

# Object-Oriented Design Pattern: Access Protector

Harri Hakonen and Ville Leppänen

Department of Information Technology, University of Turku and TUCS, Finland  
{Harri.Hakonen, Ville.Leppanen}@utu.fi

## Abstract

Access Protector is a class behavioral design pattern which distributes the routines of the original class hierarchy to given access-protection layers. The distribution utilizes dynamic binding and automatic upcasting to guarantee that the intended object access policies are transitive while the original runtime behavior between the objects does not change.

In more detail, the pattern states explicitly what rights the client and the supplier object have on the objects passed through the class interface. Rather than being a property of an object, Access Protector defines access levels, i.e. restrictive views, into it by using subclassing. This property makes the access restrictions transitive so that granted permissions cannot be loosened when the object reference is forwarded. Also, the object can be referred via multiple access levels at the same time depending on what kind of object integrity protection is required toward different clients.

Access Protector can also be called as Anti-Code-Wrapper or Routine Waiver. The name Anti-Code-Wrapper emphasizes the delegation aspect of the design pattern, whereas Routine Waiver refers to routine reorganization.

**Keywords.** Design pattern, object-oriented, protective view, access protection, immutability.

## 1 Introduction

The Access Protector pattern captures and generalizes object/subobject-level accessing solutions and ideas both from software industry and academic research. The pattern can be applied when, for example, one designs mutation permission categories which are resolved at compilation time. Furthermore, the pattern can be used to share the object's internal representation without compromising its integrity or class invariants.

The structure of this paper is as follows. In Section 2, we give an example where the integrity of a containing object depends strongly on its subobjects. The example, albeit it is simple, presents many problems in object sharing. In Section 3, we discuss these problems and show that object aliasing should be dealt with specific accessing mechanisms. The problem is stated in general terms in Section 4, along with its alternative formulations for the most occurring contexts. Access Protector is a complicated design pattern and, therefore, we present its solution, structure, and implementation form through a well-known layers of access privileges: mutable—partially immutable—deep immutable in Sections 5–7. The rationale is that this triplet is often encountered in class libraries and has also been actively researched. In Section 8, we restructure the class diagram of the example of Section 2 according to the Access Protector pattern. Also, we depict an object diagram to demonstrate that the dynamic behavior of the objects does not change in this process. Section 9 lists the benefits and the liabilities of the pattern, and Section 10 exemplifies the pattern use in software industry and in academic research. Finally, Section 11 relates the pattern with some other known patterns.

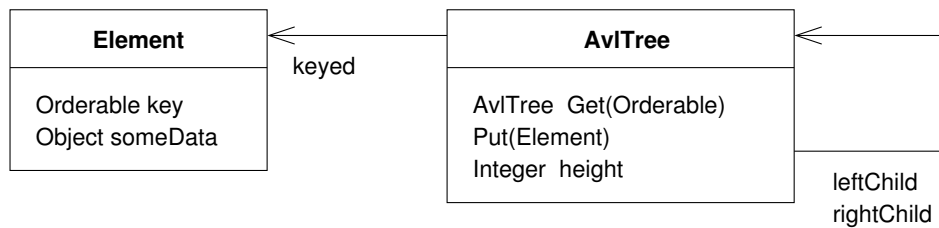


Figure 1: A class diagram for the AVL tree.

## 2 Example

To have a simple and self-sufficient example let us consider an implementation of an AVL tree, which is a balanced (binary) search tree where the height of the left and right subtree differ at most by one. The AVL tree has two invariant conditions, one for *ordering* the data and another for *balancing* the structure. Figure 1 illustrates a recursive implementation of the AVL tree where

- an *empty* AVL tree is represented by NIL reference, and
- a non-empty AVL tree is represented by a *node* containing an orderable element (*keyed*) and two children (*leftChild* and *rightChild*) which are AVL trees.

The height of the AVL tree is 0 for NIL and  $\max\{\text{(the height of the left child)}, \text{(the height of the right child)}\} + 1$  for a node. In addition, the AVL tree has routines *Get(Orderable)* and *Put(Element)* to fetch and to store a node containing the element with a given key. Note that the recursive definition of the AVL tree unifies each (sub)node to its corresponding AVL (sub)tree.

To elaborate, the class *Element* defines an association between a totally orderable key (*key*) and a satellite data (*someData*). A node of the AVL tree, i.e. a recursive *AvlTree* object, is identified by its *keyed.key* attribute. The *keyed.key* values also define the ordering of the nodes in the tree. Function *Get(Orderable key)* retrieves a reference to the node with the given key, and procedure *Put(Element keyed)* inserts a new node containing the *keyed* element at the proper position in the tree.

Although the preceding design captures the intended behavior of the AVL tree, it assumes implicitly that the client does not mutate the element keys nor the node ordering/balancing in the subtrees. In other words, the definition of the *AvlTree* class omits the issue of object sharing (or *aliasing*) and, at most, only documents the usage assumptions.

## 3 Motivation

As demonstrated in our AVL example, there are two cases which require references to the (internal) attributes of an object. In the first case, a caller object *c* invokes a routine of a target object *t* and inputs objects  $i_1, \dots, i_n$  via references. After the call, both *c* and *t* can refer to the  $i_j$  objects (and their subobjects). This is a common situation especially when we are creating object *t* or inserting more data into it. We call this an *input leakage* of the internal attributes. In the second case, *output leakage*, references to some internal parts  $o_1, \dots, o_m$  of object *t* can be given to outside, for example, by member routines. The input and output leakages can break strict data encapsulation, if the state of the referred object affects some other object's integrity constraints, i.e. its class invariant. In other words, the problem is how to protect the object's invariant when aliasing its subobjects.

