Designing Generic Algorithms for Operations Research

Bruno Bachelet, Antoine Mahul, Loïc Yon

Designing Generic Algorithms for Operations Research

Bruno Bachelet[1], Antoine Mahul[2] and Loïc Yon[3]
LIMOS, UMR 6158-CNRS,
Université Blaise-Pascal, BP 10125, 63173 Aubière, France.

Research Report LIMOS/RR03-20

[1]bruno.bachelet@isima.fr - http://frog.isima.fr/bruno
[2]antoine.mahul@isima.fr - http://frog.isima.fr/antoine
[3]loic.yon@isima.fr - http://frog.isima.fr/loic

Abstract

Design solutions have been proposed to implement generic data structures, however there is no technique that advanced for algorithms. This article discusses various problems encountered when designing reusable, extensible, algorithms for operations research. It explains how to use object-oriented concepts and the notion of genericity to design algorithms that are independent of the data structures and the algorithms they handle, but that can still interact deeply together. An object-oriented design is often considered to be less efficient than a classical one, and operations research is one of these scientific fields where efficiency really matters. Hence, the main goal of this article is to explain how to design algorithms that are both generic and efficient.

Keywords: object-oriented design, operations research, algorithm implementation, genericity, reusability.

Résumé

Des solutions de conception ont été proposées pour implémenter des structures de données génériques. Cependant il n'existe pas de technique aussi évoluée pour les algorithmes. Cet article discute de différents problèmes rencontrés dans la conception d'algorithmes réutilisables, extensibles, pour la recherche opérationnelle. Il explique comment utiliser les concepts orientés objet et la notion de généricité pour concevoir des algorithmes qui sont indépendants des structures de données et des algorithmes qu'ils manipulent, mais pouvant néanmoins interagir fortement entre eux. Une conception orientée objet est souvent considérée comme moins efficace qu'une conception dite classique, et la recherche opérationnelle est l'un de ces domaines scientifiques où l'efficacité est vraiment importante. Ainsi, le principal but de cet article est d'expliquer comment concevoir des algorithmes qui sont à la fois génériques et efficaces.

Mots clés : conception orientée objet, recherche opérationnelle, implémentation d'algorithme, généricité, réutilisabilité.

Abstract

Design solutions have been proposed to implement generic data structures, however there is no technique that advanced for algorithms. This article discusses various problems encountered when designing reusable, extensible, algorithms for operations research. It explains how to use object-oriented concepts and the notion of genericity to design algorithms that are independent of the data structures and the algorithms they handle, but that can still interact deeply together. An object-oriented design is often considered to be less efficient than a classical one, and operations research is one of these scientific fields where efficiency really matters. Hence, the main goal of this article is to explain how to design algorithms that are both generic and efficient.

Keywords: object-oriented design, operations research, algorithm implementation, genericity, reusability.

Introduction

The authors of this article work on various projects in the field of operations research: they use optimization techniques for graph problems in hypermedia synchronization (cf. [3]) or in bus routing (cf. [12]), and neural approximation in communication networks (cf. [8]). In all these studies, developing generic but efficient algorithms has been a challenge.

By *generic*, we mean software components that are extensible (their behavior can be adapted to fit various goals, which is also called *reusability*), and independent (although they can interact deeply together, the components must be as independent of each other as possible). This independence means for algorithms to abstract both the data structures and the algorithms they handle.

When an object-oriented design is said to be inefficient, it is usually due to an overuse of inheritance. But, what really leads to an inefficient implementation with inheritance is the notion of virtuality, and more precisely, the dynamic polymorphism that is induced. Virtual methods require more time to be executed. The cause is not the polymorphism mechanism itself but the fact that it can prevent the *inlining* of the method (i.e. the call to the method is replaced by the body of the method itself), cf. [11] and [7]. Although some design solutions exist to attempt avoiding dynamic polymorphism (e.g. the *delegation*, cf. [6]), the genericity seems to be preferred in languages that can afford it. The massive use of genericity leads to *generic* programming (cf. [10]). In this article, we propose to use both object-oriented and generic programming to avoid dynamic polymorphism when calling critical methods.

Section 1 briefly recalls the solutions to design generic data structures and how to conceive algorithms as independent as possible of them. Section 2 presents simple design solutions that basically make the algorithms more generic. However, they can not answer some recurrent situations. Section 3 explains that algorithms may need to add some data to the parameters they receive, but in order to maintain the encapsulation of such algorithms, it can not be expected from the caller of the algorithm to prepare the parameters for these additional data. Some solutions are proposed to keep the algorithm generic. Section 4 explains that, sometimes, a problem can be modeled by several ways, and that different models of the same problem can be managed at the same time. Solutions are proposed to facilitate the shift from one model to another, and the exchange of data between them.

1 Generic Data Structures

1.1 Inheritance versus Genericity

To design efficient and generic data structures, the solution commonly used is the genericity that provides not a single class but a parameterizable class, i.e. a metamodel, for a data structure. It describes how the class should be, some of the data types it manipulates being parameters. Hence, for different sets of parameters, the metamodel is instantiated, providing parameterized classes, for which the full compilation and optimization process is performed, completely identical to equivalent handwritten classes. That means genericity in a design has no direct impact on its efficiency.

