Data Accessor

Provide data abstraction to enhance reusability.

Author[1] Marco Nissen
Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken
E-mail: marco@mpi-sb.mpg.de
URL: http://www.mpi-sb.mpg.de/~marco
Copyright © 1999 by Marco Nissen, Max-Planck-Institut für Informatik, Germany.
Permission granted for EuroPLoP for reprint.

Thumbnail *Data Accessor* intends to enhance reusability and flexibility of algorithms[2] that need to access data. If an algorithm uses a standardized intermediate data structure it is possible to reuse it for other ways of accessing data only by adapting the *Data Accessor*. The algorithm remains the same. This pattern abstracts data access as the iterator pattern abstracts structural access.

Also Known As Data Manager.

Example Imagine a car navigation system that aims to find the best route between two cities according to a certain criteria. In detail, there are different possibilities for the criteria, for example the following: "shortest distance", "shortest driving time" and "lowest cost" (according to gas consumption). The network will be represented by nodes and edges, where each node corresponds to a city and each edge to a street. An edge additionally contains a C-struct which again contains a `double` value for each of the three values from above. An algorithm that computes shortest paths between two cities may then easily be parameterized just by demanding a pointer-to-member variable (in `C++`.)

The question is, what happens if the data is not stored explicitly, but implicitly, i.e. has to be computed on-the-fly ? Assume for sake of argument that the driving time depends not only on the distance between two cities, but also on the current speed of the car at the current street. To compute the shortest paths for shortest driving time, we therefore had to proceed in two phases: first, compute all driving times explicitly, second, run the algorithm.

The other possibility is to adapt the algorithm such that we have different algorithms for different data, or one major algorithm that has case-statements whenever data has to be computed. However, this complicates resulting code and probably introduces new errors.

---

[1] Pattern form adapted from [POSA96].
[2] An algorithm is said to be reusable if it can be re-used in different environments *without* changing the actual code. An algorithm is said to be flexible if its behavior can be adapted *without* changing the actual code.

Context *Data Accessor* is applicable if you are developing algorithms that rely on complex data structures to be iterated and on attributes of the objects in the data structures, that are unknown at the time of programming the algorithm. More specifically, you cannot anticipate how the attributes are implemented.

Problem Algorithms that use underlying data structures require changes in the algorithm-code when the data structures change. Data structure wrappers cause improper conversion overhead[3].

How can algorithms be implemented to be reusable in different environments ?

Forces On the one hand, reuse of algorithms cause new problems (see problem section). On the other hand, a different design approach might cause unwanted overhead. On may observe that most algorithms when accessing data exploit knowledge about the implementation of underlying datastructures, i.e. they do something that is called "white box reuse". On the contrary, we like to permit "black box reuse", i.e. we assume nothing about the implementation of underlying data structure.

Basically, this pattern tries to balance between

- maximizing maintainability by reducing code complexity
- maximizing reusability by introducing black box reuse
- maximizing flexibility by permitting easy exchange of data structures and adaptation of algorithm behavior
- minimizing performance overhead and increased number of classes and objects caused by the additional indirection
- minimizing loss of efficiency due to decoupling is a point of concern, because some algorithm can only operate if they know the concrete implementation.

---

[3] see the discussion of *Adapter* in the section *Related Patterns*

Solution The idea is to provide one handler object for each single attribute, e.g. one for colors, one for length and so on. We decouple data access from the algorithm by introducing an additional indirection that manages the access.

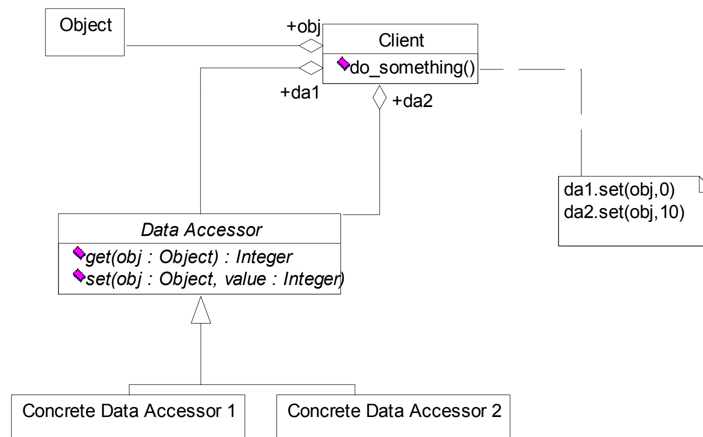
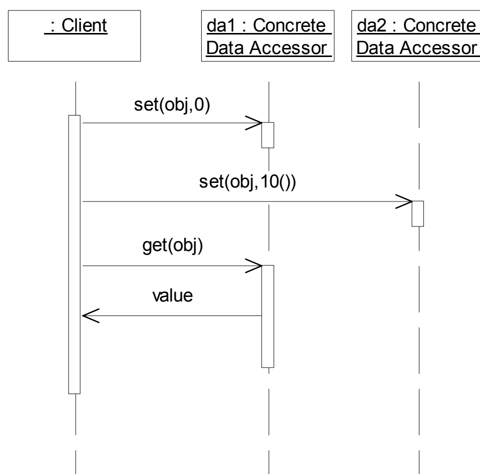


