Rafael M. Gasca, Juan Antonio Ortega, Miguel Toro

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

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

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

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 fr to produce a new algorithm for every problem, kno look for known patterns that serve as a starting poir the use of an existing algorithmic design.

The teaching of problem solving by means of a design, actually considers every problem as a differe and then the student learns to develop and to different algorithms. The results of this teaching a shows serious shortcomings. In order to improve th and produce the conceptual change in the student it, we have reduced the design process to a creativ of steps and templates that the students know, wh a path to take from the initial problem statement to a solution. The student then looks for the set of sc devices that can be used to solve the problems, the individual instructions of a particular languag

The constructionist approach to learning is p the bibliography (Papert, 1991). The name const derives as a variant of the related psychologica *constructivism.* The processes of learning are co both active and creative. The learner must discove of knowledge, which are original ideas, at least rela The active and creative role of the learning proce on the idea that students should be able to use i

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

Moreover, the sequential schema encloses a algorithm described by means of abstract operation by the *ISD*. The sequential schema is specified by the common properties from the problems and i non-relevant details. A concrete algorithm can b concrete instance of a schema or combination o An abstract sequence is necessary when the stud these schemas. We say that a finite set of objects c type is organized in the form of a sequence if it is define the following operations:

- The first object of the sequence that will subsequent access to the other obje sequence.
- The next object of a given object, that will accede to an object through the object th to him, and the object *i* is reached acr objects that preceded to him.
- The operation that defines the last ob sequence.

The above operations permit the **sequential a** objects of a sequence. The treatment through t access is designated **sequential treatment.** In thi propose an iterative design technique for the treatment.

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

- To determine the basic objects those sequence. This determination permit comprehension of the problem at the high abstraction. It reduces the complexity of t
- To detect or induce a sequence structure objects to treat. For it is necessary to operations of sequences.
- To select a schema among the basic combination of schemas most adequate objects of the sequence.

This development of algorithmic design st advantages and disadvantages when taught this main advantage is that the way of integrating the an algorithm is not as basic instructions but th abstract entities, and furthermore they have been v that they will produce correct programs. Also the d of the basic operations on the sequences will have t

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*:

$$<> \in S$$
$$s \in S: e \in T: <e> + s, s + <e> \in S$$

Where $<>$ 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

number of the sequence. The sequence is defined set of objects of the same type, put in an order cursor that can take the positions from 1 to the cardin of the sequence plus one. In this way, it is eas when the sequence is finished and when the sequenc A pointer designates the sequence interest point a at an object distinguished within the same. It s reference for the operations and the task of changi object to another object. In order to accom specification of a sequence, the following attr considered:

> *s*: sequence of *T*
> *i*: Integer

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

3 Sequential devices

The students have to solve many problems by algorithms. In these problems there is input/output i and it can be abstracted through what is des *sequential device.* The following problem may be a

Write an algorithm that generates a file with the squares of the positive prime numbers that er are less than 100.

In this example, the input sequential device is that is formed by the positive prime number less by the positive prime numbers ending in 7 and le and an output sequential device that is a file w number. This number is the result of the sum of th the prime numbers of the initial sequence.

The resolution of the problem will consist then the corresponding sequential schema to an inpu device to obtain the results that are then put i sequential device. Our object-oriented approach p *OSD* and *Schemas* that can be used in a creative permits the students to solve complex problems form of constructionism realistically reflects hov learning takes place.

3.1 Input Sequential Device, ISI

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
```

$$i \in \{1..\#s+1\} \wedge m = (i \leq \#s) \wedge (i \leq \#s \Rightarrow v = s(i))$$
```
  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 the object-oriented programming paradigm, with purpose of constructing reusable and quality soft reusability is considered as the ability of producin components that can be used in different applica produce correct and robust software.