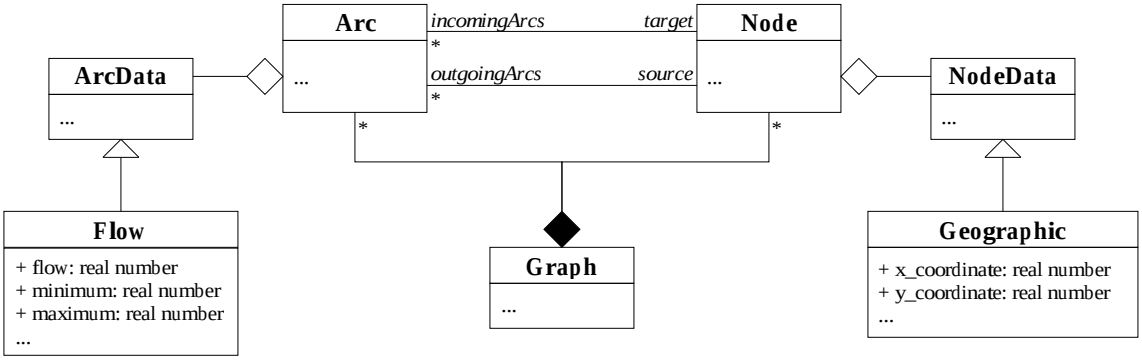

Figure 1: Graph model with inheritance.

Let consider a class `Graph` that represents a graph in operations research (a graph is composed of nodes and arcs, where each arc links two nodes). The aim is to provide a data structure that can be used to model various kinds of graphs, e.g. flow graphs that model flows moving from point to point or geographical graphs that model spots with coordinates and roads that separate them. That means the data structure must be able to carry various data types on both the arcs and the nodes of the graph.

A model using inheritance is proposed in Figure 1 (in this article, all the diagrams are presented with UML, cf. [9]). It defines superclasses `NodeData` and `ArcData` that represent the data carried respectively by the nodes and the arcs of the graph. In the two previous examples, that means subclasses `Flow` (to model flow graphs) and `Geographic` (to model geographical graphs) must be defined. An algorithm manipulating, for instance, flow graphs will expect the data on the arcs to belong to the class `Flow`. Thus, to use this algorithm for another kind of graphs, the `Flow` class must be inherited and some of its methods overridden. Due to the nature of the algorithms developed on such graphs, these methods should be called very often and dynamic polymorphism for them may lead to inefficiency.

Thus, a model with genericity is proposed in Figure 2. The classes `NodeData` and `ArcData` become *concepts* (or *interfaces*), cf. [2], of respectively the parameters `TN` and `TA` of the metaclass `Graph`. This way, flow or geographical graph types can be instantiated simply after defining classes `Flow` and `Geographic`, which must satisfy respectively the concepts `ArcData` and `NodeData` so the graph can handle them. The major drawback of this approach is that all the nodes (respectively all the arcs) must carry the same kind of data. It is a recurring problem when considering inheritance versus genericity to design a data structure. However, the generic design can be combined with the inheritance design, but the efficiency gained by the former will be lost by the dynamic polymorphism of the latter.

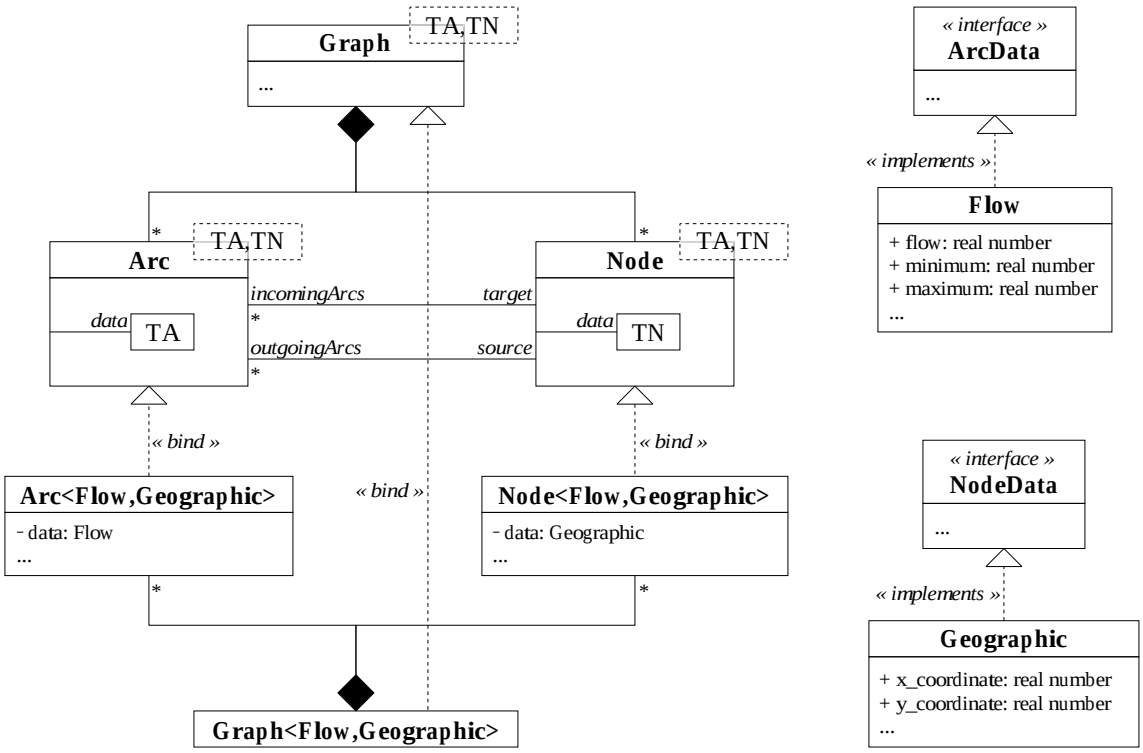

Figure 2: Graph model with genericity.

