An Approach to Algorithm Design by Patterns

Javier Galve-Francés[1] Julio García-Martín[2]
Jose.M. Burgos-Ortíz Miguel Sutil-Martín
Universidad Politécnica de Madrid[3]

Abstract

This paper proposes two behavioral patterns DIVIDE-&-CONQUER and BACKTRACKING to facilitate the development of algorithm based upon algorithm design techniques. The common informal seudocode that specifies the solving strategy for a general algorithm design technique is enhanced to provide a helpful guide to develop particular algorithms by following the ***divide and conquer*** *and the* ***backtracking*** *design techniques.*

1. Introduction

Algorithm design is a creative activity that is not subject to recipes. The existence of many important problems for which no efficient algorithms are known is an evidence of this fact. In practice, the space of choices to develop algorithms is enormous. For this reason, and in opposition to the development from scratch, the task of solving algorithmic problems can be broached applying more abstract design techniques. These techniques offer common solving strategies for different problems. A design technique is often expressed in seudocode as a template that can be particularized for concrete problems. Let us name this template *algorithm schemas*.

The basic idea for algorithm schemas consist on identifying structural similarities among algorithms that solve different problems. These similarities can be characterized in terms of some abstract components and their relationships. In this situation, the algorithm schema is understood to be a behavioral template that specifies the common procedural abstraction acting over the abstract components. Moreover, the schema encloses an abstract algorithm described by abstract operations provided by the components. As the schema is specified by abstracting the common properties of the problems and ignoring its non-relevant details, a concrete algorithm can be seen as a concrete instance of the schema.

Traditionally, algorithm schemas have been seen as informal specifications that guide the designs. As a consequence, this traditional approach leads to obtain implementations where the "flavor" of the schema is lost or unrecognizable. In opposition to this informal approach, a more formal approach allows the introduction of a higher-degree of clarity to the descriptions, making them more precise and reusable. These more rigorous specifications are a very valuable reference for the programmer, as well as provides a support to analyze the correctness and efficiency of algorithms.

In this proposal, we try to enhance the traditional role played by the algorithm schemas in order to provide a more systematic approach. Design patterns are the description

[1] This work has been partially supported by the Spanish PRONTIC project TIC95-0967-C02-01.
[2] This work has been partially supported by the Spanish PRONTIC project TIC96-1012-C02-02.
[3] LSIIS Department, Facultad de informática, Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain. Email: {jgalve, juliog, jmburgos}@fi.upm.es

language used for this goal. This way, we model algorithm schemas as well-established patterns, whose components encapsulate the abstract components underlying the schema.

The current paper presents two patterns to model two well-known examples of algorithm design schemas; we call them DIVIDE-&-CONQUER and BACKTRACKING. Both patterns are organized in a two-level structure, which emphasizes the separation between the abstract schema and concrete algorithms. At the highest-level, each pattern is defined by the tuple of components: AbstractProblem / Schema / AbstractSolution. A concrete algorithm based upon one of the schemas is obtained by deriving subclasses from AbstractProblem and AbstractSolution.

2. The DIVIDE-&-CONQUER Pattern[4]

2.1. Intent

The intent of the DIVIDE-&-CONQUER pattern is to provide algorithm-based solutions for a characterized set of problems by following a divide-and-conquer strategy. This strategy is based on breaking one large problem into several smaller problems easier to be solved. Organized as a two-level structure (abstract/concrete), the pattern separates the problem domain from the solution domain. Moreover, it defines the complete skeleton of the divide-and-conquer strategy, deferring to subclasses the steps which determine a concrete algorithm.

2.2. Motivation

One of the most powerful techniques for solving problems is to break them down into smaller, more easily solved pieces. Smaller problems are less overwhelming, and they permit us to focus on details that are lost when we are studying the entire problem. For example, whenever we can break the problem into smaller instances of the same type of problem, a recursive algorithm starts to become apparent.