Turning away from the target object's viewpoint to the caller's perspective, the implementation must also solve how to prevent a target object from having uncontrolled side effects on the objects which the caller gives as actual arguments. Fortunately, there is a connection between the output leakage and side effects. If we can construct a safe object reference, e.g. a handle with a recursive read-only permission, we can obviously use the same construction to avoid side effects to the actual arguments. The caller can then form restricted versions of the object references and give only them to the target object, or the target can guarantee to the caller that it makes such conversion automatically when the actual arguments are assigned to the formal arguments.

Traditionally, the leakages are either ignored or managed (a) by copying the referred object and possibly its subobjects to eliminate possible side effects, or (b) by defining access mechanisms to the referred object. The latter handling approach is typically implemented by simple wrapper routines or specialized iterators, which are used to ensure that the client cannot violate any class invariants. As an example about the recent research discussion about this kind of mechanisms, see [2, 13]. To motivate the solution given in Section 5 let us compare shortly what properties the copying, wrapping, and iterating techniques have with respect to aliasing.

**Copying** To handle the input and output leakages the referred object  $r$  can be copied to/from the target object. However, this approach introduces at least the following issues. (i) Copying is expensive in terms of used memory and processing time. (ii) If  $r$  is composite we have to know exactly what objects to copy. Is it enough to copy only  $r$  (i.e. shallow copy) or, for the sake of safety, should we copy all the objects accessible from  $r$  (i.e. deep copy)? How about some semi-deep copy? To make this more complicated, the selection of objects to be copied depends on how they are accessed later. Although copy-on-write data structures address this issue, they rarely solve the next concerns. (iii) Because copying handles objects like values, the identity of  $r$  is not passed between the caller and target objects. (iv) If the copied objects need to be consistent, but it is possible that some information of one copy object can change without breaking the related invariants, the other copies need to be updated. (v) In principle, unless we need snap-shots from objects it makes seldom sense to replicate them in a single thread. (vi) The requirement for copying must be expressed explicitly in the contract of the interface which is passed through. To conclude, copying should be used only as a last resort.

**Wrapper routines** One way to avoid copying is to use code wrapping and to repeat the code: We hide the subobjects from the outside access and recursively expand the relevant routines to the top-level object. We could, for example, hide the keyed element in the `AvlTree` class to (partially) protect the ordering invariant but, consequently, `AvlTree` requires routines `Object GetSatellite()` and `PutSatellite(Object)` to handle the satellite data. However, code wrapping is not always applicable. Firstly, wrapping does not comply with the object-oriented principles of simplicity and modularity. Secondly, the behavior of (unconstrained) type generic subobjects is unknown and, therefore, their detailed data cannot be wrapped. Moreover, if the instance structure of the subobjects is not fixed, we cannot use wrapping<sup>1</sup>. Thus, in our AVL example, it is impossible to protect the balancing invariant by code wrapping alone.

**Iterators** Code wrapping also requires an iterator abstraction [5, 10] which hides the actual structure of the object. Instead of revealing a direct reference to the subobject, the subobject is encapsulated by an iterator object. For example, the fetch function of the AVL (sub)tree should be declared as `AvlTreeIterator Get(Orderable key)`. After that, the tree traversal would be based on four iteration

---

<sup>1</sup>Here we mean *simple* function wrapping. Of course a function can have an argument that is interpreted at runtime and in this way modifies the behavior of the function.

commands [11] in the `AvlTreeIterator`: move cursor up to the parent, down to the  $i$ th child, back to the previous sibling and forth to the next sibling. Unfortunately, multicomponent data structures can require multiple traversal commands, and, therefore, the iterators suit mostly for the recursive structures. The interface to the iterator is static and it often defines only a sequential (more generally “a move to a specific neighbor”) traversal rather than unpredictable walk-throughs. Furthermore, iterators expose only local relationships between subobjects, and they prevent “the client knows the best way” traversing. For example, the `AvlTreeIterator` could support the standard preorder, inorder, and postorder walks, but there is no reason to include every possible, e.g. client specific, traversal order into the iterator interface. Finally, it is also worthy noticing that iterators basically protect the structure of the object, not the contained data elements.

Each of these three leakage management techniques — i.e. copying, wrapper routines, and iterators — provides a single access point (SAP) which protects the integrity of the invariants. However, the (sub)object aliasing as such breaks SAP principle. To restore the possibility for guarding the invariant, we have to express explicitly the software contract between the aliasing customers and providers. It can be stated manually, for example, as a concretization of some suitable design pattern, or it can be generated automatically, if the programming language supports it directly [12]. This paper focuses on the design pattern approach.

## 4 Problem and Its Contexts

Access Protector gives a solution to the following design technical problem.

### Problem statement

*How to guarantee at compilation time that aliasing a subobject does not provide a way to break the integrity invariants of its containing objects?*

This problem presents itself in various forms and, thus, Access Protector can be used, for example, when one

- needs a mechanism that states explicitly how a subobject relates to the state and state changing of its containing object, i.e. when the public aspects of a class invariant must be preserved,
- wants to share object’s implementation and manage the consequent output leakage at the same time,
- wants to avoid side effects to routine arguments,
- is implementing different immutability levels for an object, or
- wants to gain benefit from the stabilized implementation of an object in terms of time and space consumption without breaking the information hiding principle.

The underlying properties of the pattern are discussed in Section 9.