1.2 Independence from Data Structures

The design of data structures with the generic approach proved to be efficient (e.g. the STL [2]). But this is not enough to ensure the independence of the algorithms from the data structures they handle. To achieve this purpose, the solution commonly used is to propose one or more classes that become interfaces between the algorithm and the data structure it handles. Usually, an interface is proposed for each kind of operations on the data structure. For instance, to search through a data structure, the well-known *iterator* interface (cf. [5]) is used. We can also imagine an interface to access global informations on the data structure such as its size.

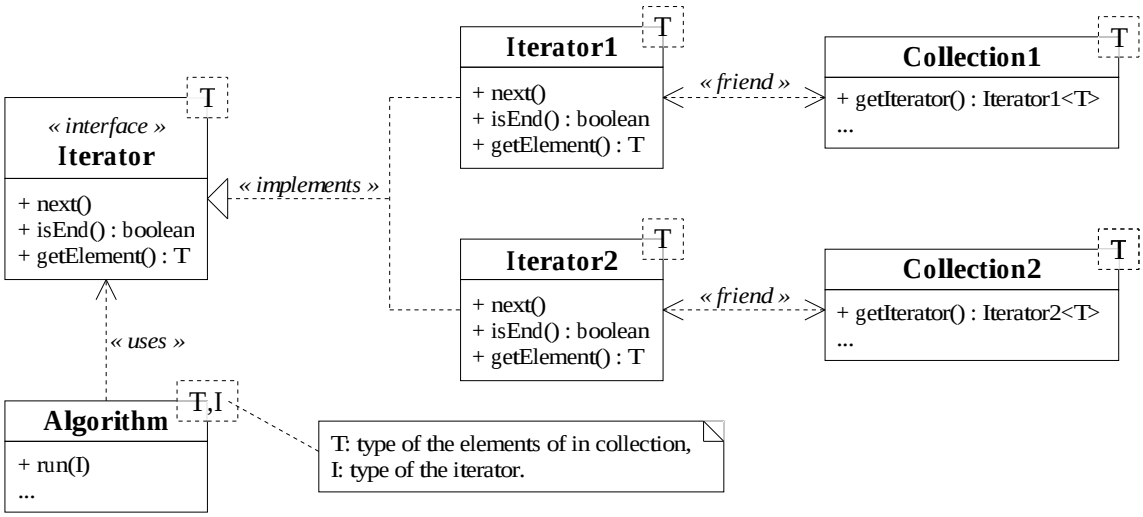

Figure 3: Algorithm parameterized on the iterator type.

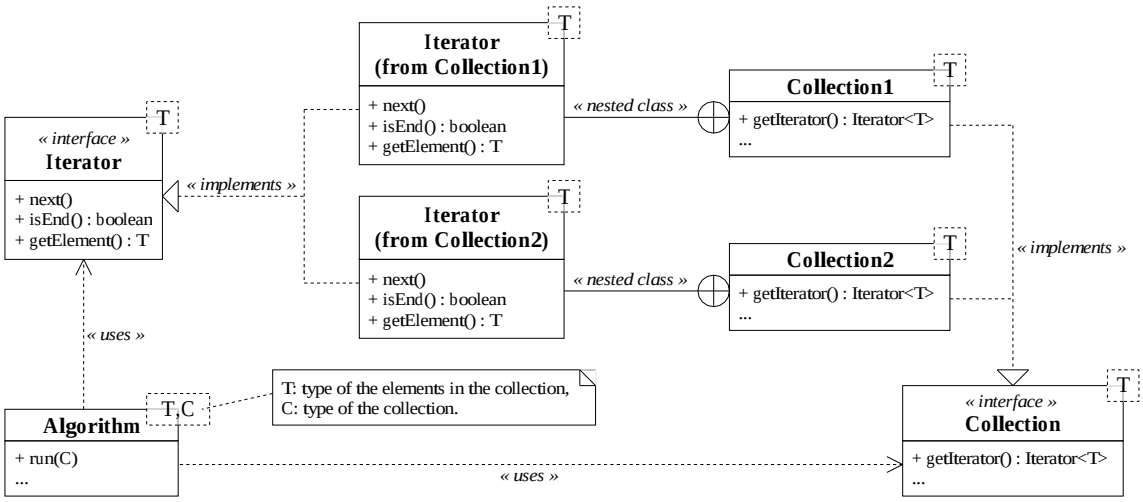

Figure 4: Algorithm parameterized on the data structure type.

In our discussion, we will consider a single iterator interface, but it can be generalized to any other interface. Some designs propose that the algorithm receives directly the iterators instead of the data structure, this way the algorithm is completely independent of the data structure, e.g. a collection (cf. Figure 3). However, if many iterators are required by the algorithm, the caller must provide all of them, which leads to a partial break of the encapsulation of the function: details have to be known from the caller so it provides the relevant iterators. The following example illustrates how to use the collection from the modeling of Figure 3:

```
method Algorithm<T,I>::run(I i)
 while not i.isEnd() do
  ...i.getElement()...
  ...
  i.next();
 end while;
end method;
```

A better design would be to propose a parameterizable version of the algorithm where the parameter is the type of the data structure the algorithm handles (cf. Figure 4). The collection is still provided to the algorithm, but the meta-algorithm is independent of it. However, the data structure needs to implement a specific concept: with the iterator example, the collection must provide methods that create iterators on its own structure. The type of the iterator must also be provided by the data structure as shown in Figure 4 with the nested type `Iterator`. It means a completely independent collection will need an adapter before the algorithm can use it. The following example illustrates how to use the data structure from the modeling of Figure 4:

```
method Algorithm<C,T>::run(C c)
 c.Iterator i = c.getIterator();

 while not i.isEnd() do
  ...i.getElement()...
  ...
  i.next();
 end while;
end method;
```

2 Toward Generic Algorithms

The previous section explains how to make an algorithm independent of the data structures it handles. The same way, this section recalls a design solution to make it independent of the algorithms it manipulates. In a second part, various techniques are discussed to make an algorithm extensible.

