Gasca, Rafael M.; Ortega, Juan Antonio; Toro, Miguel
An Object-Oriented Approach for Learning of Algorithm Design with Sequential Devices and Schemas
Instituto Politécnico Nacional
Distrito Federal, México

Available in: http://www.redalyc.org/articulo.oa?id=61540304
An Object-Oriented Approach for Learning of Algorithm Design with Sequential Devices and Schemas

Rafael M. Gasca, Juan A. Ortega and Miguel Toro
Department 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 for the use of an existing algorithmic design.

The teaching of problem solving by means of algorithm design, actually considers every problem as a different world, and then the student learns to develop and to use different algorithms. The results of this teaching are weak and show serious shortcomings. In order to improve this approach and produce the conceptual change in the students to study it, we have reduced the design process to a creativity of steps and templates that the students know, which leads them to a path to take from the initial problem statement to a possible solution. The student then looks for the set of schemas and devices that can be used to solve the problems, to replace the individual instructions of a particular language.

The constructionist approach to learning is presented in the bibliography (Papert, 1991). The name constructivism derives as a variant of the related psychological constructivism. The processes of learning are considered both active and creative. The learner must discover knowledge, which are original ideas, at least relatively. The active and creative role of the learning process is based on the idea that students should be able to use inductive methods to learn new concepts.
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 (Gibbs, 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 concrete algorithm described by means of abstract operations by the ISD. The sequential schema is specified by a set of common properties from the problems and ignore non-relevant details. A concrete algorithm can be a concrete instance of a schema or combination of schemas. An abstract sequence is necessary when the study of these schemas. We say that a finite set of objects of a 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 allow subsequent access to the other objects of the sequence.
- The next object of a given object, that will allow to access to an object through the object that precede to him, and the object i is reached across the objects that preceded to him.
- The operation that defines the last object of the sequence.

The above operations permit the sequential accessing objects of a sequence. The treatment through this access is designated sequential treatment. In this paper we propose an iterative design technique for the sequential treatment.

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

- To determine the basic objects those form a sequence. This determination permits a deeper comprehension of the problem at the highest level of an abstraction. It reduces the complexity of the design.
- To detect or induce a sequence structure of the objects to treat. For it is necessary to produce the operations of sequences.
- To select a schema among the basic set of combination of schemas most adequate for solving the objects of the sequence.

This development of algorithmic design structures have advantages and disadvantages when taught this main advantage is that the way of integrating the design of an algorithm is not as basic instructions but through the abstract entities, and furthermore they have been verified that they will produce correct programs. Also the description of the basic operations on the sequences will have to be organized taking into account the position of the elements that are in the sequence.
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 $S$:

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

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

$$
s: \text{sequence of } T \\
i: \text{Integer}
$$

Where $i$ is the position where the interest point is, 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 involved, and it can be abstracted through what is described as a sequential device. The following problem may be an example:

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

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

The resolution of the problem will consist then in the corresponding sequential schema to an input device to obtain the results that are then put in a sequential device. Our object-oriented approach permits the students to solve complex problems in a form of constructionism realistically reflects how learning takes place.

3.1 Input Sequential Device, ISD

...
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:

\[
\begin{align*}
  s: & \text{Sequence of } T \\
  i: & \text{Integer} \\
  m: & \text{Boolean} \\
  v: & T \\
\end{align*}
\]

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:

\[
i \in \{1..\#s+1\} \land m = (i \leq \#s) \land (i \leq \#s \Rightarrow v = s(i))
\]

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**
    - \( s: \text{Sequence of } T \)
    - \( i: \text{Integer} \)
    - \( m: \text{Boolean} \)
    - \( v: T \)

### 3.1.3 Implementation of the ISD

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

The input sequential devices are implemented as an abstract class, that we have designated ISD. In an ISD, there are attributes, an object \( v \) of the sequence \( T \) in which the interest point is found and a Boolean \( m \), that is set to false when the device has no more.