## 5 Solution

The pattern makes possible to increase the granularity of protection control over the object references. The solution regroups the routines into multiple classes or interfaces according to the required access

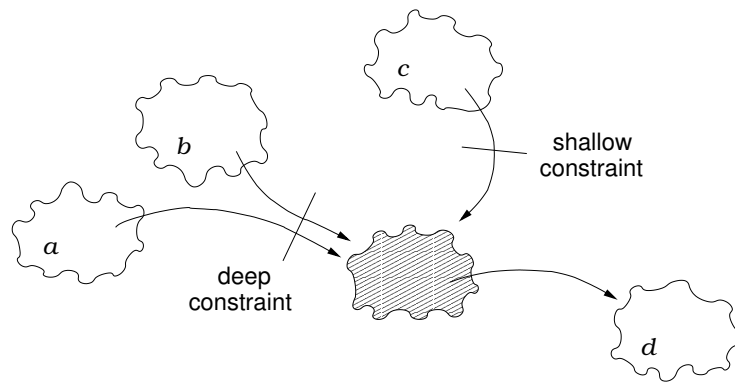


Figure 2: Shallow immutable and deep immutable views to the shaded object.

protection hierarchy. Each access protected class defines a certain *view* (i.e. a role or a constrained reference) to an object, and an object can be referred simultaneously via several kind of views. Although the protection hierarchy can be as general as a directed acyclic graph, usually much simpler structures, such as chains, have sufficient expression power. Thus, in what follows, our focus is on the well-known chain of access protections called “mutable—shallow immutable—deep immutable”. To use more general terminology, one can always interpret ‘mutable’ as ‘the least restrictive view’ and ‘deep immutable’ as ‘the most restrictive view’. The term ‘shallow immutable’ (and ‘partial immutability’) represents all of the intermediate views between these two extremes.

A target object  $t$  is *mutable* to a caller object  $c$ , if  $c$  can alter the attributes of  $t$ . The integrity of the mutable object is defined by class invariants together with pre- and postconditions of the routines. On the other end, object  $t$  and its subcomponents can be referred recursively as read-only. This type of object protection is called (*deep*) *immutable*, since an immutable view can only give immutable views. In this case, if caller  $c$  accesses target object  $t$  only via immutable views (references),  $c$  cannot break the class invariants of  $t$ . An intermediate object protection, *shallow immutability*, guarantees that the attribute values of  $t$  are preserved. However, this restriction does not cover the objects referred by  $t$ : The subobjects are *free* for change. Thus, only the surface of  $t$  is protected.

In Figure 2, the shaded object is viewed via deep immutability and shallow immutability constraints by objects  $a$ ,  $b$ , and  $c$ . The objects  $a$  and  $b$  cannot change the shaded object nor object  $d$ . Although  $c$  does not alter the shaded one either, it can freely mutate  $d$ . Notice that changes made to object  $d$  can change the behavior of the shaded object and these effects are also seen by the objects  $a$  and  $b$ .

Shallow immutability causes that the (public) subobjects must not have value semantical effects on the invariants of the containing object. Therefore, the object acts as a container, and its main purpose is to record the existence of the subobjects. For example, because  $c$  views the shaded object via a shallow immutable reference in Figure 2, the state of (publicly accessible)  $d$  cannot be constrained by the shaded object. Notice that in general the shaded object defines which subobjects it allows to be accessed via shallow immutable, deep immutable, and/or mutable views. Since most object-oriented languages support shallow immutable views, we focus here on other kind of *partial immutability*, and deep immutability.

Partial immutability of an object  $t$  can be seen as a constraint over the transitive object closure of  $t$ . In other words, let us consider the objects that are reachable via a reference to  $t$ : A partial immutable view defines *certain* attributes in *certain* objects to be immutable. In practice, when one accesses a subobject it means the original partial immutable view of the containing object is projected to the subobject and this yields a new kind of partial immutable view, i.e. a type, for the subobject. Thus, when a partial

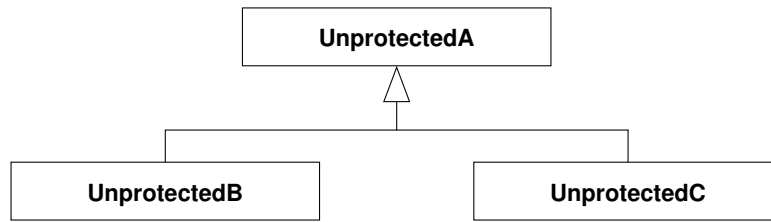


Figure 3: Unprotected class hierarchy.

immutable view is realized with a class system it induces a set of classes that represent all the needed protective views.

## 6 Structure

Suppose that we want to have three access-protection layers: deep immutable, partially immutable and mutable. Access Protector can result in very complex class diagrams, and, therefore, we present first how to construct the plain inheritance hierarchy. After that, we show the positioning of routines and attributes of one class to clarify the changes in the client/supplier relationships.

**Inheritance relationships** Initially, we have a hierarchy of unprotected classes shown in Figure 3. Figure 4 illustrates the most complex access protected inheritance hierarchy that can be constructed from it. The partially immutable part of the class hierarchy tends to have the most complex structure, but in practice it also contains unnecessary inheritance relationships that can be removed.

The structure of the inheritance hierarchy can be imagined to form a two-dimensional lattice with at least three access-protection layers. Each layer represents one protection variant of the original hierarchy, and the layers are connected by inheritance between the corresponding classes, as in Figure 4. The top layer consists of the deep immutable version of the original hierarchy, the middle layer has the partially immutable classes, and the bottom layer contains the mutable classes. Thus, the complexity of a completely access-protected class hierarchy is manageable, since there are two orthogonal concepts: the original hierarchy and the protection hierarchy. To keep the semantics of Access Protector clear, the original inheritance relationships (and their replications) describe the intended polymorphism. Moreover, the inheritance relationships of the protection hierarchy are used to define the strengthening of the access-protection levels. In this respect, it is natural to accept that dynamic downcasts are not allowed in the protection hierarchy.