Consider the classic sorting algorithm *Mergesort*. Let us suppose we take a pile **P** with n elements to sort and split (operation *Divide*) them into piles **A** and **B**, each with half the elements. For example:

$$\mathbf{P} = \{ 12, 19, 1, 4, 8, 2, 20, 10 \}$$

$$\mathbf{A} = \{ 12, 19, 1, 4 \} \qquad \mathbf{A}' = \{ 1, 4, 12, 19 \}$$

$$\mathbf{B} = \{ 8, 2, 20, 10 \} \qquad \mathbf{B}' = \{ 2, 8, 10, 20 \}$$

$$\mathbf{P}' = \{ 1, 2, 4, 8, 10, 12, 19, 20 \}$$

After sorting both piles, it is easy to combine (operation *Combine*) the two sorted piles (at the example, **A'** and **B'**). To merge **A'** and **B'**{PRIVATE}{PRIVATE}, note that the smallest item must sit at the left of one of the two piles. Once identified, the smallest element (operation *IsSmall*) can be removed (operation *DirectSolution*), and the second smallest item

[4] We present the DIVIDE-&-CONQUER and BACKTRACKING patterns separately in different sections.

will again be a top one of the two piles. Repeating this operation merges the two sorted piles (the pile solution P').

Mergesort is a classic divide-and-conquer algorithm. Whenever we can break one large problem into two smaller problems, we are ahead of one of these cases because the smaller problems are easier. The point is taking advantage of the two partial solutions to put together a solution of the full problem. Obviously, not all problems can be so neatly decomposed.

2.3. Applicability

Use the DIVIDE-&-CONQUER pattern in the following situations:

- To develop algorithm-based solutions for problems can be decomposed following the divide-and-conquer strategy. The implementation of the invariant parts of the strategy are provided for free, leaving up to subclasses the implementation of the parts that can vary.

- To carry out proofs about correctness and efficiency of algorithm based solutions. The pattern can behave as a framework of algorithmic analysis. The clear separation between the common abstract-level and concrete problem/solution domains can help to get better-structured and data-independent algorithms, more suitable to be reused or analyzed. Given an algorithm as a DIVIDE-&-CONQUER pattern, its implementation is distributed along the components of the pattern. Then, it is possible to establish local analysis, firstly, and afterwards to reassemble these local results in a unique general result.

2.4. Structure

The structure of the DIVIDE-&-CONQUER pattern is shown on figure 1.

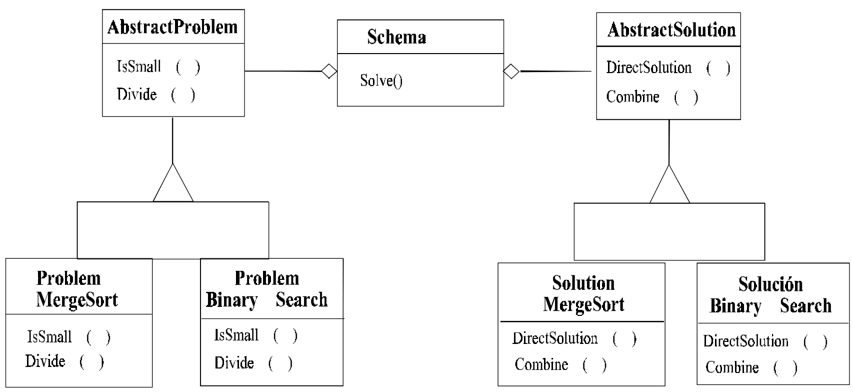

Figure 1. *DIVIDE-&-CONQUER (structure). Organized as a two-level structure. On the top, the pattern describes the abstract components for the divide-and-conquer schema and their relationships. On the bottom, the subclasses determining each concrete algorithm.*

2.5. Participants

- **AbstractProblem.** It abstracts the problem domain in the divide-and-conquer schema. It encapsulates two abstract operations:

    - *IsSmall,* which determines whether the problem is small enough or not to be solved directly (i.e., it is a basic-case), and

    - *Divide,* which splits the problem into smaller subproblems.

- **AbstractSolution.** It abstracts the solution domain in the divide-and-conquer schema. It encapsulates two abstract operations:

    - *DirectSolution*, which returns an answer for a basic-case problem, and

    - *Combine,* which combines partial solutions to get a solution of the full problem.

- **Schema.** It provides the strategy - the abstract algorithm- underlying the divide-and-conquer schema. The operation *Solve* is completely defined by combining operations from **`ProblemDomain`** and **`SolutionDomain`**. The semantics of `Solve` (i.e., its implementation) determines the D&C strategy expressing how the problem is solved.

- **ConcreteProblem.** It provides concrete implementations for the primitive operations defined in AbstractProblem. These implementations determine the meaning of the operations for a concrete problem domain.

- **ConcreteSolution.** It provides concrete implementations for the primitive operations defined in AbstractSolution. These implementations determine the meaning of the operations for a concrete solution domain.