Figure 1: Abstract view on *Data Accessors*.

In the UML diagram in figure 1, an algorithm uses two handler objects *da1* and *da2* , both inherited from "Abstract *Data Accessor*". `da1.set(obj,0)` sets the value corresponding to the first *Data Accessor da1* and object *obj* to . `da2.set(obj,10)` sets the value corresponding to the second *Data Accessor da2* and object *obj* to 10. Note that *da1.get(obj)* and *da2.get(obj)* generally give different results.

In the scenario from the example section the solution looks like this: network algorithms often use several independent attributes for nodes and edges while it is not clear if they are all stored in a single structure.



With *Data Accessors* it is more convenient to express the idea of attributes since we do not have to commit ourselves to a certain implementation decision.

In the sequence diagram on the left, we will see how the methods are called in both *Data Accessor* and that the last computed value will be 0.

Data Accessor

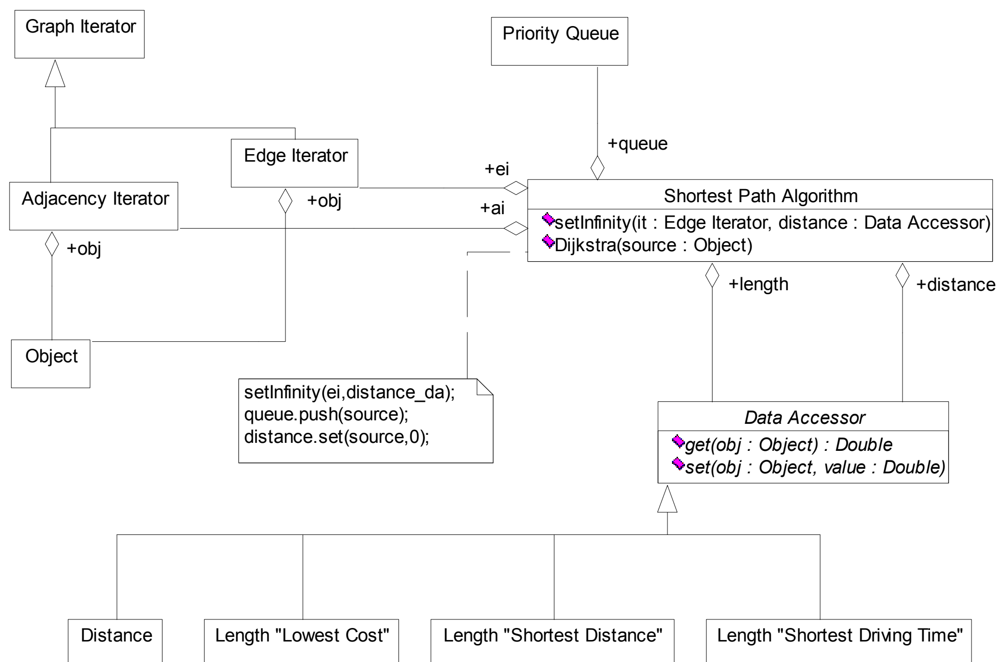


Figure 2: Illustration of expressing attributes with *Data Accessors*: each kind of "Length"-*Data Accessor* fits for the algorithm.