Access Protector has the following participants with respect to the inheritance relationships:

- *DeepImmutable classes* (i) define an interface (view) for unchangeable objects, (ii) contain functions that return only copied objects or deep immutable references, and (iii) hide the attributes and internal representation by wrapping.
- *PartiallyImmutable classes* (i) allow changes to the free subobjects, and (ii) include functions returning constrained views to subobjects, i.e. returning references which reflect the integrity rules that protect the whole object structure.
- *Mutable classes* (i) allow the object to change its state, and (ii) contain the procedures.

**Client/supplier relationships** To recall, a function is a routine (returning a value) that does not change the behavior of the object, i.e. it can have only benevolent side effects. On the other hand, a

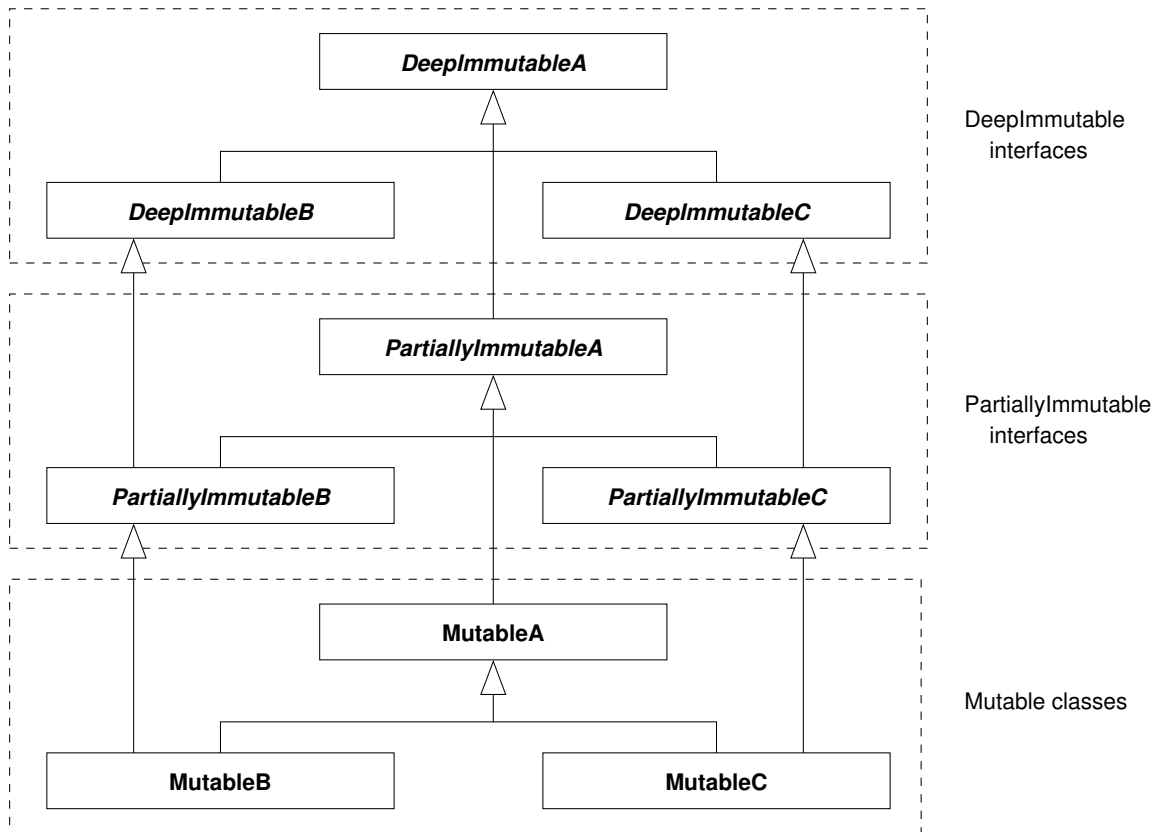


Figure 4: Completely access-protected class hierarchy with three layers of subhierarchies for immutability.

procedure is based on side effects and does not return anything.

Suppose we have initially class Unprotected, shown in Figure 5, with intermixed functions and procedures. The class includes function FixedSubpart() which returns a reference into a subobject that participates in the class invariant. Conversely, the referred object returned by FreeSubpart() does not have any effects on the invariants. The class also comprises a Procedure() and a query function for each attribute. This class can be divided into three access-protection roles, as illustrated in Figure 6. Each protection role belongs to its own hierarchy layer in Figure 4.

Access Protector has the following participants with respect to the client/supplier relationships:

- *DeepImmutable classes* (i) redefine attributes to be non-public in order to prevent outside assignments, (ii) wrap each originally public attribute with a function returning a deep immutable reference, and (iii) redefine all the other routines to be functions returning deep immutable references.
- *PartiallyImmutable classes* can redefine functions of DeepImmutable to return covariant references to objects having no effect on the class invariant. These functions must be *free* with respect to the object integrity. In other words, the return type of a function, e.g. FreeSubpart(), must project the constraints which define the partiality the class models.
- *Mutable classes* contain all the procedures, since those change the state of the object.

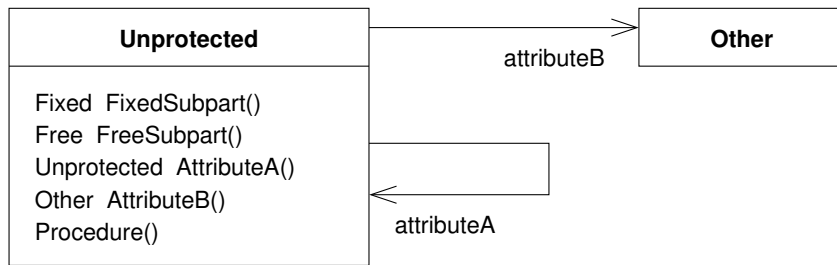


Figure 5: An unprotected class.