2.6. Collaborations

- **`ConcreteProblem`** and **`ConcreteSolution`** rely on **`Schema`** to implement concrete algorithms by following a divide and conquer schema (see Figure 2).

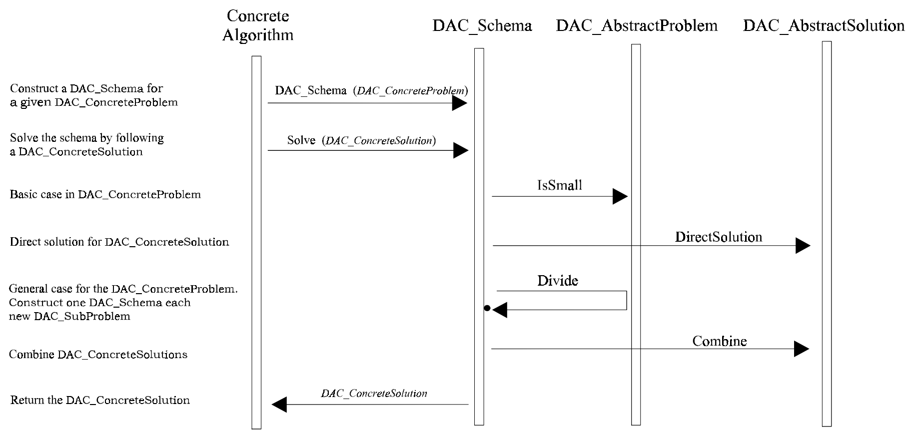

Figure 2. *DIVIDE-&-CONQUER (collaborations). One ConcreteProblem and one ConcreteSolution construct a concrete algorithm as one instance of the pattern. Then, the operation **Solve** solves the algorithm by following the divide-and-conquer strategy.*

2.7. Consequences

- *The DIVIDE-&-CONQUER pattern defines an ABSTRACT FACTORY of algorithms.* It allows the definition of new algorithms based upon the divide-and-conquer design schema. The pattern's structure strongly establishes which components define the abstract-level and how are the relationships among them. Besides, the relationships with concrete components of the pattern (subclasses) are established.

- *The pattern can be used to build algorithm libraries*, in which the common behavior supplied by the algorithm design schemas can be reused.

- *The patterns stress the so called "Hollywood principle"* [Swe85], due to the fact that the operation *Solve* fully provides the engine for the schema.

- *The pattern infers an error-free strategy to develop algorithms*. The operation *Solve* in the `Schema` calls only those operations provided by the `AbstractProblem` and `AbstractSolution`. However, to implement these operations, other operations defined by `ConcreteProblem` and `ConcreteSolution` -or by other components- can be used.

- *The patterns provide a systematic approach that simplifies the development of algorithms.* In both patterns, the abstract operations defined in the `AbstractProblem` and `AbstractSolution` must be overridden by the concrete operations in the `ConcreteProblem` and `ConcreteSolution` components. To reuse the schemas effectively, subclass developers must understand which operations are defined for overriding.

2.8. Implementation

Let us note the following implementation issues:

1. *Implementation-driven guidelines*. All operations involve in the DIVIDE-&-CONQUER pattern are defined by the `AbstractProblem` and the `AbstractSolution`.

2. *Primitive operations.* Operations defined in the `AbstractProblem` and the `AbstractSolution` are primitive. Then, they must be overridden. For example, they could be declared as pure virtual (in C++ conventions) or as part of an interface (Java conventions). The operation *Solve* must be never overridden.

3. *Naming conventions*. It is possible to identify by its name what is the intended meaning of each operation defined in AbstractProblem and AbstractSolution classes.

2.9. Sample Code

The DIVIDE-&-CONQUER pattern can be used to model the classical MergeSort algorithm [Horowitz 78]. The classes `MergeSort_Problem` and `MergeSort_Solution` are derived from the interfaces `AbstractProblem` and `AbstractSolution`, respectively.