In the UML diagram in figure 2, the algorithm code is represented by a shortest path algorithm that is invoked with the parameter *source*. *Source* and *ai* are objects of type *adjacency iterator* which traverses the adjacency structure of a graph. *Edge iterator* traverses the set of edges of a graph. *queue* is a priority queue that is needed by the algorithm. Now, the algorithm uses two *Data Accessors*, *length* and *distance* to store these values. In the beginning of the algorithm, all distances are initialized to infinity and that of the source node to zero. The rest is a straightforward implementation of a shortest path algorithm (Dijkstra, in our case).

The point here is the application of the two *Data Accessors* which make the algorithm independent of how data access is done. For example, computing length values for edges in the graph can be exchanged easily by replacing the length *Data Accessor*, i.e. if the algorithm computes shortest paths according to shortest distance, we can also compute shortest paths according to lowest cost just by a simple replacement.

Sample Code *Java™*

In Java, it is possible to implement *Data Accessors* as classes that implement the following interface[4]:

```
public interface DataAccessor {
  public Double get (Object obj);
  public void   set (Object obj, Double value);
 }
```

A very simple *Data Accessor* may look like this:

```
public class LengthEuclideanDistance
 implements DataAccessor {
  public Double get (Object obj) {
    return ...
    // compute Euclidean distance value for obj
   }
  public void set (Object obj, Double value) {
    // store the value
   }
 }
```

Similarly, different *Data Accessors* can be written for "shortest driving time" (`LengthShortestTime`) and "lowest cost" (`LengthLowestCost`).

An application of the *Data Accessors* may look like this (DataAccessor pred will be used for storing the predecessor relation of a shortest path tree):

```
DataAccessor distance=new DistanceAccessor();
DataAccessor length=new LengthEuclideanDistance();
DataAccessor pred=new PredecessorAccessor();
Algorithm(distance,length,pred);
```

---

[4] this example is taken from [NW96]

C++

The implementation for Java can be used in a similar way for C++, as well. The base class looks like this[5]:

```
class DataAccessor {
public:
   double get(Object obj);
   void set(Object obj, double value);
 };
```

For sake of argument we want to permit something like the following code, which prints all data that is stored in a certain sequence (like array or list):

```
void printData(Iterator it, Iterator end,
               DataAccessor da) {
  while(it!=end) {
    cout << "Value at position ";
    cout << it.getPos() << ": ";
    cout << da.get(it);
    cout << endl;
    ++it;
   }
 }
```

Say, we want to compare an algorithm that works on precomputed values with the case in which we compute all values on-line. With this pattern, it will be very easy: write one algorithm (like the one above) that takes two iterators for the structural access and one *Data Accessor* for data access. The rest consists of two implementations of *Data Accessors*, one for the case in which all values were precomputed and one for the other case in which all values were computed on-line.

If the container is a parameterized array and if it has two methods for data access (getData and setData) we get the following:

```
class ArrayDataAccessor : DataAccessor  {
  array& MyArray;
public:
  ArrayDataAccessor(array& A) : MyArray(A) {}
  double get(iterator it) {
    return MyArray.getData(it); }
  void set(iterator it, double val) {
    MyArray.setData(it,val); }
};
```

---

[5] Object is the base class for all objects that are to be used by the class *DataAccessor*.

The other case looks like this (here, Functor is a class that implements `operator()`):

```
class OnlineDataAccessor : DataAccessor {
  Functor functor;
public:
  OnlineDataAccessor(Functor f) { functor=f; }
  double get(iterator it) {
     return functor(it.getPos()); }
  void set(iterator it, double val) { }
};
```

The simple algorithm from above can now be called for a precomputed array:

```
array A;
ArrayDataAccessor ada(A);
printData(A.begin(),A.end(),ada);
```

…or for the other case[6]…

```
SpecialFunctor f;
OnlineDataAccessor oda(f);
PrintData(A.begin(),A.end(),oda);
```

Actually, it is possible to use the template mechanism of C++ intensively to get maximal efficiency and reusability[7].

Resulting Context

Maintainability: code that is based on *Data Accessors* separates data access from the actual algorithm code. One can concentrate on the algorithm without bothering about how data access is actually done. If we have different ways of data accesses like array access and "on-the-fly"-computation it should be clear that different implementations lead to different versions of the algorithm unless we provide a uniform interface to both ways. This is what is done by introducing *Data Accessors*. Once we have implemented a set of *Data Accessors* for a fixed set of corresponding objects we may reuse them without any effort, just by exchanging the class.