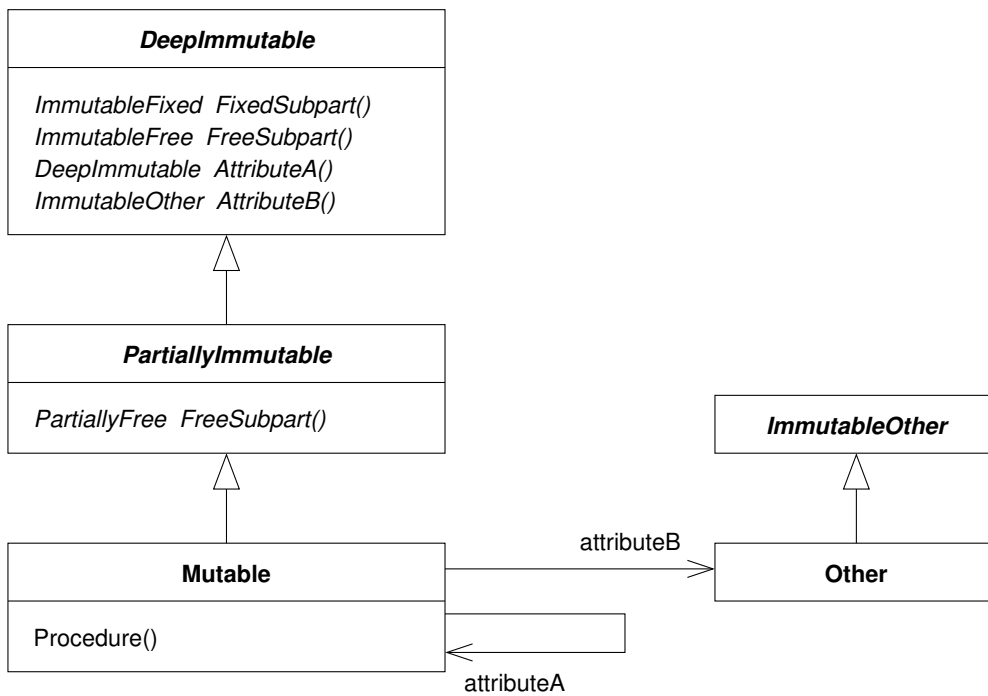


Figure 6: Three access-protection layers of Unprotected.

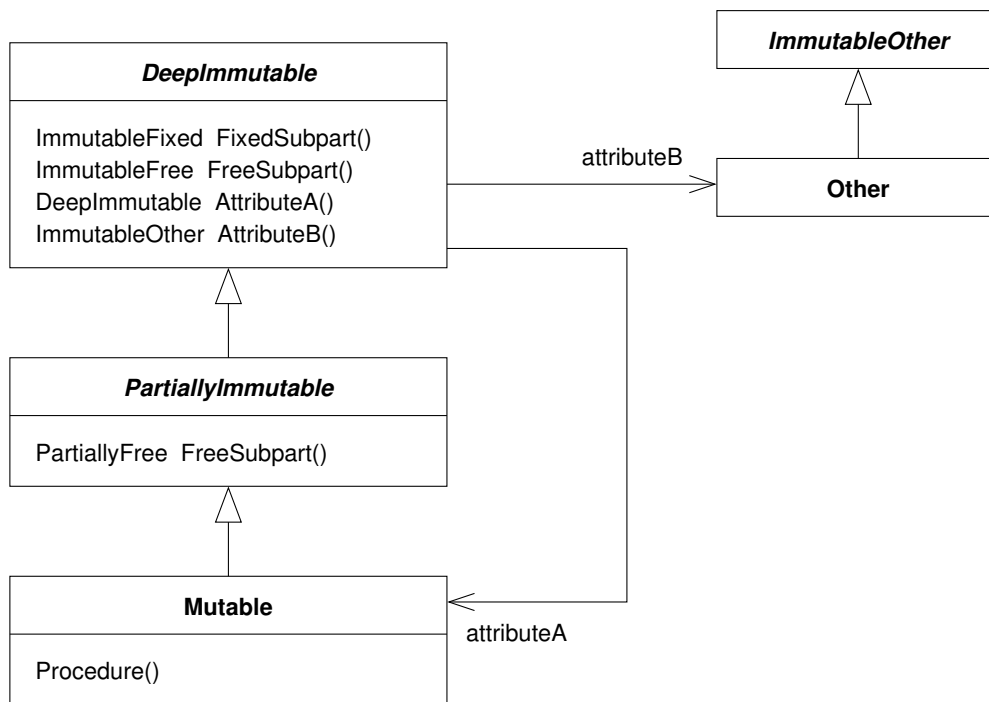


Figure 7: Implementation without interfaces.

## 7 Implementation variations

Although melting the access-protection layers with the original class hierarchy is, in most cases, quite a mechanical process, it can bloat the hierarchy so that it becomes difficult to understand. Ideally, object-oriented programming languages should support the implementation of Access Protector by aiding the definition of the (partly) immutable interfaces and the distribution of the routines into them. Fortunately, there are many ways to implement Access Protector also “manually” without native language support.

Figure 6 depicts a direct implementation by interfaces `DeepImmutable`, `PartiallyImmutable`, and `ImmutableOther`, and by classes `Mutable` and `Other`. However, there is no way to prevent downcasting the protection away on the client code.

Another simple basic implementation, shown in Figure 7, is based on the idea that all data is implemented in the abstract class `DeepImmutable`. This approach is also straightforward to code so that the hidden features<sup>2</sup> are published at the appropriate subclasses.