2.1 Abstraction of Algorithms

As shown in the previous section, an algorithm can be modeled as a class with a method, `run()` for instance, that is called to perform the algorithm. Moreover, such a class can aggregate the parameters of the algorithm, thus instances will represent the algorithm with different parameters. Based on this modeling, the design pattern *strategy* (cf. [5]) allows to make components independent of an algorithm: as it is represented by a class, it is possible to define an abstract superclass to gather all the algorithms that solve a same problem. As shown in Figure 5, the classical problem of the shortest path between two nodes in a graph (`ShortestPathAlgo` abstract class) can be solved with various algorithms (cf. [1]): `BellmanAlgo`, `DijkstraAlgo`...

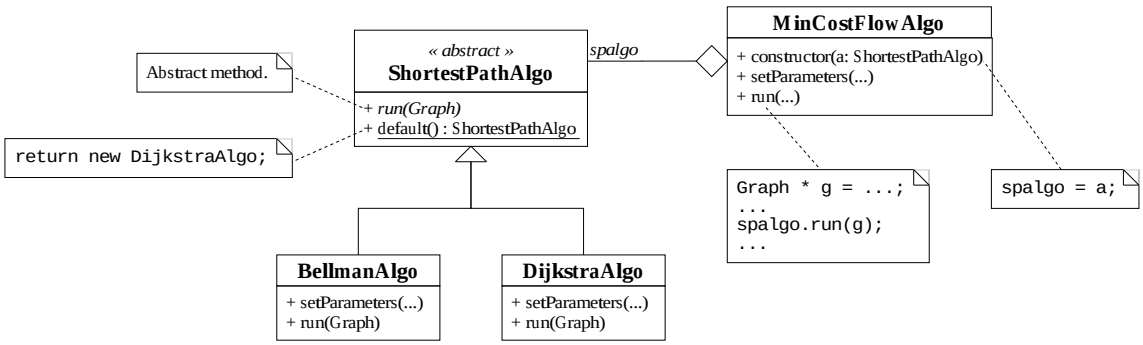

Figure 5: Abstraction of algorithms.

The `run()` method of the `ShortestPathAlgo` class is abstract so the subclasses must override it. This way, the different shortest path algorithms become interchangeable in any algorithm that manipulates `ShortestPathAlgo` (e.g. `MinCostFlowAlgo`). The virtuality implied here will not impact the whole efficiency of the design, because the algorithms are supposed to have complex behavior, so the time requested in the call mechanism to the method is insignificant compared to the execution time of the method itself. However, as `ShortestPathAlgo` provides a common interface for all the algorithms, it can not be used to parameterize the algorithms. A specific method must be added to each algorithm, e.g. `setParameters()`, whose duty is to initialize the parameters of the algorithm.

```
BellmanAlgo     s = new BellmanAlgo;
MinCostFlowAlgo f = new MinCostFlowAlgo(s);

s.setParameters(...);
f.setParameters(...);
f.run(...);
```

The example above, based on the modeling of Figure 5, shows that it is possible to decide which shortest path algorithm to use inside the minimum cost flow algorithm at the execution time. The algorithm must be created and parameterized before it can be used in the method `run()` of the minimum cost flow algorithm. Note that the duty of the `setParameters()` methods can be performed by the constructors of the classes. It is also important to provide a method in the

abstract class `ShortestPathAlgo` to return a default algorithm object of one of its concrete class to the final user. Basically, it will be the class that is recognized to be the most efficient, but we can imagine a more sophisticated approach where, for instance, an analysis of the structure of a graph allows to select the best algorithm to solve a specific problem on this graph.

2.2 Extension of Algorithms

This section discusses three ways of making an algorithm extensible, the idea being that some parts of its code are delegated in separate methods that can be replaced by the user. This way the behavior of the whole algorithm can be modified, whereas most of its code is not (and can not be) altered. Moreover, the user does not need to know all the details about the implementation of the algorithm, only relevant information on the methods he can replace is necessary.

2.2.1 Virtual Method Approach

The design pattern *template method* (cf. [5]) is a classical solution to make an algorithm extensible. It externalizes parts of the method `run()` of an algorithm into virtual methods, e.g. `operation1()` and `operation2()` in Figure 6, called the *parameter methods*. Hence, through inheritance, these methods can be overridden to modify their behavior, leaving the body of `run()` unchanged.

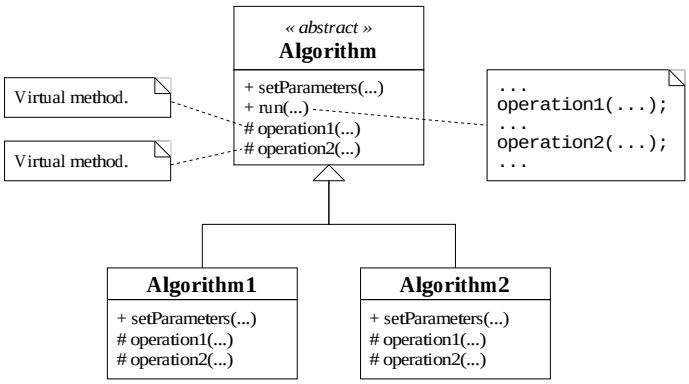

Figure 6: Extension of an algorithm, virtual method approach.