---

[6] SpecialFunctor does the mentioned on-line computation and it is derived from Functor

[7] Actually, this can also be implemented in GJ, a generic version of Java -- see [BOSW97] and [NW96].

Reusability: if an algorithm is implemented using this pattern, changes in the underlying data structures only require changes in the used *Data Accessors*. Thus, the algorithm code remains unchanged when the implementation of a data access changes.

In the car navigation system from the example section, the application of design patterns can be very fruitful. We write one algorithm that is able to compute shortest paths between two locations and parameterize it with graph iterators for structural abstraction and *Data Accessors* for data access abstraction. For each of the possible attributes, we write one *Data Accessor* and the algorithm can be reused in every case.

Flexibility: suppose we have a *Data Accessor* for "driving time" on the edges of the network and the algorithm works together with it. If we like to add some plausibility checking procedure for the times, it can be easily done by writing a wrapper class for the *Data Accessor* that changes it into a checking one (application of the observer pattern). For example, it corrects all durations if they are less than three minutes. Another possibility is to add some caching to "on-the-fly"-computation *Data Accessors*.

Benefits of this pattern are:

- enhanced maintainability, reusability and flexibility
- algorithms can be adapted to future environments without changing the actual code of the algorithm
- adding attributes to existing objects does not result in changing the objects

Drawbacks of this pattern are:

- additional indirection and hence, more complicated structure
- increased number of classes and objects

On the one hand this pattern clearly introduces a more complex structure, for there are additional class dependencies which would not exist if the algorithm were implemented without *Data Accessors*. On the other hand, the code complexity introduced is very low, because the design is clearly very simple. On the performance side, it has been shown that the overhead caused by *Data Accessors* is very little if you compute shortest paths in a network generated by railroad data ([KNW97]: the overhead ranged from 10% to 68%). This promises to be an acceptable price in view of the benefits, if the drawbacks are not critical. Examples for critical situations are real-time systems or cases in which it can be anticipated that algorithms will never be reused.

Known Uses

- The graph iterator example illustrates the use of *Data Accessor* in the C++-Library of Efficient Data Types and Algorithms [LEDA].
- The Standard Template Library [MS96] provides access to containers via iterators but implements data access by using the *-operator of the iterator class. This results in restricting the access to one distinct data attribute per object of a container, which might be adequate for sorting algorithms but is definitely not for graph algorithms. The straightforward work-around for multiple attributes, i.e. container that store structures complicates the design and destroys reusability in algorithms (algorithms need to know the names of the member variables in the structure - or at least the offset). Another possibility is to use function pointers, but this leads to further complications and probably run-time overhead.

  If you are planning to write algorithms and associate only one attribute to a single structural entity of a container which is always stored explicitly in a storage cell, [MS96] is the design of your choice. In any other case you should prefer[8] *Data Accessors* in order to avoid code complexity.

Related Patterns

- **Iterator:** Iterator may be used for abstraction of structural access instead of a direct connection of underlying objects and *Data Accessor*. For example, in graphs there are nodes and edges and three (basic) types of iterators: node iterators, edge iterators and adjacency iterators (the latter traverse the neighborhood of fixed nodes). Here, a *Data Accessor* can be responsible for an attribute of nodes or edges. If '`color`' is a *Data Accessor* that represents the color attribute of nodes and '`it`' is a node iterator that traverses the node set of a graph, then it will be easy to access the correct attribute of the underlying node: '`color.get(it)`' (or '`color.set(it, value)`').
  This additional indirection for structural access has its drawbacks (indirection), but also benefits (higher flexibility) - this is discussed in more detail in the iterator pattern.
- **Adapter:** Adapter is similar in that it enhances reusability, but Adapter adapts signatures while *Data Accessor* provides a clean design that makes adapting interfaces superfluous whenever we have different interfaces for different ways of data access.

---

[8] However, it is possible to do complicate things in STL like having a collection of collections of attributes and iterate over the main collection. Data Access is done by accessing the elements of the sub-collection. Unfortunately, this is nothing more than a dynamic variant of the class in the example section.

There are the following drawbacks when using adapters:
Writing *adapters* to cope with different interfaces yields to erroneous and complicated software. Adapters also cause performance overhead because of the indirection code. For example, we have an algorithm `A`, a handler object `H` and some objects `O`. The algorithm uses the methods of the intermediate object `H` for accessing the attributes of `o` in `O`. Unfortunately, `H` needs to be changed if we want to re-use `A` in a different environment. Here, the interface of the handler object is as complicated as many attributes and objects types there are in the algorithm.