To prevent downcasts we can utilize private inner classes to produce runtime objects with hidden types. As an example, Figure 8 shows the implementation of a class representing 2D planar point and an immutable version of it as a private inner class. Interface `ImmutablePlanarPoint` could be replaced with an abstract class. The intermediate level (partially immutable) is not present, but it could be implemented as another inner class.

<sup>2</sup>In C++ and Java they can be implemented as protected member functions.

```

public interface ImmutablePlanarPoint {
    public int getX();
    public int getY();
} // ImmutablePlanarPoint

public class PlanarPoint implements ImmutablePlanarPoint {
    private int x;
    private int y;

    public PlanarPoint() { }

    public int getX() { return x; }
    public int getY() { return y; }

    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }

    public ImmutablePlanarPoint getView() { return new PlanarPointView(); }

    private class PlanarPointView implements ImmutablePlanarPoint {
        // No instance variables.
        public PlanarPointView() { }

        public int getX() { return PlanarPoint.this.getX(); }
        public int getY() { return PlanarPoint.this.getY(); }
    } // PlanarPointView
} // PlanarPoint

```

Figure 8: An immutable planar point with a private inner class in Java language.

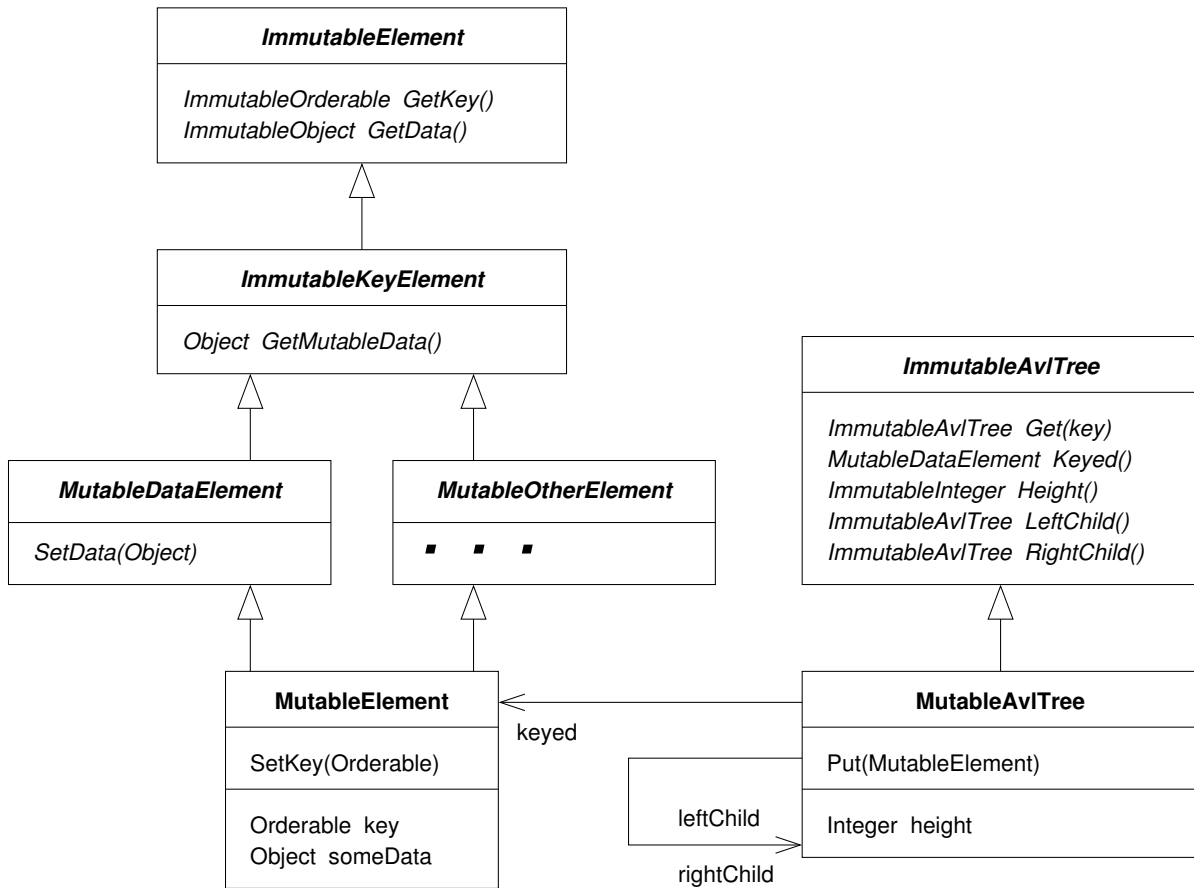


Figure 9: The AVL tree with protective access views.

## 8 Example Resolved

Let us revert to the AVL example (see Figure 1) and demonstrate how to apply Access Protector pattern in stages. The immutable interface must ensure that the state of the objects does not change and thereby both the ordering and balancing invariants cannot be violated. Thus, we define two classes, `ImmutableAviTree` and `MutableAviTree` so that `MutableAviTree` extends `ImmutableAviTree` by inheritance. Class `ImmutableAviTree` is deferred, i.e. interface, or abstract class, since it defines a restrictive view rather than an object as such. We must emphasize that inheritance is used here primarily as a tool for code regrouping (or reuse) and it does not define a true behavioral “is-a” relationship.

The idea is to group the query routines, i.e. function `Get(key)`, into class `ImmutableAviTree` and the others, i.e. procedure `Put(key)`, into `MutableAviTree`. Furthermore, to prevent outside assignments, `keyed`, `leftChild`, `rightChild` and `height` are defined as non-public attributes in `MutableAviTree`. These attributes can be accessed only through functions `Keyed()`, `LeftChild()`, `RightChild()` and `Height()`. Because the balancing invariant of the AVL tree should not be violated by a client, these functions return references to the objects of type `ImmutableAviTree` (see Figure 9).