Both `MergeSort_Problem` and `MergeSort_Solution` are implemented as collections of numbers (not necessarily arrays). Their operations are described below:

1. `AbstractProblem` and `AbstractSolution` are implemented as interfaces of abstract operations:

```
interface AbstractProblem {
   public boolean SmallEnough ();
   public List Divide ();
}

interface AbstractSolution {
   public AbstractSolution Combine (List solutions);
   public AbstractSolution DirectSolution (AbstractProblem p);
}
```

2. The operation *Solve* provides the abstract algorithm to the pattern. It can be specified abstractly, not giving concrete details about how it is implemented. Moreover, different implementations of Solve are possible. Below, it is described a concrete implementation in Java.

```
class Schema {
   private AbstractProblem  the_problem;
   private AbstractSolution the_solution;
   Schema (AbstractProblem problem, AbstractSolution solution) {
        the_problem = problem;
        the_solution = solution;
   }
   public AbstractSolution Solve ()  {
        if (the_problem.SmallEnough ())  {
             return the_solution.DirectSolution(the_problem);
        }
        else  {
           List subProblems = the_problem.Divide ();
           List subSolutions = new List();

           while (! subProblems.IsEmpty()) {
                Schema newDAC = new Schema (subProblems.Head(), the_solution);
                subSolutions.Add (newDAC.Solve());
                subProblems = subProblems.Rest();
           }
           return the_problem.Combine (subSolutions);
        }
  }  /* Solve */

} /* Schema */
```

3. In the class `MergeSort_Problem`, the operation *IsSmall* determines if a collection of numbers is small enough. To do this, it is determined if its size is one. On the contrary, the operation *Divide* splits the collection into two sub-collections of lower size.

```
class MergeSort_Problem implements AbstractProblem {
   ...
   MergeSort_Problem (...) {
        ...
   }
   public boolean SmallEnough () {
        return data.length == 1;
   }
   public List Divide ()  {
```

```
            ...
        }
    }
```

1. In the class `MergeSort_Solution`, the operation *DirectSolution* returns a collection as the result for the basic-case (i.e., if the operation IsSmall succeeded). In this case, the direct solution is the collection itself. On the other hand, given two sub-collections (sub-solutions), the operation *Combine* returns the sorted sum collection (*Merge* operation [Horowitz 78]).

```
class MergeSort_Solution implements AbstractSolution {
    ...
    MergeSort_Solution (...) {
        ...
    }
    public MergeSort_Solution DirectSolution (MergeSort_Problem p) {
        ...
    }
    public MergeSort_Solution Combine (List solutions)  {
        ...
    }
}
```

5. In the class `MergeSort_Solution`, the operation *DirectSolution* returns a collection as the result for the basic-case (i.e., if the operation *IsSmall* succeeded). In this case, the direct solution is the collection itself. On the other hand, given two sub-collections (sub-solutions), the operation *Combine* returns the sorted sum collection (*Merge* operation [Horowitz 78]).

2.10. Known Uses

The DIVIDE-&-CONQUER pattern can be used to solve a wide range of problems. In general, it is possible to employ the pattern with every problem that fits in the divide-and-conquer schema. Well known examples of uses of the pattern can be found in [Horowitz 78] and [Skiena 97]. Among other, we can recall the following:

- Sorting : sorting elements in an array (mergesort, quicksort).
- Searching : searching for an element in an ordered array (binary search)
- Solution to the travelling salesman problem (problems with graphs)
- Matrix multiplication (Strassen's matrix multiplication)
- Fourier transforms.
- Fast exponentiation.

As a fact, to comment that it is even possible to apply similar ideas of sorting and searching to other non-array data structures. In this sense, the pattern does not constrain the solutions to a particular sort of data structures.

3. The BACKTRACKING Pattern

3.1. Intent

The intent of the BACKTRACKING pattern is to provide algorithm-based solutions for a characterized set of problems by following a backtracking strategy. This strategy is based on looking for a systematic way to go through all the possible configurations of a space. These configurations may be all possible arrangements of objects (permutations) or all possible ways of building a collection of them (subsets). Organized as a two-level structure (abstract/concrete), the pattern separates the problem domain from the solution domain. Moreover, it defines the complete skeleton of the backtracking strategy, deferring to subclasses the steps which determine a concrete algorithm.

3.2. Motivation

Backtracking is a systematic way to go through all the possible configurations of a space. These configurations may be all possible arrangements of objects (permutations) or all possible ways of building a collection of them (subsets). Other applications may demand enumerating all spanning trees of a graph, all paths between two vertices, or all possible ways to partition the vertices into color classes. What these problems have in common is that we must generate each one of the possible configurations exactly once. Avoiding both repetitions and missing configurations means that we must define a systematic generation order among the possible configurations.