The major drawback of this approach is obviously the use of the dynamic polymorphism that may lead to inefficiency, especially when the parameter methods are fast and often called. Another disadvantage is the rigidity to extend an algorithm: it is impossible at the execution time to propose an extension of the methods but those defined by the subclasses of the algorithm.

2.2.2 Abstract Visitor Approach

To make the extension more flexible, the notion of *visitor* is introduced in [5]. It proposes to embed the parameter methods into objects. More precisely, a visitor possesses methods that match the parameter methods. To be operational, the algorithm must aggregate a visitor, which provides the missing parts in its `run()` method. The visitor can be provided to the algorithm during its construction, or later, before the call to the `run()` method, or even as argument of the `run()` method. In Figure 7, inside the `run()` method of the `Algorithm` class, a `visitor` object that implements the `Visitor` interface is used to call its embedded parameter methods `operation1()` and `operation2()`.

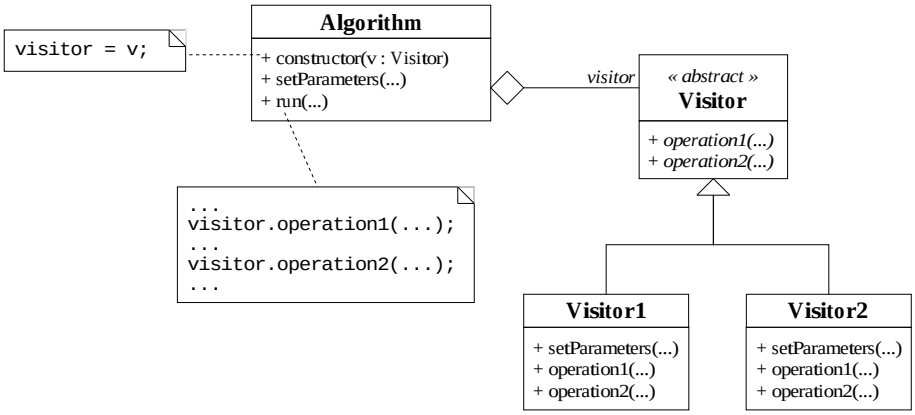

Figure 7: Extension of an algorithm, abstract visitor approach.

The following example shows the flexibility of this approach. It is possible to decide, during the execution, which visitor to use to run the algorithm. However, the major drawback, due to the virtuality of the parameter methods, remains.

```
Visitor1  v = new Visitor1;
Algorithm a = new Algorithm(v);

v.setParameters(...);
a.setParameters(...);
a.run(...);
```

2.2.3 Visitor Interface Approach

To finally avoid the dynamic polymorphism, the visitor must become a parameter, not of the `run()` method, but of the `Algorithm` class itself. That means the class becomes parameterizable with the type of the visitor as parameter. Thus, as shown in Figure 8, the algorithm aggregates a visitor that must satisfy a `Visitor` concept.

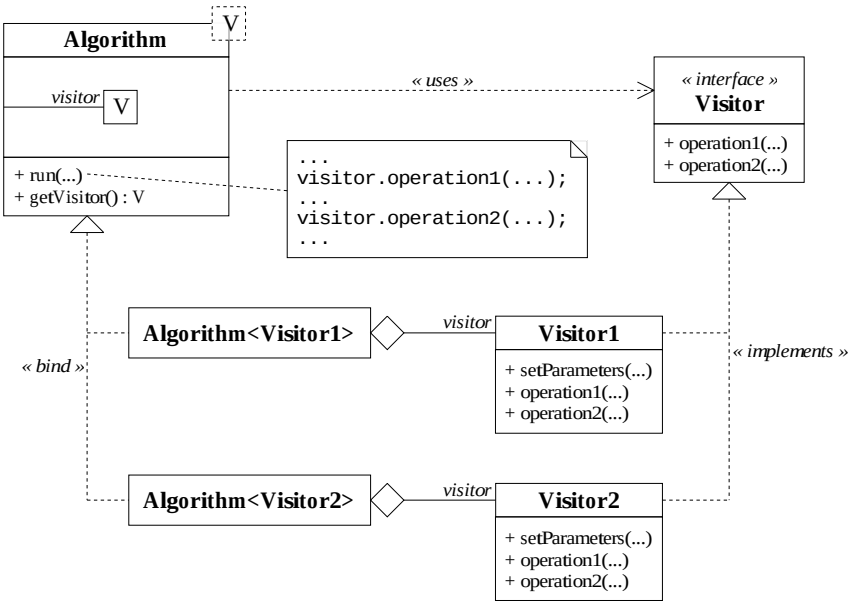

Figure 8: Extension of an algorithm, visitor interface approach.

The following example illustrates this modeling. This approach is very similar to the previous one, but the dynamic polymorphism is avoided, thus there is no loss of efficiency. Nevertheless, the flexibility proposed with the abstract visitor approach is lost. As in the previous approach, the visitor could be provided directly as an argument of the constructor of `Algorithm`, instead of being automatically created by the algorithm. This way the visitor interface approach can be combined with the abstract visitor approach to provide flexibility.

```
Algorithm<Visitor1> a = new Algorithm<Visitor1>;

a.getVisitor().setParameters(...);
a.setParameters(...);
a.run();
```

2.3 Conclusion

To obtain a "good" genericity of the algorithms, it seems important to apply the strategy pattern with the solutions proposed here to make the components independent, and to combine it either with the visitor interface approach (when efficiency matters), or with the abstract visitor approach (when flexibility is preferred), to allow sufficient extensibility for the algorithm. Note that the STL proposes the notion of *functor* that is similar to the notion of visitor interface. Other approaches are proposed, for instance in [4] that defines generic versions of many behavioral design patterns introduced in [5].