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

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

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

```
\[v := v + 1\]
while NOT IsPrime(v) and v < mp
  \[v := v + 1\]
endwhile
if v \geq mp:
  m := false
endif
end
EndClass
```

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

Multiple ISD objects can be used at the same time, making it possible to traverse a same data structure in different ways that will provide different ISD objects. The modification of a data structure can cause problems, while a system that traversing it. In order to avoid these problems, we believe that these modifications are not possible.

### 3.2 Output Sequential Devices, Classes

#### 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 a following operation:

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

According to this model, the specification for a 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
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 $F\text{Write}(v: T)$ 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 components and a finite set of operations defined over these components. Each component has a single output operation that generates one piece of data or an output device. Each component can have several input devices that receive data to process. A component realizes an output device to the given output device and a component can also realize both input and output devices. The implementation of these operations is always provided by the students.

The schemas can be basic or combinations of them. In the former case, always need at least one ISD and one OSD appropriately. The basic schemas that we propose for learning of algorithm design are:

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

In following sections we will explain the previous schemas.

#### 4.2 Modeling of Sequential Schemas

The modeling intends to capture the static behavior of the schema, showing the elements that compose it and obtain an intuitive representation. These models are used 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 them. Different quantifiers, predicates and expressions are defined on the different treatments that the learners wish to be parameterized `Schema` class. That is:

- **Quantifiers**: They determine the types of operations that can be performed on the components.
- **Predicates**: They define conditions that must be satisfied by the components.
- **Expressions**: They specify how the components interact with each other.

Fig 2: Diagram of the schemas application

![Diagram of the schemas application](image-url)
• **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:** \{\( \forall i: 1..#s \cdot (s(i), p) \in \text{dom } P \land (s(i), q) \in \text{dom } E \)\}

**Post:** \{s' = Q i:1..#s | P(s(i), p) \cdot E(s(i), q)\}

would like to consider in the predicate or expression, respectively.

The postcondition indicates that the \( OSD \), defined will be a new sequence that results from applying the corresponding quantifier to the input sequence such as the predicate \( P_i(s(i)) \) is satisfied, then the corresponding operation \( O_q \) is accomplished taking into account the expression \( E(s(i)) \). 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 method \( \text{execute()} \). It is as follows,

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

Where \( n_q \) is the neutral element of the quantifier, \( O_q \) operation bound to the corresponding quantifier and \( E(s(i)) \) 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 social security number.
- Obtain the maximum number introduced in the input until a negative number is introduced.

#### 4.3.3 Multiple Accumulative Schema

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

**Parameters of the Multiple Accumulative Schema**

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

**Specification of the Multiple Accumulative Schema**

Pre: $\forall i: 1..n \cdot (s(i),p_i) \in \text{dom } P_i \land \ldots \land (s(i),p_n) \in \text{dom } P_n \land (s(i),q_i) \in \text{dom } E_i \land \ldots \land (s(i),q_n) \in \text{dom } E_n$

Post: $\{s' = Q_1 \cdot 1..n \cdot P_1(s(i),p_i) \cdot E_1(s(i),q_i) \cup \ldots \cup Q_n \cdot 1..n \cdot P_n(s(i),p_n) \cdot E_n(s(i),q_n)\}$

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 $O_{q_i}$ is accomplished taking into account the expression $E_i(s(i))$.

**Implementation of the Multiple Accumulative Schema**

Types:

$T_r : T_{r_1} \ldots T_{r_m}$

procedure execute()

var

$r_1:T_{r_1} \ldots r_m:T_{r_m}$

$r:T_r$

$k:T$

begin

$\langle r_1,\ldots,r_m,> = <n_{q_1},\ldots,n_{q_n}>$

while isd.HasMoreObjects()

$k := \text{isd.Next}( )$

if $P_i(z,p_i)$

$r_{r_i} := r_1 \cdot O_{q_i} \cdot E_i(z,q_i)$

endif

...$

if P_n(z,p_n)$

$r_{r_n} := r_n \cdot O_{q_n} \cdot E_n(z,q_n)$

endif

endwhile

osd.Write(r)

end

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

**Examples of application of Multiple Accumulative Schema**

- Obtain in a Sequential File of Persons, maximum and average of the age of the "driving" persons.