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

The implementation of an *ISD* by means of an ab 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 sequen represented in memory (for example sequential fil etc.), however they also may be "*calculated*" an one element present in memory constantly and deduced through the corresponding calculat numbers). In some cases, there is an intermedia with all the objects in the memory; the treatment ord

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.
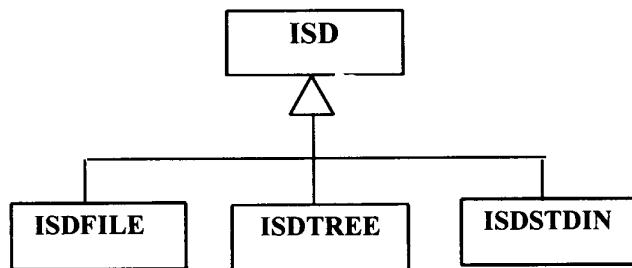

Figure 1: Hierarchy of *ISD* classes

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
```

```
   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 number or no.

Multiple *ISD* objects can be used at the same possible to traverse a same data structure in diff what will provide different *ISD* objects. The modi a data structure can cause problems, while a traversing it. In order to avoid these problems, v that these modifications are not possible.

3.2 Output Sequential Devices, (

3.2.1 Definitions and Operations

These devices permit to put the results of the ap a sequential treatment on a device previously cre an operational point of view, this has been def following operation:

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

According to this model, the specification fo 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 th

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

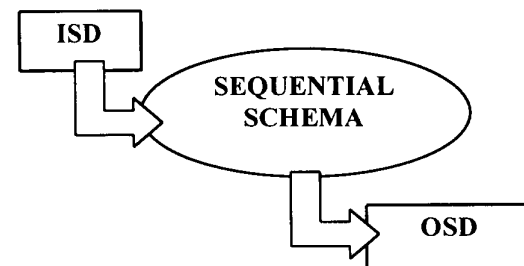schema to an *ISD* object allows us to obtain a de object. In this figure, every arrow indicates an obje

Fig 2: Diagram of the schemas application

The schemas can be basic or combinations of t always need at least one *ISD* and one *OSD* appropriately. The basic schemas that we prop learning of algorithm design are:

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

In following sections we will explain the previou schemas.

4.2 Modeling of Sequential Sche

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

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

- **Quantifiers:** They determine the types of o

- **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 • (s(i),p)$\in$dom P $\wedge$ (s(i),q) $\in$dom E}
Post: {s'= Q i:1..#s | P(s(i),p) • E(s(i),q)}

would like to consider in the predicate or ex respectively.

The postcondition indicates that the *OSD*, den will be a new sequence that results from app corresponding quantifier to the input sequence su the predicate $P_i(s(i))$ is satisfied, then the corre operation O_{q1} is accomplished taking into ac expression $E_i(s(i))$. The bullet (•) can read as "it i that" and the bar(|) as "such that".

The implementation of this schema is as follov

Implementation of the Accumulative Schema
The schema is implemented by means of a class 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 operation bound to the corresponding quantifier an corresponding type to the neutral element of the qu

Examples of application of the accumulative schen

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

4.3.3 Multiple Accumulative Schema

These schemas are introduced since in some proble be necessary to obtain the result by applying diffe of quantifiers.

{$E_1,...,E_n$}: set of expressions

{$P_1,...P_n$}: set of predicates

Specification of the Multiple Accumulative Schema

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

Post: {s'= Q_1 i:1..#s | P_1(s(i),p_1) • E_1(s(i),q_1) $\cup...\cup$
Q_n i:1..#s P_n(s(i),p_n)•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_{qi} is accomplished taking into account the expression $E_i(s(i))$.