3 Managing Extensions of Data Structures

When designing generic algorithms, it is often necessary to extend a data structure, so its elements provide additional attributes that an algorithm may temporarily need. For instance, to solve a minimum cost flow problem some algorithms require to affect a potential to the nodes of the graph. However, the nodes of flow graphs do not possess such data and it can not be expected from the caller of the algorithm to add this data, it would break the encapsulation of the component.

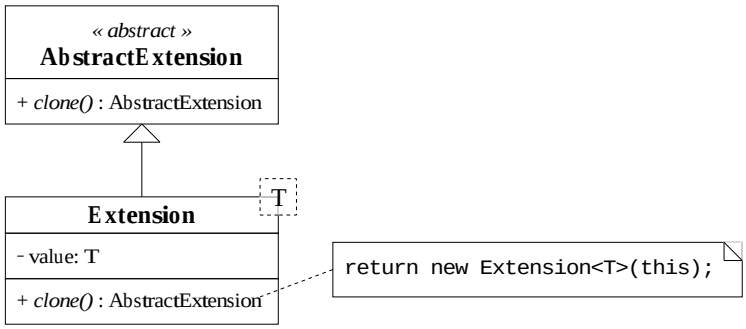

Figure 9: `Extension` class.

The first idea is to add a "free" attribute to the nodes of the graph. This attribute is a reference to an object of a class `AbstractExtension`. When a graph is built, no data is pointed. Then, if an algorithm needs to add data, it can make the "free" attribute reference an object belonging to a class inherited from `AbstractExtension`. The parameterizable class `Extension` is proposed to offer a generic way of encapsulating an entity inside an object with the interface `AbstractExtension`.

This first approach requires that before an algorithm uses the "free" attribute it must memorize the data another algorithm may have stored in it and restore it once its process finishes. It is typically a stack, so the second idea is to replace the "free" attribute by a stack of "free" attributes. When an algorithm needs to add data, it must put it on top of the stack and remove it after its process.

However, the execution of some algorithms may be iterative, and between two iterations other actions can be performed. For instance, for the learning with pruning of a neural network, there are two independent iterative algorithms: the learning algorithm that modifies, at each iteration, the weight of the neural network, and the pruning algorithm that may remove, at each iteration, some arcs that prove to be useless. The whole process is to perform some iterations of the learning, then one of the pruning and repeat until certain conditions are satisfied. Both the learning and the pruning need to put additional data on the nodes of the graph that must remain between two iterations of each algorithm. That means the order in which the algorithms add data to the nodes can not be modeled as a stack. Any algorithm can add or remove, at any time, its own data on the nodes.

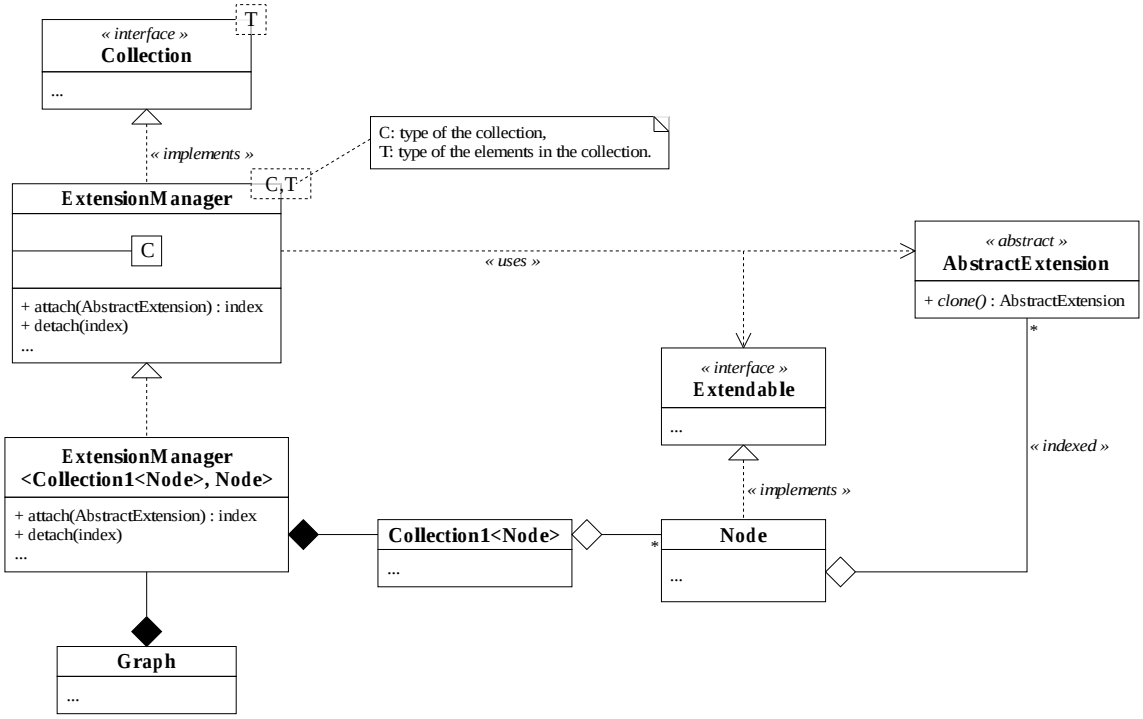

Figure 10: Additional data management modeling.

Figure 10 presents a solution to design a data structure that can manage the insertion or the removal of data on a set of elements. The example of a collection of nodes in a graph is considered. Instead of manipulating a collection of nodes directly, e.g. with the `Collection1<Node>` class (cf. Figure 4), the graph handles an object of the class `ExtensionManager` that implements the `Collection` interface, so no changes in the code of the `Graph` class is required (except for the declaration of the collection of nodes). This `ExtensionManager` class is an adapter that aggregates a collection; any call to methods of the `Collection` interface is delegated to its inner collection.