Additionally, adapter sometimes make objects loose their identity, which can be bad. More precisely, if an object adapter is used, the original object and new one which contains the other are different. In `C++`, where object-identity is often done by using "=", the according operator has to be overloaded.

- **Template Method:** Template Method is similar to this pattern, but it decouples primitive operations from algorithms while *Data Accessor* decouples data access from algorithms.
- **Property List [**SR98**]:** If you have to add an attribute to an existing class that uses a property list, you can add a new slot to it and simulate having a new attribute with the object. Algorithms that use *Data Accessors* express this demand in a descriptive way, i.e. they are not interested in the way data access is actually implemented. Therefore, property lists are a way for implementing the mechanism of *Data Accessors*. Adding a new slot in a property list to add a new attribute results in writing a new *Data Accessor* for it.

  However, replacement of *Data Accessors* by Property Lists commits the algorithm to the use of Property Lists, i.e. if somebody does not want to use them because of efficiency reasons, it would not be possible to adapt the algorithm without changing the code.

  More dramatically, the use of "anything" could eliminate type-safety.
- **Pseudo-Reference [A99]:** In the very special case in which an element A is associated with a single other element A', it is better to use this pattern. In this pattern, A is replaced by a pseudo reference Â, which knows not only A but also knows how to compute A' by calling an associated reference system. However, this mechanism complicates data access if multiple different data is associated with elements.

Idioms

- **Functor:** A functor is a technique for parameterizing algorithms with primitive computations, while the *Data Accessor* pattern addresses the more abstract question of data access. Sometimes *Data Accessors* are implemented similarly as functors. Note that *Data Accessors* are more like an idea and using functor is an implementation technique.
- **Function Pointer:** Function pointers provide a means of parameterization of algorithms like functors but is again a technique. If we provide one function pointer for "get"-access and one function pointer for "set"-access we get something which is virtually the same as a *Data Accessor*. The difference to the sketch-image lies in implementation, but not in the intent.
- **Type Parameterization:** The template mechanism in `C++` or genericity in Java [BOSW97] permit type abstraction in algorithms. This enables us to write parameterized stacks or parameterized sorting algorithms. This differs from data access abstraction, i.e. templates are a way of implementation for abstraction.

Credits I like to thank Peter Sommerlad for shepherding this pattern. Special thanks to the Writers Workshop at EuroPLoP for giving suggestions for improvement.

References

A99

Aguiar, Ademar:
Pseudo-Reference. Proceedings of the EuroPLoP '99

BOSW97

G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler:
Making the Future Safe for the Past: Adding Genericity to the Java Programming Language .
Proceedings of the 12th ACM Symposium on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '97)

GHJV95

E. Gamma, R. Helm R. Johnson and J. Vlissides:
Design patterns. *Addison-Wesley*, 1995.

KNW97

D. Kühl, M. Nissen, K. Weihe:
Efficient, adaptable implementations of graph algorithms.
WAE, Workshop on Algorithms Engineering, '97.

KW97

D. Kühl and K. Weihe:
Data Access templates. C++ Report, 9/7, 15 and 18-21, 1997.

LEDA

K. Mehlhorn and S. Näher:
LEDA - A Platform for Combinatorial and Geometric Computing.
Cambridge University Press, 1999.
LEDA homepage: `http://www.mpi-sb.mpg.de/LEDA/`.

MN97

K. Mehlhorn and S. Näher:
The LEDA Platform of Combinatorial and Geometric Computing.
to appear with *Cambridge University Press*, 1999.

MS96

Musser, David R. and Saini, Atul:
STL tutorial and reference guide : C++ [plus plus] programming with the standard template library. Addison Wesley, 1996.

NW96

M. Nissen and K. Weihe:
Attribute Classes in Java and Language Extensions. Konstanzer Schriften in Mathematik und Informatik, Universität Konstanz (66/1996)
Web: http://www.informatik.uni-konstanz.de/~ nissen/JavaPaper/

SR98

Sommerlad, Peter and Rüedi, Marcel:
Do-it-yourself Reflection. Proceedings of the EuroPLoP '98

POSA96

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal,
Pattern Oriented Software Architecture – a System of Patterns.
John Wiley and Sons, 1996.