Implementation of the Multiple Accumulative Schema

```
Types:
  T_r: T_{r1...} T_{rm}
procedure execute( )
 var
 r_i:T_{r1...} r_m:T_{rm}
 r:T_r
 z:T
begin
  <r_1,...,r_n>:= <n_{q1},...,n_{qn}>
  while isd.HasMoreObjects( )
   z:=isd.Next( )
   if P_1(z,p_1):
     r.r_1:= r_1 O_{q1} E_1(z,q_1)
   endif
   ...
   if P_n(z,p_n):
     r.r_n:= r_n O_{qn} E_n(z,q_n)
               endif
               endwhile
               osd.Write(r)
     end
```

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

Examples of application of Multiple Accumulative Schema

• Obtain in a Sequential File of Persons, maximum and

These schemas are particular cases of the ac schema. As in previous schemas, searching is traversing a sequence with the intention of findin that perhaps appears in it, and that is distinguishe others by holding a predicate *P*. The specifi implementation of these schemas use the existentia Its neutral element is *false* and the operation asso this quantifier is the Boolean operation *Or*. The ef this algorithm can be improved introducing a Boole that is set to be true when the searched object is f

4.3.5 Property checking Schema

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

Parameters of the Property checking Schema
The parameter of the schema is a predicate *P* to cl

Specification of the Property checking Schema
Pre: {$\forall$i:1..#s • (s(i),p)$\in$ dom P}
Post: {s'=$\forall$i:1..#s • P(s(i),p)}

Where the precondition indicates that the predica be defined for all values of s*(i)*. The postconditi that the output sequential device will be the valu from applying the universal quantifier to the inpu

Implementation of the Property checking Schem

The method *execute* is as follows:

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

4.3.6 Filter Schema

Parameters of the Filter Schema

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

Specification of the Filter Schema

Pre: $\{ \forall i:1..\#s \bullet (s(i),p) \in dom\ P \wedge (s(i),q) \in dom\ E \}$
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 $\oplus$ 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)* ≡ *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.

Simple Mixer Schema:

In this case this schema is accomplished for a fir N of input sequential devices, but it can be easily e any finite number of input devices. It consists together the sequences that represent each one of 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 vector of N input sequential devices:

$$isd:\ Vector[N]\ of\ ISD$$

Then the implementation of the method execute

```
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 mos sorting schema of several input sequential device sorted, in this case assuming that there are N d 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 ti first element is repeated in the sequence.

Implementation of the Conditional Mixer Schem

ISD1

MIXER

```
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.

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

In the first step, the learners identify two schem Schema and a Sum Schema, an *ISD* that is ISDNPRIME class and an *OSD* that is one of the class. Figure 4 shows the objects that can partici problem solving.
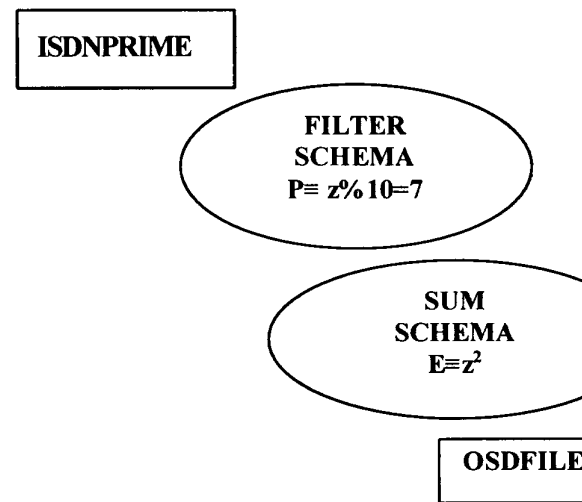

Figure 4: Identification of devices and sch

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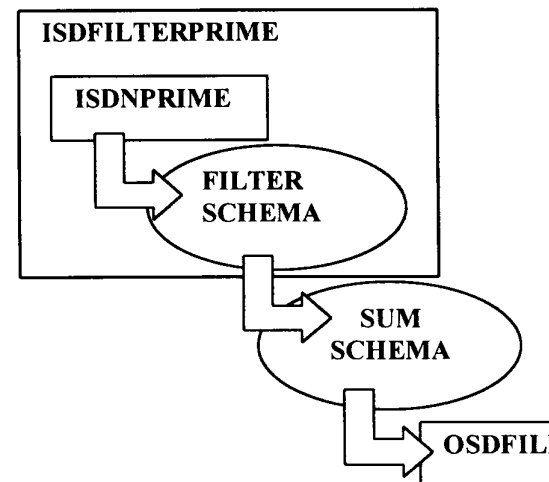

Figure 5: Algorithm Design using Combinati and schema to create a new *ISD*.