These schemas are particular cases of the accumulator schema. As in previous schemas, searching is traversing a sequence with the intention of finding that perhaps appears in it, and that is distinguished from others by holding a predicate $P_i$. The specific implementation of these schemas use the existential quantifier. Its neutral element is $false$ and the operation associated to this quantifier is the Boolean operation $Or$. The effect of this algorithm can be improved introducing a Boolean 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 each of a sequence. The universal quantifier is used in this schema.

**Parameters of the Property checking Schema**

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

**Specification of the Property checking Schema**

Pre: $\forall i: 1..n \cdot (s(i),p) \in \text{dom } P$

Post: $\{s' = \forall i: 1..n \cdot P(s(i),p)\}$

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

**Implementation of the Property checking Schema**

The method execute is as follows:

procedure execute()

var

$r: \text{Boolean}$

$k:T$

begin

$r := \text{true}$

while isd.HasMoreObjects() And $r = \text{true}$

$k := \text{isd.Next}( )$

$r := r And P(z,p)$

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

$\text{Pre:} \{ \forall i:1..\#s \cdot (s(i),p) \in \text{dom } P \land (s(i),q) \in \text{dom } E \}$
$\text{Post:} \{ s' = \oplus i:1..\#s | P(s(i),p) \land E(s(i),q) \}$

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 $\text{true}$ 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 finite $N$ of input sequential devices, but it can be easily extended to any finite number of input devices. It consists of taking in turn all the sequences that make up $s$, together the sequences that represent each one of them, and writing them in sequence according to the order that has been established.

Specification of the Simple Mixer Schema

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

Implementation of the Simple Mixer Schema

In this case the implementation has as input parameters a vector of $N$ input sequential devices:

$$\text{isd: } \text{Vector}[N] \text{ of ISD}$$

Then the implementation of the method executes:

```
procedure execute()
var
  z: T
  i: Integer
begin
  i := 1
  while $i \leq N$
    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 frequently is a sorting schema of several input sequential devices, the input sequences being arranged in order, in this case assuming that there are $N$ devices. The specification would be:

Specification of the Conditional Mixer Schema

$\text{Pre: } \{ \text{Sorted}(s_1, \text{sort}) \land \ldots \land \text{Sorted}(s_N, \text{sort}) \}$
$\text{Post: } \{ \text{Sorted}(s', \text{sort}) \land \forall i:1..\#s' \cdot \text{mult}(s'(i),s(i)) \}$

The $\text{mult}$ predicate indicates the number of times the first element is repeated in the sequence.

Implementation of the Conditional Mixer Schema
begin
i:=1
while i≤N
  if isd[i].HasMoreObjects():
    z[i]:= isd[i].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)
isd.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 "zzzzzzz"

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 schemes: ISDNPRIME Schema and a Sum Schema, an ISD that is an ISDNPRIME class and an OSD that is one of the SUM class. Figure 4 shows the objects that can participate in problem solving.

![Figure 4: Identification of devices and schemes](image)

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

![Figure 5: Algorithm Design using Combinations of devices and schema to create a new ISD](image)
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 ISDPRIIME 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.

![Diagram](image)

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.

![Diagram](image)

6 Conclusions and Future Work

The well-constructed library, appropriate to this domain, the learner purposes, provides a powerful tool for constructing complex problems. The essence of model based learning is to decide how the aspects of the problem can be explicitly described by means of sequential design schemas.

References


Gamma E., Helm R., Johnson R. and Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley 1995


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


---

**Rafael M. Gasca** obtained the Ph.D. degree in Computer Science in 1998 at the Seville University where he is professor since 1991 in the Department of Languages and Computer Systems at the Seville University. His 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 where 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 the behavior patterns.

---

**Miguel Toro** obtained the Ph.D. degree in Engineering in 1987 at the Seville University in Spain. He has been 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.