The duty of an `ExtensionManager` object is to manage the addition or the removal of an `Extension` object for each node of the set it encapsulates. Hence, an algorithm, that wants to add an extension, calls its `attach()` method with a template of the extension to clone and to

put on each node. An index is returned indicating the location of the extension in the indexed lists the nodes aggregate. Thus, it allows the algorithm to directly ask a node for a specific extension, using this index. Finally, to remove an extension, the algorithm calls the `detach()` method with the index as argument so the manager knows which extension to remove from the nodes. To manipulate the sets of extensions, the manager uses the `Extendable` interface implemented by the nodes.

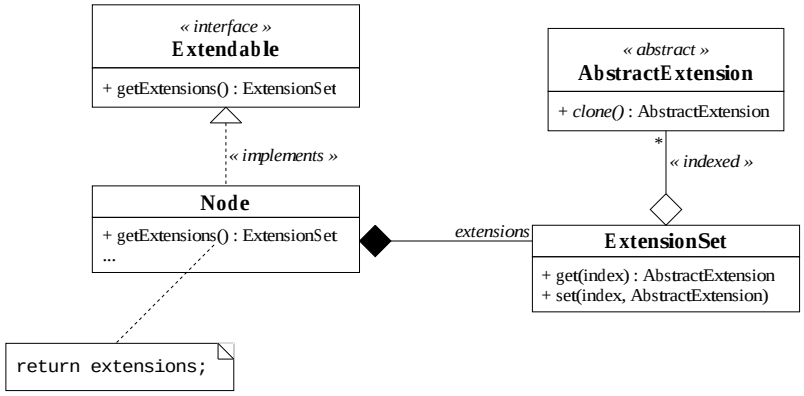

Figure 11: `Extendable` interface implemented with delegation.

Two solutions are possible to implement the `Extendable` interface. First, the `Node` class can delegate the management of the extension list to another class, e.g. `ExtensionSet` in Figure 11. Second, the interface `Extendable` can become a class that manages the extension list (as the `ExtensionSet` in the previous design), and the `Node` class inherits from `Extendable`, cf. Figure 12. This specialization is efficient because no dynamic polymorphism is implied. In this second design, implementation is inherited from the `Extendable` class. It arises problems with languages that forbid multiple implementation inheritance, that means `Node` can not inherit implementation from another class, which may be important for some designs.

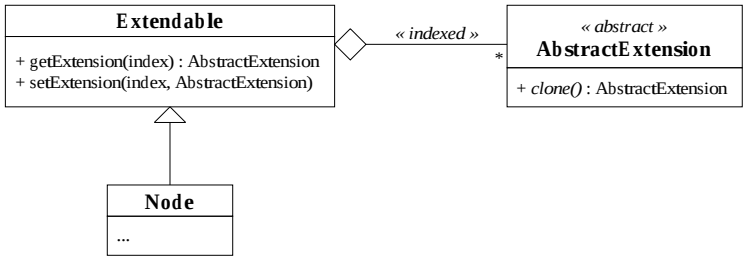

Figure 12: `Extendable` interface implemented with specialization.

In terms of maintenance and reusability, this design allows to add an `ExtensionManager` without modifying the data structure that aggregates the original collection, e.g. `Graph`. The structure of the class that aggregates the extensions has not to be modified, it only requires to aggregate an object of the `ExtensionSet` class or to inherit from the `Extendable` class, both solutions providing the set of extensions. Dynamic polymorphism has been avoided as much as possible, however algorithms need to downcast the extensions from `AbstractExtension` to `Extension<T>`. That means some type checking at the execution time is required, which usually leads to inefficiency because the extensions are single data that we can reasonably assume to be called very often. The efficiency is ensured only if this type checking is avoided, which is possible with langages such as C++ that proposes the checking (with its instruction `dynamic_cast`)

or not (with its instruction `static_cast`).

4 Maintaining Several Models of a Same Problem

Another recurring problem is to deal with several models of a problem at the same time. For instance, a graph can also be modeled as a matrix, and algorithms often need to manage a `Graph` object and the equivalent `Matrix` model. An algorithm may need to convert the `Graph` object into a `Matrix` object, perform some optimization, and data from the `Matrix` object must be interpreted to modify the `Graph` object. The simplest solution is a converter object that provides a method to transform a graph into a matrix, and another to interpret the results back from the matrix to the graph. The drawback of this approach is obvious: each time a modification is made on the graph, the converter object must be called to rebuild completely the matrix.

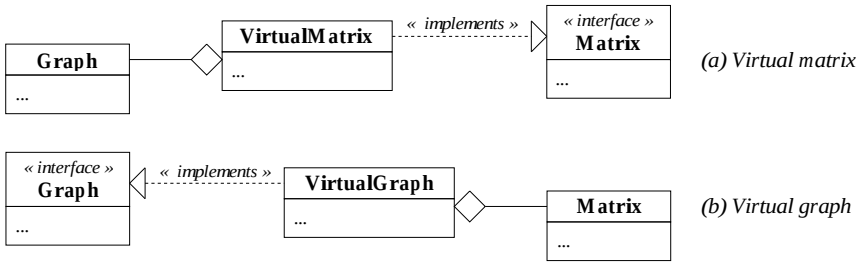

Figure 13: Virtual data structure.

A second design solution can be proposed: one of the two components, `Graph` or `Matrix`, can be "virtual". This means only one of the two data structures physically exists, and the other one is simply an adapter of the other. Figure 13(a) proposes a `VirtualMatrix` class that implements the `Matrix` interface and aggregates the graph to convert. Each time a method of the `Matrix` interface is called, the `VirtualMatrix` object delegates the execution to its associated graph.