Consider the problem of calculate the set of permutations for a given set **A = {1, ..., n}**. To design a suitable state space for representing permutations, we start by counting them. There are **n** distinct choices (partial solution sets) for the value of the first element of a permutation of the set **A**.

$\mathbf{S_1 = \{ 1 \}}$ $\mathbf{S_2 = \{ 2 \}}$ **...** $\mathbf{S_n = \{ n \}}$

Once we have fixed this value of **A1**, there are **n-1** candidates remaining for the second position, since we can have any value except **A1** (repetitions are forbidden).

CANDIDATES = { 2, 3, ... , n }

$\mathbf{S_{12} = \{ 1, 2 \}}$ $\mathbf{S_{13} = \{ 1, 3 \}}$ **...** $\mathbf{S_{1n} = \{ 1, n \}}$

Repeating this argument yields a total of **n!** distinct permutations.

The search procedure works by growing solutions one element at a time. At each step in the search, we will have constructed a partial solution with elements fixed for the first **k** elements of the set A. From this partial solution, we will construct the set of possible candidates for the (**k+1**)st position. We will then try to extend the partial solution by adding the next element from **CANDIDATES**. So long as the extension yields a longer partial solution, we continue to try to extend it. However, at some point, the candidates set might be empty, meaning that there is no legal way to extend the current partial solution. If so, we must backtrack, and replace the last item in the solution value, with the next candidate. It is this backtracking step that gives the strategy its name.

3.3. Applicability

Use the BACKTRACKING pattern in the following situations:

- To develop algorithm-based solutions for problems can be decomposed following the backtracking strategy. The implementation of the invariant parts of the strategy are provided for free, leaving up to subclasses the implementation of the parts that can vary.

- To carry out proofs about correctness and efficiency of algorithm based solutions. The pattern can behave as a framework of algorithmic analysis. The clear separation between the common abstract-level and concrete problem/solution domains can help to get better-structured and data-independent algorithms, more suitable to be reused or analyzed. Given an algorithm as a BACKTRACKING pattern, its implementation is distributed along the components of the pattern. Then, it is possible to establish local analysis, firstly, and afterwards to reassemble these local results in a unique general result.

3.4. Structure

The structure of the BACKTRACKING pattern is shown on figure 3.

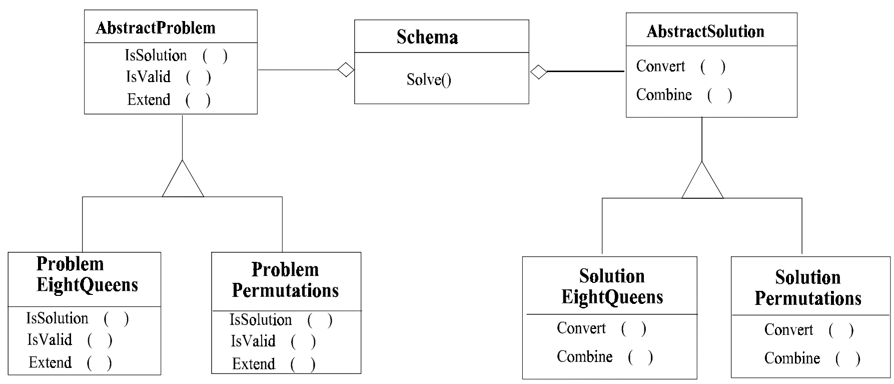

Figure 3. *BACKTRACKING pattern (structure). Organized as a two-level structure. On the top, the pattern describes the abstract components for the backtracking schema and their relationships. On the bottom, the subclasses determining each concrete algorithm.*

3.5. Participants

- **`AbstractProblem.`** It abstracts the problem domain in the backtracking schema. It encapsulates two abstract operations:
    - *IsSolution,* which determines whether the problem is just a solution or not,
    - *IsValid*, which determines if the problem should be still considered, and
    - Exdend, which constructs the next subset of problems to be considered.
- **`AbstractSolution.`** It abstracts the solution domain in the backtracking schema. It encapsulates two abstract operations:
    - *Convert,* which converts a basic-case problem in a solution, and
    - *Combine,* which combines partial solutions to get a solution of the full problem.
- **Schema.** It provides the strategy - the abstract algorithm- underlying the backtracking schema. The operation Solve is completely defined by combining operations from **`ProblemDomain`** and **`SolutionDomain`**. The semantics of `Solve` (i.e., its implementation) determines the backtracking strategy expressing how the problem is concretely solved.
- **ConcreteProblem.** It provides concrete implementations for the primitive operations defined in AbstractProblem. These implementations determine the meaning of the operations for a concrete problem domain.
- **ConcreteSolution.** It provides concrete implementations for the primitive operations defined in AbstractSolution. These implementations determine the meaning of the operations for a concrete solution domain.