On the extreme, function `Keyed()` should return an immutable reference, since the ordering of the AVL tree relies on the search tree property of the key elements. Therefore, to have a proper distinction between the roles of `Keyed()` and `keyed`, class `Element` must be divided into classes `ImmutableElement` and `MutableElement`. This can be seen as a safe (i.e. “zero-knowledge”) approach, because we do not

have to know how the client uses the Element objects: The client is forced to make a deep copy, if it is not satisfied with the transitive read-only restriction. However, it is possible to refine the access hierarchy on Element without violating the invariants of the AVL objects by taking the client's context into account followingly. The AVL invariants concern only attribute key of Element, and we call these kind of attributes *fixed*. On the other hand, the attribute *someData* is *free* because the AVL tree sees it only as a placeholder that has no restrictions. Thus, as far as the AVL definition is concerned, both reference *someData* and its target object of type Object can change. If we want to separate the two cases, we can define classes `ImmutableKeyElement` and `MutableDataElement`<sup>3</sup>. Furthermore, especially in the case of aggregate Element attributes, we can also have other similar mutating operations (class `MutableOtherElement`<sup>4</sup>), but let us content that it is necessary to embed only the preceding client contexts. Now, it is possible to have the least restrictive view for the return value of the AVL function `Keyed()` — `MutableDataElement`.

The solution of the class diagram illustrated in Figure 9 can be further refined by preventing the possible side effects in the argument passing. Instead of `ImmutableAvlTree Get(key)`, one should require the interface to be `ImmutableAvlTree Get(ImmutableElement key)`. Essentially, the same consideration applies also to `Put(key)`, but we omit the fine details here.

Figure 9 defines the static structure of the Access Protector. The dynamic behavior of the pattern is depicted in the object diagram in Figure 10. Note that if Access Protector is realized using interfaces, the original object structure does not change. This means that the dynamic behavior remains the same. As reflected also by the class diagram, there are four fundamental usage contexts, i.e. roles, for the AVL tree objects.

- *Tree owner* Determining and coordinating object ownerships is one of the corner stones in object-oriented software construction. The owner is responsible for preserving the integrity of the owned objects in the system. Thus, it must have a mutable view on the properties that are owned. In the case of our AVL tree, the tree owner manages the whole AVL structure via the root node.
- *Tree browser* Because query operations do not change objects, the owner of an object can share any part of the structure for browsing purposes. In the AVL example, the node implementation can be aliased via the interface `ImmutableAvlTree`. This access can be granted either by the tree owner or another browser.
- *Data assigner* The AVL tree is a container of objects having key and data attributes. The mapping between an existing key and some satellite data does not affect the AVL tree. Thus, the data can be assigned independently. Of course, the tree owner can also be the data assigner, but is not necessary.
- *Data updater* The contents of free data can be updated without violating the invariant conditions. Compared to data assigners, the need for data updaters is in practice much higher. For example, in a bank account system updating the current address of the account holder would be a typical task for a data updater.

---

<sup>3</sup>The `ImmutableKeyElement` is actually partially immutable restriction with respect to *someData*. This demonstrates that the partial immutability should be seen as a property of an attribute rather than property of an object having it.

<sup>4</sup>Note that Java does not support directly this kind of multiple inheritance of classes whereas Eiffel and C++ do.

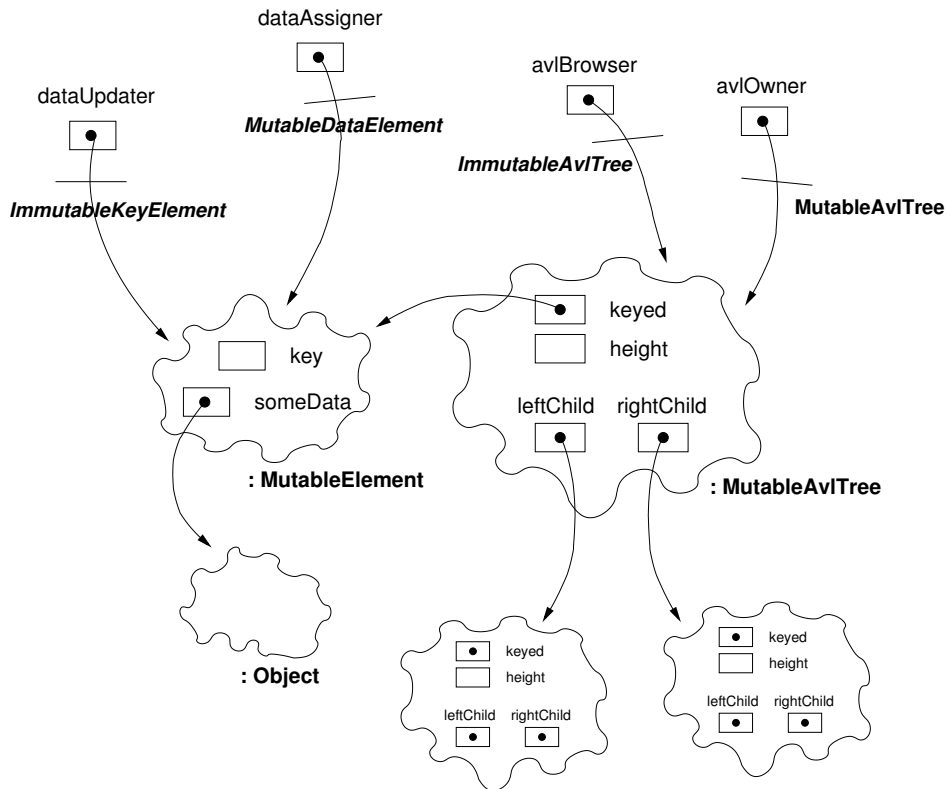


Figure 10: An object structure of the AVL tree.