On the contrary, Figure 13(b) proposes to make the graph structure virtual. With this design solution, the graph (respectively the matrix) can be modified at any time because each time information for the matrix (respectively for the graph) is requested, it is built from the graph (respectively from the matrix) structure. However, if the methods of the virtual component need time to be executed, the design is inefficient. It should be used when the methods are fast, the best being that they only provide a mapping from matrix (respectively from graph) elements to graph (respectively to matrix) elements.

The third solution consists in maintaining several physical models of a problem at the same time. It means there are two classes `Graph` and `Matrix`, and when a modification occurs in the `Graph` object, it must be reflected in the `Matrix` object (for the sake of simplicity, the opposite case is not considered). That means algorithms will manipulate an adapter of the `Graph` class, e.g. `ObservedGraph`, that follows the same `GraphInterface` interface. Moreover, the design pattern *observer* (cf. [5]) is implemented for the observed graph, which means external objects, the observers, can ask observer managers in the observed graph to be informed when certain operations occur. The observed graph decides to notify its relevant changes to one or more of its observer managers, which inform then the observers. An observer, receiving a notification, can decide to modify the matrix in order to keep the coherence with the graph. Thus, any algorithm can manipulate the `ObservedGraph` as any graph, and each time a relevant operation occurs, the matrix is modified. This solution is efficient only if a few calls to the observers are performed.

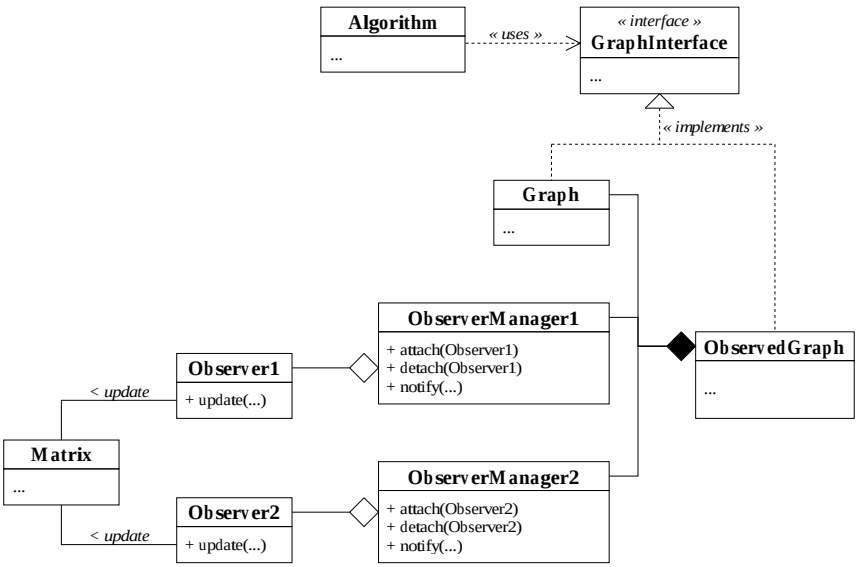

Figure 14: Maintaining several models.

To conclude, the first solution can be used only if the models are not requested to be maintained together. If a direct mapping between the elements of the two models can be achieved, the second solution with a single adapter is efficient. Finally, if the relation between the models is more complex and two physical models are necessary, the third approach should be chosen. In all these solutions, dynamic polymorphism has been avoided.

Conclusion

The object-oriented paradigm focusing more on the data than on the behavior of a program, it seems better suited to build reusable data structures than reusable algorithms. However, it is possible to provide design solutions to model generic and efficient algorithms combining the object-oriented and the generic programming. This article presents how to make a component independent of both the data structures and the algorithms it handles. It also explains how to bring extensibility in the code of an algorithm, without losing its efficiency. We also discuss recurring problems, such as how to manipulate additional information on data structures without losing neither the efficiency nor the genericity of the design; and such as keeping several models of a problem up-to-date at the same time. The design solutions presented along this article have been implemented and their reusability and extensibility experienced successfully in several operations research projects.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows - Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999.

[3] Bruno Bachelet, Philippe Mahey, Rogério Rodrigues, and Luiz Fernando Soares. Elastic Time Computation for Hypermedia Documents. In *SBMidìa'2000*, pages 47–62, 2000.

[4] Alexandre Duret-Lutz, Thierry Géraud, and Akim Demaille. Generic Design Patterns in C++. In *6th USENIX Conference on Object-Oriented Technologies and Systems*, pages 189–202, 2001.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] Ralph E. Johnson and Jonathan Zweig. Delegation in C++. In *Journal of Object-Oriented Programming*, volume 4-11, pages 22–35, 1991.

[7] Stanley B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.

[8] Antoine Mahul and Alexandre Aussem. Neural-Based Quality of Service Estimation in MPLS Routers. In *Supplementary Proceedings of ICANN'03*, pages 390–393, 2003.

[9] Pierre-Alain Muller. *Instant UML*. Wrox Press, 1997.

[10] David R. Musser and Alexander A. Stepanov. Generic Programming. In *Lecture Notes in Computer Science*, volume 358, pages 13–25. Springer-Verlag, 1989.

[11] Martin J. O'Riordan. Technical Report on C++ Performance. Technical report, International Standardization Working Group ISO/IEC JTC1/SC22/WG21, 2002.

[12] Loïc Yon, Alain Quilliot, and Christophe Duhamel. Distance Minimization in Public Transportation Networks with Elastic Demands: Exact Model and Approached Methods. In *21st IFIP TC 7 Conference*, 2003.