3.6. Collaborations

- **`ConcreteProblem`** and **`ConcreteSolution`** rely on **Schema** to implement concrete algorithms by following the backtracking schema (see Figure 4).

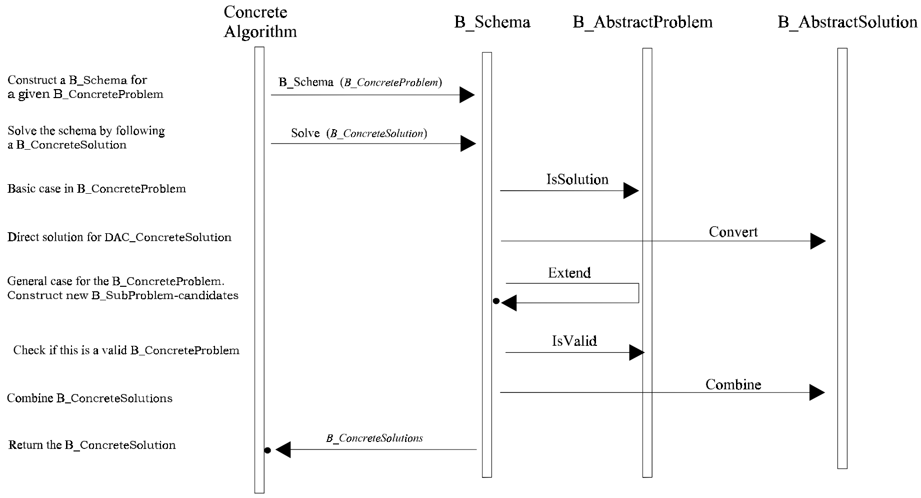

Figure 4. *BACKTRACKING pattern (collaborations). One ConcreteProblem and one ConcreteSolution construct a concrete algorithm as one instance of the pattern. Then, the operation **Solve** solves the algorithm by following the backtracking strategy.*

3.7. Consequences

- *The BACKTRACKING pattern defines an ABSTRACT FACTORY of algorithms.* It allows the definition of new algorithms based upon the backtracking design strategy. The pattern's structure strongly establishes which components define the abstract-level and how are the relationships among them. Besides, the relationships with concrete components of the pattern (subclasses) are established.

- *The pattern can be used to build algorithm libraries*, in which the common behavior supplied by the algorithm design schemas can be reused.

- *The patterns stress the so called "Hollywood principle"* [Swe85], due to the fact that the operation *Solve* fully provides the engine for the schema.

- *The pattern infers an error-free strategy to develop algorithms*. The operation *Solve* in the `Schema` calls only those operations provided by the `AbstractProblem` and `AbstractSolution`. However, to implement these operations, other operations defined by `ConcreteProblem` and `ConcreteSolution` -or by other components- can be used.

- *The patterns provide a systematic approach that simplifies the development of algorithms.* In both patterns, the abstract operations defined in the `AbstractProblem` and `AbstractSolution` must be overridden by the concrete operations in the `ConcreteProblem` and `ConcreteSolution` components. To reuse the schemas effectively, subclass developers must understand which operations are defined for overriding.

3.8. Implementation

Let us note the following implementation issues:

4. *Hard implementation guidelines*. All operations need by the BACKTRACKING pattern are provided by the `AbstractProblem` and the `AbstractSolution`.

5. *Primitive operations.* Operations defined in the `AbstractProblem` and the `AbstractSolution` are primitive. Then, they must be overridden. For example, they could be declared as pure virtual (in C++ conventions) or as part of an interface (Java conventions). The operation *Solve* must be never overridden.

6. *Naming conventions*. It is possible to identify by its name what is the intended meaning of each operation defined in AbstractProblem and AbstractSolution classes.

3.9. Sample Code

The BACKTRACKING pattern can be used to model the classical 6-Queens problem [Horowitz 78]. The classes **Queens_Problem** and **Queens_Solution** are derived from the interfaces **AbstractProblem** and **AbstractSolution**, respectively. Their operations are described below:

1. **AbstractProblem** and **AbstractSolution** are implemented as interfaces of abstract operations:

```
interface AbstractProblem {
   public boolean IsSolution ();
   public List Extend ();
   public boolean IsValid ();
}
```

```
interface AbstractSolution {
   public List Concat (List solutions);
   public AbstractSolution Convert ();
}
```

2. The operation *Solve* provides the abstract algorithm to the pattern. It can be specified abstractly, not giving concrete details about how it is implemented. Moreover, different implementations of Solve are possible. Below, it is described a concrete implementation in Java.

```
class Schema {
   private AbstractProblem  the_problem;
   private AbstractSolution the_solution;
   Schema (AbstractProblem problem, AbstractSolution solution) {
        the_problem = problem;
        the_solution = solution;
   }
   public List Solve () {
        if (the_problem.IsSolution())        {
               return the_solution.Convert(the_problem);
        }
        else  {
           List extendedProblems  = the_problem.Extend ();
           while (!extendedProblems.IsEmpty()) {
               Problem problem = extendedProblems.head();
                if (problem.IsValid())  {
                     Backtracking newBack = new Backtracking (problem);
                     the_solution = the_solution.concat (newBack.Solve());
                } /* if */
                extendedProblems = extendedProblems.tail();
            } /* while */
            return the_solution;
         } /* else */

   }  /* Solve method */
} /* Schema */
```

3.10. Known Uses

Backtracking is a systematic way to go through all the possible configurations of a space. These configurations may be all possible arrangements of objects (permutations) or all possible ways of building a collection of them (subsets). Other applications may demand enumerating all spanning trees of a graph, all paths between two vertices, or all possible ways to partition the vertices into color classes.

4. Related Patterns

Although the *Archetype* pattern [Ainsworth 93, Chandy 94] keeps quite similarities with the present proposal, it is highly focused on parallel processing and does not provide a general schema for designing algorithms.

The central axis for our two abstract schemas is a variant of the *Template Method* pattern [Gamma 95], where the operation *Solve* plays the role of abstract operation (in both patterns). As said before, the pattern can be seen as an *Abstract Factory* of algorithms [Gamma 95], where its abstract components are prefixed in advance, until they are refined later with concrete Problem/Solution components.

On the other hand, the operation Solve can be defined in such a manner that the AbstractSolution component could act as an *Adapter*, in order to obtain a different presentation for the solutions provided by an algorithm.

5. Conclusions & Future Work

The main advantage provided by DIVIDE-&-CONQUER and BACKTRACKING patterns is the ability of developing new algorithms using well-founded design algorithm strategies already known and tested. As a fact, the literature on algorithms is plenty of well-known algorithms that are specified this way.

Recently, we have defined new patterns for other strategies (such as linear recursion, dynamic programming, probabilistic algorithms, etc.). Moreover, we have applied the two presented patterns to model a very representative set of examples. These examples cover a wide range of families of algorithms (such as sorting, searching, traversals, etc ..) acting over different data structures (arrays, matrixes, graphs, etc.).

Acknowledgements

Thanks to Bobby Woolf for encouraging us to improve this article and for his patience while waiting for us to complete it. Thanks also to the Program Committee reviewer for his valuable suggestions on the last draft of this article.

Bibliography

[Ainsworth 93] P.Ainsworth: *Multimedia Interactive Environment Using Program Archetypes: Divide-and-Conquer*, Technical Report CS-TR-93-13, Caltech University, 1993.

[Chandy 94] P.Dyson : *Patterns for Abstract Design*, Technical Report, Univ.of Essex, England, 1997.

[Dyson 97] P.Dyson : *Patterns for Abstract Design*, Technical Report, Univ.of Essex, England, 1997.

[Galve et al. 93] J.Galve, J.C.González, A.Sánchez, J.A.Velázquez: *ALGORITMICA: Diseño y Análisis de Algoritmos Funcionales e Imperativos*, RA-MA

Ed., 1993.

[Gam 95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns. Elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[Horowitz 78] E.Horowitz, S. Sahni: *Fundamentals of Computer Algorithms*, Computer Science Press, 1978.

[Skiena 97] Steven S. Skiena: *The Algorithm Design Manual*, Springer-Verlag, New York, 1997.

[Swe 85] R. Sweet. *The Mesa Programming Environment*, SIGPLAN Notices, 20 (7), 216-229, July, 1985.