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.
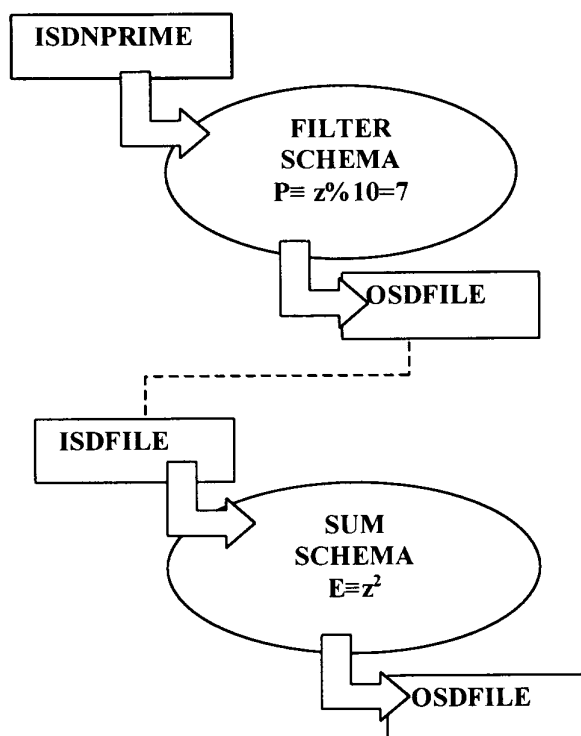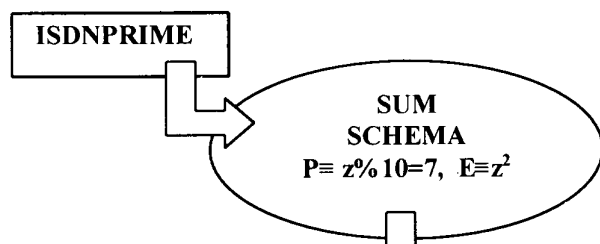

Figure 6: Algorithm Design using an intermediate file.

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

The well-constructed library, appropriate to this do the learner purposes, provides a powerful tool for con solving complex problems. The essence of model b this learning is to decide how the aspects of the proble be explicitly described by means of sequential de schemas.

6 Conclusions and Future Worl

The fundamental mechanisms for encoding knowl been the input/output devices and schemas. The ma the proposed metaphoric model has been the learn algorithm design. It allows the students a con environment for learning. Also, it provides a cons medium that encourages design with a wide variet computational objects.

The main advantage provided by this framew ability of developing new algorithms using well algorithm design strategies already known and veri different schemas cover a wide range of families of a (such as traversals, searching, mix, etc.) acting ove data structures (arrays, tree, files) and other con elements (prime numbers).

In future works we will use this metaphoric mod algorithmic schemas such as greedy algorithms, bacl branch and bound, etc.

References

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

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

Burgos J.M., Galve J., García J., Sutil, M. *Enseñ programación basada en esquemas* In V Jornada Enseñanza Universitaria en Informática, JENUI200 402, 2000

Fleury A.E. *Programming in Java: Student-Cc Rules*. In Proceedings of the 31th SIGCSE 1 Symposium on Computer Education 2000 pp. 197-2(

Gamma E., Helm R., Johnson R. and Vlisside *Patterns: Elements of Reusable Object-Oriented*

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

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

Ramos I. and Dominguez J.J. *Docencia en gesti ción de proyectos software: un enfoque construct.* Jornadas sobre la Enseñanza Universitaria de la I JENUI2000 pp. 327-333, 2000

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

Juan Antonio Ortega obtained the Ph.D. degree in Computer Science in 2000 at the Seville Universit is professor since 1992 in the Department of Languages and Computer Systems at the Seville U research interests are in the semiqualitative simulation of dynamic systems and in the obtaining of t behaviour patterns.

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