## 9 Consequences

Access Protector pattern has the following *benefits*:

- *Explicit access protocols to objects* Instead of describing the access responsibilities between the clients and suppliers of an object just by documentation means, the pattern allows us to define, name, and test these contracts by the type system at compilation time.
- *Subtle hierarchy of the access levels* The access privileges can be defined as a directed acyclic graph by using the inheritance relationships. However, it is often sufficient to have a chain-like organization of privileges.
- *Access-protection views are transitive* The access level is a property of an object reference, not part of the object itself. This allows to apply the same (or stricter) access policy also to subobjects.
- *Access constraints cannot be loosened* Due to the way the pattern utilizes inheritance mechanism, polymorphism cannot be used to break the protection semantics: Access restrictions strengthen by (implicit) upcasting. To prevent explicit downcasting, hidden inner classes can be used.
- *Ability to distinguish immutability and mutability* One of the simplest applications of the pattern is to declare immutability and mutability accesses for an object. These views make it possible to avoid unnecessary copying of objects which are passed to/from a routine.
- *Controlled and safe aliasing of subobjects* The pattern introduces a way to define mechanisms that can be used to share object's subobjects, i.e. the internal representation, without compromising

the integrity of the object. In fact, the pattern emphasizes that *information hiding* and *encapsulation* are logical concepts that do not require that the internal implementation of an object must always be isolated from the object's use context. At the same time, because the accesses are defined through interfaces, the underlying implementation can still be changed<sup>5</sup>. Simply put, why not to take advantage from the efficient subobject implementations which, in addition to being stabilized, also conform to the intended public behavior, e.g. to object's (abstract) attributes?

The *liabilities* of Access Protector pattern include:

- *Large number of interfaces and classes with complex relationships* When the pattern is applied, the inheritance hierarchy under refinement is replicated (see Figure 4) and the member routines are redistributed (see Figure 6). This combination can yield a complex class hierarchy and, thus, if the pattern is used “manually” the scope of pattern application should be small. Due to this problem, we have developed an experimental version of Java 5.0 with an extended type system for expressing access protection so that the inheritance hierarchy replication becomes virtual [12].
- *Most of the routines are dynamically bound* Access Protector is mainly build on non-concrete classes that inherently means dynamic binding of routines. Especially in real-time systems, the runtime overhead can become unmanageable. As the preceding concern suggests, the pattern should be applied only to the most *definition critical* part of the system.
- *Selection of the intermediate layers* The top and the bottom access-protection layers are often the most easiest to determine: Typically, the top layer allows only observations and the bottom layer has the initial properties of the system before using the pattern. The more detailed characteristics of the object state and its changing are reflected in the intermediate layers. This means that the pattern suits only for situations where the object's integrity definitions are stable.

## 10 Known Uses

The underlying ideas behind the pattern have been publicly recognized at least a decade ago, as far as the authors know. The recent software industrial examples can be seen in the API specification for the Java 2 Platform Standard Edition 5.0 [16] by Sun Microsystems, Inc.

- Package `javax.swing.text`: Interface `AttributeSet` defines an immutable view for a collection of unique attributes and it is extended by interface `MutableAttributeSet` to allow changes. Class `SimpleAttributeSet` gives the concrete implementation to these properties.
- Package `java.util.regex`: Interface `MatchResult` defines query methods to access the results of a match against a regular expression. It is implemented by class `Matcher` which also defines the actual matching engine for character sequences. However, the matching cannot be triggered via the `MatchResult`.
- Package `java.util`: Class `Collections` defines function `unmodifiableCollection(c)` that wraps the collection data structure `c` so that the procedure methods cause exception. This can be seen as a way to implement Access Protector with a wrapper object (although it breaks the Liskov substitution principle).

---

<sup>5</sup>In fact, Access Protector could be described more generally with subinterfaces instead of subobjects. However, we find the pattern more approachable in the concrete context.

From the academic research perspective, the Access Protector pattern can be used to control the effects of object aliasing. The most recent research focusing to this topic includes: *Flexible alias protection* (FAP) [3, 14], *JAC & transitive read-only* [8, 9, 15, 12], *deeply immutable* and *shadow immutable views* [7], and *ownership types* [2, 4]. The nature of all these works is to extend an object-oriented language with some constructions that can be used to provide compilation time support. There exists also research on solutions providing only runtime mechanisms.

## 11 See Also

Access Protector is the opposite of the Adapter pattern [5]. The Proxy pattern could be used to implement constrained views. The Immutable design pattern [6] can be seen as a special case of the Access Protector. Whereas the Flyweight design pattern [5] focuses on sharing, the Access Protector focuses on flexible safe sharing (partial immutability). See also the comments [9] about the relationship to Flyweight pattern.

## Acknowledgments

We wish to thank our EuroPLOP shepherd Arno Schmidmeier for his valuable advice and comments. Also, we want to express our gratitude to Jouni Smed for his support.

## References

- [1] B. Bokowski and J. Vitek. Confined Types. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'99*, ACM SIGPLAN Notices, 34(10):82–96, 1999.
- [2] C. Boyapati, B. Liskov, and L. Shriram. Ownership Types for Object Encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'03*, pages 213–223, 2003.
- [3] D. Clarke, J. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'98*, ACM SIGPLAN Notices, 33(10):48–64, 1998.
- [4] D. Clarke, J. Potter, and J. