objects every system manipulates" (Meyer, 1999).

The abstractions lately used in the teaching of the algorithm design are based on schemas (Burgos *et al.*, 2000) and patterns (Proulx, 2000). In this work, we use sequential schema. It is considered as a behavioral template that specifies the common procedural abstraction influencing the abstract sequence. The sequential schema applies specific actions during a period of finite time and it is capable of responding to a specific behavior. In this article, the only actions considered are those in which a single action is in progress at any time. Until one action ends, the other cannot begin. The individual actions can be executed one or more times.

Moreover, the sequential schema encloses an abstract algorithm described by means of abstract operations provided by the *ISD*. The sequential schema is specified by abstracting the common properties from the problems and ignoring its non-relevant details. A concrete algorithm can be seen as a concrete instance of a schema or combination of schemas. An abstract sequence is necessary when the students apply these schemas. We say that a finite set of objects of the same type is organized in the form of a sequence if it is possible to define the following operations:

The first object of the sequence that will permit the subsequent access to the other objects of the sequence.
The next object of a given object, that will allow us to accede to an object through the object that precedes to him, and the object *i* is reached across the *i-1* objects that preceded to him.
The operation that defines the last object of the sequence.

The above operations permit the **sequential access** to the objects of a sequence. The treatment through this type of access is designated **sequential treatment.** In this work, we propose an iterative design technique for the sequential treatment.

At this point, we propose that the application of sequential schemas accomplish the sequential treatment. It consists of the following:

To determine the basic objects those define the sequence. This determination permits the best comprehension of the problem at the highest possible abstraction. It reduces the complexity of the problem.
To detect or induce a sequence structure in the set of objects to treat. For it is necessary to propose the operations of sequences.
To select a schema among the basic schemas or combination of schemas most adequate to treat the objects of the sequence.

This development of algorithmic design strategies has advantages and disadvantages when taught this way. The main advantage is that the way of integrating the elements of an algorithm is not as basic instructions but through more abstract entities, and furthermore they have been verified such that they will produce correct programs. Also the development of the basic operations on the sequences will have to be defined for each particular implementation at the highest abstract level of the same way. Nevertheless this technique does not guarantee a solution for all problems and the obtaining of solutions can be little efficient.

In this article, a specific notation based on precondition and postcondition will serve to describe what an algorithm must solve. The precondition is an assertion that expresses the properties that must be satisfied whenever the algorithm is used and the postcondition is the assertion that describes the properties that the algorithm guarantees when it returns, supposing that it satisfied the precondition. We usually use the main *Z* constructs and logic predicates for the specification of these assertions. The *Z* notation has been chosen by its clarity and utility.

The operational specification or implementation will serve to express how to solve problems. We will use a basic pseudocode.

2 Sequences

2.1 Characterization

In order to learn the sequential treatment, we need to characterize the **sequences.** Let there be a set of objects of type *T*, then is defined the set of sequences of *T* denoted as *S*:

$$\langle\rangle \in S$$
$$s \in S:\ e \in T:\ \langle e\rangle + s,\ s + \langle e\rangle \in S$$

Where $\langle\rangle$ stands for the empty sequence, and the rest of the sequences can be defined as the result of adding an object belonging to the set of objects of type *T* (by the right or by the left) to an already existing sequence. The introduction of the empty sequence in the sequence concept can carry two possibilities of treatment that depends on considering the case of the empty sequence as a particular case, or integrating it in the general case.

The sequences are represented enclosing its objects between the symbols < and >. The objects of the sequence are put in the order that are found in the sequence and separated by commas. With the purpose of accomplishing concise and legible specifications, we use special predicates that permit us to reduce the complexity of the precondition and the postcondition.

2.2 Modeling

Many possibilities exist on modeling the sequences. The determination of a model depends on the type of operations that would likely be specified.

We propose to model the sequences as a set of numbered boxes, beginning with number 1 and ending with the cardinal number of the sequence. The sequence is defined as a finite set of objects of the same type, put in an order that has a cursor that can take the positions from 1 to the cardinal position of the sequence plus one. In this way, it is easy to detect when the sequence is finished and when the sequence is empty. A pointer designates the sequence interest point and it point at an object distinguished within the same. It serves as a reference for the operations and the task of changing from an object to another object. In order to accomplish the specification of a sequence, the following attributes are considered:

s: sequence of *T*
i: Integer

Where *i* is the position where the interest point is found and *T* is the type of objects of the sequence.

3 Sequential devices

The students have to solve many problems by means of algorithms. In these problems there is input/output information, and it can be abstracted through what is designated as *sequential device*. The following problem may be an example:

Write an algorithm that generates a file with the sum of the squares of the positive prime numbers that end in 7 and are less than 100.

In this example, the input sequential device is a sequence that is formed by the positive prime number less than 100 or by the positive prime numbers ending in 7 and less than 100, and an output sequential device that is a file with a single number. This number is the result of the sum of the squares of the prime numbers of the initial sequence.

The resolution of the problem will consist then of applying the corresponding sequential schema to an input sequential device to obtain the results that are then put in an output sequential device. Our object-oriented approach provides *ISD, OSD* and *Schemas* that can be used in a creative process. It permits the students to solve complex problems easily. This form of constructionism realistically reflects how real-world learning takes place.

3.1 Input Sequential Device, ISD

3.1.1 Definitions and Operations

In this section we indicate how the learners must define an input sequential device. An *ISD* is defined as any programming

element whose behavior is similar to a sequence in a given abstraction level, and that from an operational point of view, it is defined the following operations:

- **Create:** It permits to construct the sequential device and places the interest point in the first position.
- **Next:** It permits to obtain the object where is found the interest point and to pass to the following position as long as is not found in the last position plus one.
- **HasMoreObjects:** It returns true if there are more objects in the sequence and false when the interest point is in the last position plus one.

With these operations, the sequential treatment of any input sequential device can be easily accomplished.

3.1.2 Modeling and specification of the ISD

The *ISD* uses the sequence with an interest point as the element of modeling and specification. It is modeled as an entity formed by the following attributes:

- A sequence of objects of the type *T* that is denoted as *s*,
- An Integer value that is designated by *i* that is the position of the interest point,
- A Boolean value *m* that is false when the end of the largest sequence is reached,
- A value *v* of type *T* that is an object at the position of the interest point.

Therefore, an *ISD* is an object that has four fields:

s: *Sequence of T*
i : *Integer*
m: *Boolean*
v: *T*

The invariant of the *ISD* asserts that it must be held by all operations of the device. This represents an integrity constraint added implicitly to all the operations defined on them. For an *ISD* is:

$$i \in \{1..\#s+1\} \wedge m = (i \leq \#s) \wedge (i \leq \#s \Rightarrow v = s(i))$$

Where #s represents the number of elements of the sequence s. This invariant indicates that the attribute *s* does not vary upon applying any one of the operations.

According to this model, the specification for the *ISD* remains as:

```
Specification ISD
  Parameters
    Types: T
  Attributes
    s: Sequence of T
    i: Integer
    m: Boolean
    v: T
  Invariant
    i ∈ {1..#s+1} ∧ m =(i≤#s) ∧ (i ≤#s ⇒v=s(i))
  Operations
    function Next( ) out r: T
      Pre: {m = true}
      Post: {r=v ∧ i' = i+1}
    function HasMoreObjects( ) out b: Boolean
      Pre: {true}
      Post: {b = m}
EndSpecification
```

3.1.3 Implementation of the ISD

The implementation of the *ISD* is carried out by means of the object-oriented programming paradigm, with the double purpose of constructing reusable and quality software. The reusability is considered as the ability of producing software components that can be used in different applications that produce correct and robust software.

The input sequential devices are implemented by means of an abstract class, that we have designated *ISD*. In principle, an *ISD* has two attributes, an object *v* of the sequence of type *T* in which the interest point is found and a Boolean element *m*, that is set to false when the device has no more elements.

The implementation of an *ISD* by means of an abstract class is:

```
Abstract Class ISD
  Parameters
    Types: T
    Attributes
      m:Boolean
      v:T
    Operations
      function HasMoreObjects( ) out b:Boolean
      begin
        b:=m
      end
      function Next( ) out r:T
EndClass
```

The objects that form a part of the sequence may be represented in memory (for example sequential files, vectors, etc.), however they also may be "*calculated*" and we have one element present in memory constantly and the rest are deduced through the corresponding calculation (prime numbers). In some cases, there is an intermediate situation with all the objects in the memory; the treatment order is defined through a calculation (traversals of trees).

The inheritance mechanism allows us to extend the functionality of the *ISD*. In order to accomplish a particular implementation of an abstract sequence as *ISD*, we must define

the *Create* procedure for each one of the possible entities. The class invariant must hold upon instance creation. Also the operations not implemented in the abstract class should be implemented. In the case of data structures, their corresponding interfaces can be used to implement these operations. This can also be accomplished with prime numbers, files and standard input, by using the basic operations that provide each programming language or by defining them adequately. The hierarchy of classes is shown in Figure 1.

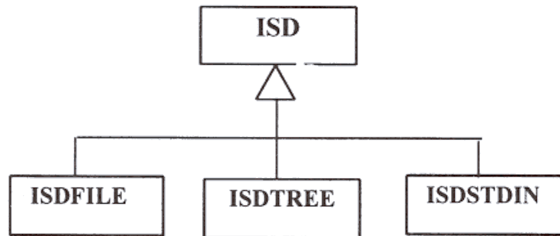


Figure 1: Hierarchy of *ISD* classes

In the classes that derive from the abstract class *ISD*, we define the corresponding operations of the *ISD* that will permit the sequential treatment by means of schemas. The following example implements these operations for a determined input sequential device.

3.1.4 Example

ISD of positive prime numbers less than a value *maxnp*

```
Class ISDNPRIME extends ISD
 Attributes
  mp: Integer
 Operations
   Create ISDNPRIME(in maxnp:Integer)
    begin
      if maxnp ≥1:
        mp:=maxnp
        v:=1
        if mp≥2:
           m:=true
         else
           m:=false
         endif
       else
        Default ISDNPRIME
      endif
    end
   function Next() out r: Integer
    begin
      r := v
      v := v+1
      while NOT IsPrime(v) and v<mp
        v := v+1
      endwhile
       if v ≥ mp:
          m := false
      endif
    end
EndClass
```

The *IsPrime* function checks if the parameter *v* is a prime number or no.

Multiple *ISD* objects can be used at the same time. It is possible to traverse a same data structure in different ways what will provide different *ISD* objects. The modifications in a data structure can cause problems, while a schema is traversing it. In order to avoid these problems, we consider that these modifications are not possible.

3.2 Output Sequential Devices, OSD

3.2.1 Definitions and Operations

These devices permit to put the results of the application of a sequential treatment on a device previously created. From an operational point of view, this has been defined in the following operation:

- **Write:** It permits an object to be placed in the position after the last one of the sequence.

According to this model, the specification for the output sequential device would be as follows:

```
Specification of OSD
Parameters
   Types: T
Attributes
  s: Sequence of T
  Operations
   procedure Write(v:T)
    Pre: {true}
  Post: {s'=s+v}
EndSpecification
```

In the above specification it is assumed that the operation *Write* can be accomplished as many times as is needed, unrestricted of space in memory.

3.2.2 Implementation of the OSD

The implementation of the Output Sequential Device will be made same way as the input sequential device was made previously:

```
Abstract Class OSD
  Parameters
     Types: T
   Operations
     procedure Write(in v:T)
  EndClass
```

3.2.3 Example

We suppose that sequential files have an operation *Fwrite(v:T)* when they are opened in a *write* way. This operation writes the object *T* in the file. We use it for the following implementation of the device.

```
Class OSDFILE extends OSD
Attributes
   f: File of T
 Operations
 Create OSDFILE (in i:IDFILE)
  begin
     f:= File(i)
     if f.Exists()
        f.Open(add)
     else
       f.Open(write).
     endif
  end
  procedure Write(in v:T)
  begin
     f.Fwrite(v)
  end
 EndClass
```

The students can construct a library of the most important devices to the sequential treatment. They may be data structures (tree, list, etc.) and other more specific devices (standard output, buffer, etc.)

4 Sequential Schemas

4.1 Definition and Classification

A Sequential Schema remains configured as a finite set of instructions that permit a connection to be established between an *ISD* and an *OSD* of such way that through their use, the students apply correct software to solving problems adequately. Figure 2 shows as the application of an adequate schema to an *ISD* object allows us to obtain a desired *OSD* object. In this figure, every arrow indicates an objects stream.

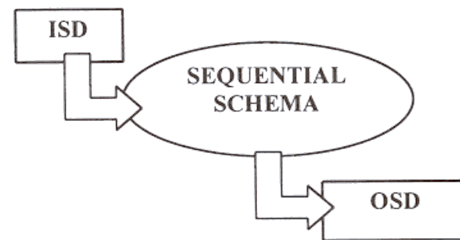


Fig 2: Diagram of the schemas application

The schemas can be basic or combinations of them. They always need at least one *ISD* and one *OSD* to work appropriately. The basic schemas that we propose for the learning of algorithm design are:

- Identity
- Accumulative
  - Counter
  - Maximum
  - Minimum
  - Sum
  - Product
  - Other (Median, Arithmetic Mean,etc.)
- Searching and property checking
- Filter
- Mixer
  - Single
  - Conditional

In following sections we will explain the previous sequential schemas.

4.2 Modeling of Sequential Schemas

The modeling intends to capture the static structure of the schema, showing the elements that compose it in order to obtain an intuitive representation. These models are valuable to document the structure of a system.

The classes that represent the *ISD* and *OSD* are parameterized by the types of the basic objects that compose the device. Different quantifiers, predicates and expressions that depend on the different treatments that the learners wish to accomplish parameterize *Schema* class. That is:

- **Quantifiers:** They determine the types of operation that are executed in the sequential treatment.
- **Predicates**: They take part in the conditional structures of the schema and that should be defined in the domains of the elements of the sequential device.

- **Expressions**: They take part in the treatment that is wanted to accomplish on the elements of the *ISD* and that also should be defined in the domains of the elements of the device.
- **ISD:** It is a device that contains the input information
- **OSD:** It is a device that contains the output information

4.3 Specification and Implementation of Sequential Schemas

For all schemas, the learners always indicate the different parameters. The input sequential device(*isd*) and output sequential device(*osd*), are both on the initial treatment positions when execution begins. Sequential schemas are based on the use of the iterative design. The specification and implementation of the basic schemas always suppose that there are a *isd* and *osd* as parameters.

4.3.1 Identity Sequential Schema

This schema is related to the change of device of a sequence. The sequential treatment moves the objects from an *isd* to an *osd* without any transformation.

Specification of the Identity Schema
Pre: {true}
Post: {s'=s}
The output sequence is identical to the input sequence.

4.3.2 Accumulative Sequential Schema

This schema is related to the application to an *ISD* of different quantifiers. It would be the sequential treatment. A quantifier *Q*, can be the counter, the maximum, the minimum, the product, or other. We use quantifiers whose neutral elements n_q and operations O_q associated with every quantifier have been previously defined.

Parameters of the Accumulative Schema
The parameters of an accumulative schema are a quantifier *Q*, an expression *E* and a predicate *P*.

Specification of the Accumulative Schema
Pre: $\{\forall i{:}1..\#s \bullet (s(i),p) \in \text{dom } P \wedge (s(i),q) \in \text{dom } E\}$
Post: $\{s' = Q\ i{:}1..\#s \mid P(s(i),p) \bullet E(s(i),q)\}$

The precondition indicates that the predicate *P* and the expression *E* should be defined for all elements of the sequence *s(i)* and the values of the parameters *p* and *q*. The auxiliary parameters *p* and *q* stand for any information that the student would like to consider in the predicate or expression, respectively.

The postcondition indicates that the *OSD*, denoted as *s'*, will be a new sequence that results from applying the corresponding quantifier to the input sequence such that if the predicate $P_i(s(i))$ is satisfied, then the corresponding operation O_{ql} is accomplished taking into account the expression $E_i(s(i))$. The bullet (•) can read as "it is the case that" and the bar(|) as "such that".

The implementation of this schema is as follows:

Implementation of the Accumulative Schema
The schema is implemented by means of a class that has a method *execute()*. It is as follows,

```
procedure execute( )
var
    r:T_r
    z:T
begin
    r:=n_q
    while isd.HasMoreObjects(
    z:=isd.Next( )
    if P(z,p)
            r:= r O_q E(z,q)
          endif
          endwhile
          osd.Write(r)
          end
```

Where n_q is the neutral element of the quantifier. O_q is the operation bound to the corresponding quantifier and T_r is the corresponding type to the neutral element of the quantifier.

Examples of application of the accumulative schema

Count the number of persons that has a given surname in a sequential file of persons.
Obtain the maximum number introduced in the standard input until a negative number is introduced.

4.3.3 Multiple Accumulative Schema

These schemas are introduced since in some problems it may be necessary to obtain the result by applying different types of quantifiers.

Parameters of the Multiple Accumulative Schema
The parameters of a multiple accumulative schema are the following:
$\{Q_1,\ldots,Q_n\}$: set of quantifiers

$\{E_1,\ldots,E_n\}$: set of expressions
$\{P_1,\ldots P_n\}$: set of predicates

Specification of the Multiple Accumulative Schema

Pre: $\{\forall\ i:1..\#s \bullet (s(i),p_1) \in \text{dom } P_1 \wedge \ldots \wedge (s(i),p_n) \in \text{dom } P_n \wedge (s(i),q_1) \in \text{dom } E_1 \wedge \ldots \wedge (s(i),q_n) \in \text{dom } E_n\}$

Post: $\{s' = Q_1\ i:1..\#s \mid P_1(s(i),p_1) \bullet E_1(s(i),q_1) \cup \ldots \cup Q_n\ i:1..\#s\ P_n(s(i),p_n) \bullet E_n(s(i),q_n)\}$

The precondition indicates that the predicates P_i and the expressions E_i should be defined for all values of $s(i)$. The postcondition indicates that the *OSD* will be a register that results from applying the corresponding quantifier to the input sequence such that if the predicate $P_i(s(i))$ is satisfied, then the corresponding operation O_{q1} is accomplished taking into account the expression $E_i(s(i))$.

Implementation of the Multiple Accumulative Schema

```
Types:
  Tr: Tr1... Tm
procedure execute( )
  var
    ri:Tr1... rm:Tm
    r:Tr
    z:T
  begin
    <r1,...,rn>:=<nq1,...,nqn>
    while isd.HasMoreObjects( )
      z:=isd.Next( )
      if P1(z,p1):
        r.r1:= r1 Oq1 E1(z,q1)
      endif
      ...
      if Pn(z,pn):
        r.rn:= rn Oqn En(z,qn)
      endif
    endwhile
    osd.Write(r)
  end
```

Where n_{qi} is the neutral element of the quantifier *i* and O_q is the operation associated to the quantifier *i*.

Examples of application of Multiple Accumulative Schema

Obtain in a Sequential File of Persons, maximum and minimum age of the Persons in this file.
Obtain the position that an object has in a sequence.

4.3.4 Searching Schema

These schemas are particular cases of the accumulative schema. As in previous schemas, searching is based on traversing a sequence with the intention of finding an object that perhaps appears in it, and that is distinguished from the others by holding a predicate *P*. The specification and implementation of these schemas use the existential quantifier. Its neutral element is *false* and the operation associated with this quantifier is the Boolean operation *Or*. The efficiency of this algorithm can be improved introducing a Boolean variable that is set to be true when the searched object is found.

4.3.5 Property checking Schema

This schema checks if the property *P* is holding for all objects of a sequence. The universal quantifier is used in this case.

Parameters of the Property checking Schema

The parameter of the schema is a predicate *P* to check.

Specification of the Property checking Schema

Pre: $\{\forall i:1..\#s \bullet (s(i),p) \in \text{dom } P\}$
Post: $\{s' = \forall i:1..\#s \bullet P(s(i),p))\}$

Where the precondition indicates that the predicate *P* should be defined for all values of $s(i)$. The postcondition indicates that the output sequential device will be the value that result from applying the universal quantifier to the input sequence.

Implementation of the Property checking Schema

The method *execute* is as follows:

```
procedure execute( )
var
  r:Boolean
  z:T
begin
  r := true
  while isd.HasMoreObjects( ) And r = true
    z := isd.Next( )
    r := r And P(z,p)
  endwhile
  osd.Write(r)
end
```

4.3.6 Filter Schema

These schemas read data of an *ISD* and write them in an *OSD*, generally altering the data in some way. The learners can combine different filters in order to obtain the expected transformation of the data.

Parameters of the Filter Schema

The parameters of this schema are a predicate *P* and an expression *E*

Specification of the Filter Schema

Pre: $\{\forall i{:}1..\#s \bullet (s(i),p) \in \mathrm{dom}\ P \wedge (s(i),q) \in \mathrm{dom}\ E\}$
Post: $\{s' = \oplus\ i{:}1..\#s \mid P(s(i),p) \bullet E(s(i),q)\}$

The precondition indicates that the predicate *P* and the expression *E* must be defined for all values of *s(i)*. The postcondition indicates that if the predicate *P(s(i),p)* is satisfied then the expression *E(s(i),q)* is written in s′. The Operator ⊕ stands for the concatenation operator, where the neutral element is the empty sequence.

Implementation of a Filter Schema

```
procedure execute()
var
    z:T
begin
  while isd.HasMoreObjects( )
      z:=isd.Next()
      if P(z,p):
         osd.Write(E(z,q))
      endif
  endwhile
end
```

In this type of schema much can be added/or removed, by substituting any type of information.

Example of application of the Filter Schema:

Obtain the new salaries of a sequence of workers such that the previous salaries are increased a quantity *q*. In this problem, the predicate *P* is *true* and the expression *E* is $E(z,q) \equiv z'=z+q$

4.3.7 Mixer Schema

This schema is applied when there is a finite set of *ISDs*. For example, we desire to obtain one sequence with the objects from a finite set of sequences, in a sequential or conditional way. Figure 3 shows as a mixer schema can treat three input sequential devices.

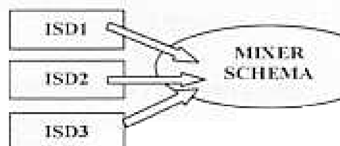


Fig 3: Diagram of the mixer schema

Simple Mixer Schema:

In this case this schema is accomplished for a finite number *N* of input sequential devices, but it can be easily extended for any finite number of input devices. It consists of linking together the sequences that represent each one of the devices according to the order that has been established.

Specification of the Simple Mixer Schema

Pre: {true}
Post: $\{s' = s_1 + s_2 + s_3\}$

Implementation of the Simple Mixer Schema

In this case the implementation has as input parameter a vector of *N* input sequential devices:

isd: *Vector[N] of ISD*

Then the implementation of the method execute would be:

```
procedure execute( )
 var
    z: T
    i: Integer
    begin
      i:=1
      while i ≤N
        while isd[i].HasMoreObjects()
          z:=isd[i].Next()
          osd.Write(z)
        endwhile
        i:=i+1
      endwhile
    end
```

Conditional Mixer Schema:

One of the mixer schemas of this type used most often is the sorting schema of several input sequential devices previously sorted, in this case assuming that there are N devices, then the specification would be:

Specification of the Conditional Mixer Schema

Pre: $\{Sorted(s_1,sort) \wedge ... \wedge Sorted(s_n,sort)\}$
Post: $\{Sorted(s',sort) \wedge \forall i{:}1..\#s' \bullet mult(s'(i),s') = mult(s'(i),s_1) + ... + mult(s'(i),s_n)\}$

The *mult* predicate indicates the number of times that the first element is repeated in the sequence.

Implementation of the Conditional Mixer Schema

```
procedure execute(
var
    z: Vector[N] of T
    i: Integer
```

```
begin
  i:=1
  while i≤N
    if isd[i].HasMoreObjects():
      z[i]:= isd[1].Next()
    else
      z[i]:= Neutral element of the operation
    endif
    i:=i+1
  endwhile
  while isd[1].HasMoreObjects() Or... Or
                  isd[N].HasMoreObjects()
    i:=SearchSorted(z)
    osd.Write(z[i])
    if isd[i].HasMoreObjects():
      z[i]:=isd[i].Next()
    else
      z[i]:= Neutral element of the operation
    endif
  endwhile
end
```

The *SearchSorted* function returns the index of the first object that satisfies the specified sort between the objects of the vector *z*.

Example of application

Calculate the mixture of several sequences of strings, where every sequence is alphabetically strings. The output sequence contains all the strings of the different sequences in the same order. In this case the neutral element is the string *"zzzzzz"*

5 Composition of sequential schemas

The previous schemas allows a broad range of composition of sequential schemas. This construction medium implies that the locus of control should be the learner. This creative process in the algorithm design is divided into two distinct phases:

- Learners identify the finite sequence of possible schemas and corresponding *ISD* and *OSD* that will solve the problem
- Learners analyze the possible compositions of the previous objects and the determination of the corresponding parameters.

This approach allows the learners to build structures of knowledge within the mind, whose aim is its application in every algorithm design problem. In the initial problem of this article:

Write an algorithm that generates a file with the sum of the squares of the positive prime numbers that end in 7 and are less than 100.

In the first step, the learners identify two schemas: a Filter Schema and a Sum Schema, an *ISD* that is one of the ISDNPRIME class and an *OSD* that is one of the OSDFILE class. Figure 4 shows the objects that can participate in the problem solving.

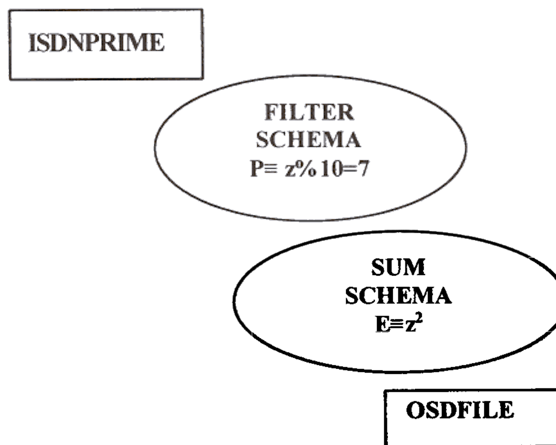


Figure 4: Identification of devices and schemas.

In the second step, the students propose a set of possible alternatives of algorithmic design in this constructive environment:

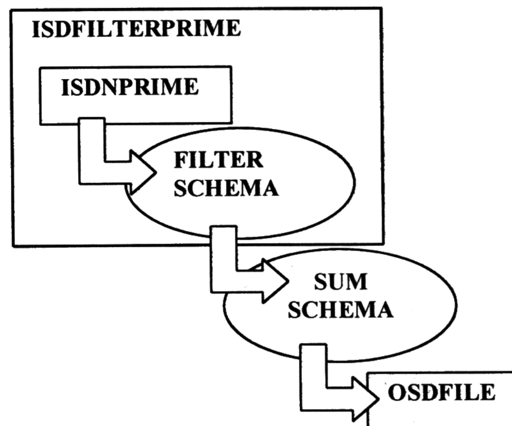
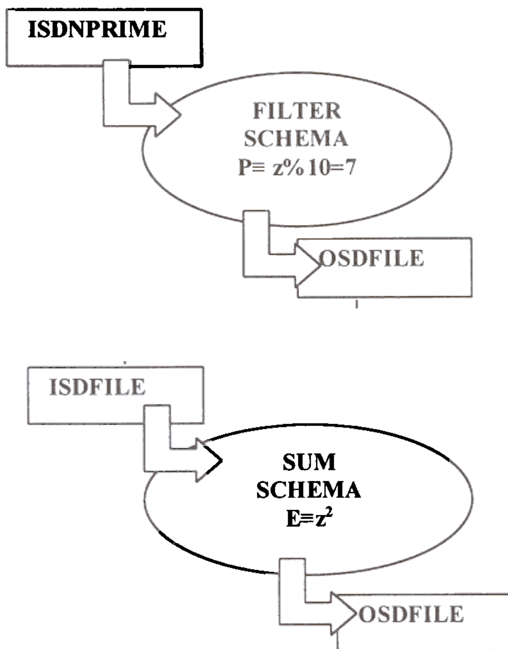


Figure 5: Algorithm Design using Combination of *ISD* and schema to create a new *ISD*.

A new *ISD* is created by means of the combination of an *ISD* and a schema. It becomes a new *ISD* that abstracts the initial *ISD* of the problem. In this case, we can say that: *ISD* + *Schema* ≡ *ISD*. This new class has as

parameters: the types of the basic objects that compose the device, a predicate and an expression. The combination of an ISDNPRIME object and a FILTER SCHEMA object constitutes a new ISDFILTERPRIME object. It is shown in Figure 5.
Finally the students apply the Sum Schema to an ISDFILTERPRIME object and they write the results in an *OSD* object that have created previously.

2. An intermediate OSDFILE object is created. Application of the Filter Schema to an ISDPRIME object writes in the previous OSDFILE. After, learners create a new ISDFILE with the file generated in the previous step and they apply the specified Sum Schema to this object. It is shown in Figure 6.



'igure 6: Algorithm Design using an intermediate file.

3 The last compositional process is an enriched Sum Schema with a Predicate and an Expression. It is shown in Figure 7.

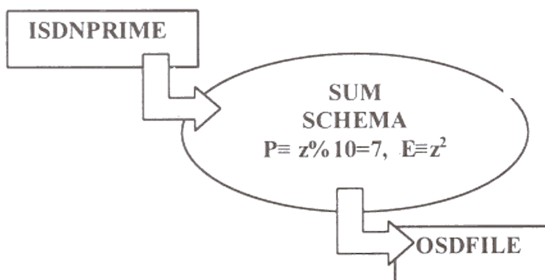


Figure 7: Algorithm Design using an enriched Sum Schema.

The well-constructed library, appropriate to this domain and the learner purposes, provides a powerful tool for conveniently solving complex problems. The essence of model building in this learning is to decide how the aspects of the problem should be explicitly described by means of sequential devices and schemas.

6 Conclusions and Future Works

The fundamental mechanisms for encoding knowledge have been the input/output devices and schemas. The main goal of the proposed metaphoric model has been the learning of the algorithm design. It allows the students a constructive environment for learning. Also, it provides a constructional medium that encourages design with a wide variety of these computational objects.

The main advantage provided by this framework is the ability of developing new algorithms using well-founded algorithm design strategies already known and verified. The different schemas cover a wide range of families of algorithms (such as traversals, searching, mix, etc.) acting over different data structures (arrays, tree, files) and other constructing elements (prime numbers).

In future works we will use this metaphoric model to other algorithmic schemas such as greedy algorithms, backtracking, branch and bound, etc.

References

Aharoni D. *Cogito, Ergo Sum! Cognitive Processes of Students Dealing with Data Structures.* In Proceedings of the 31th SIGCSE Technical Symposium on Computer Education 2000 pp. 26-30, 2000

Ben-Ari, M. *Constructivism in Computer Science Education.* In SIGCSE Bulletin 30, 1 pp. 257-261, 1998

Burgos J.M., Galve J., García J., Sutil, M. *Enseñanza de la programación basada en esquemas* In V Jornadas sobre la Enseñanza Universitaria en Informática, JENUI2000 pp. 395-402, 2000

Fleury A.E. *Programming in Java: Student-Constructed Rules.* In Proceedings of the 31th SIGCSE Technical Symposium on Computer Education 2000 pp. 197-201, 2000

Gamma E., Helm R., Johnson R. and Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Mass: Addison Wesley, 1994

Gibbs D. C. *The effect of a Constructivist Learning Environment for Field-Depedent/Independent Students on Achievement in Introductory Computer Programming.* In

Proceedings of the 31th SIGCSE Technical Symposium on Computer Education 2000 pp. 207-211, 2000

Meyer B. *Construcción de Software Orientado a Objetos* Ed Prentice-Hall 1999

Papert, S. *Situating Constructionism. Constructionism.* Harel and S. Papert, Norwood, N.J, Ablex 1991

V.K. Proulx *Programming Patterns and Design Patterns in the Introductory Computer Science Course.* In Proceedings of the 31th SIGCSE Technical Symposium on Computer Education 2000 pp. 80-84, 2000

Ramos I. and Dominguez J.J. *Docencia en gestión y estimación de proyectos software: un enfoque constructivista.* In VI Jornadas sobre la Enseñanza Universitaria de la Informática. JENUI2000 pp. 327-333, 2000

Rafael M. Gasca *obtained the Ph.D. degree in Computer Science in 1998 at the Seville University in Spain. He is professor since 1991 in the Department of Languages and Computer Systems at the Seville University. His main research topics are constraint programming and semiqualitative reasoning.*

Juan Antonio Ortega *obtained the Ph.D. degree in Computer Science in 2000 at the Seville University in Spain. He is professor since 1992 in the Department of Languages and Computer Systems at the Seville University. His research interests are in the semiqualitative simulation of dynamic systems and in the obtaining of their temporal behaviour patterns.*

Miguel Toro *obtained the Ph.D. degree in Engineering in 1987 at the Seville University in Spain. He is professor since 1985 and director since 1993 of the Department of Languages and Computer Systems at the Seville University. His research interests include software engineering, dynamic system simulation and formal methods.*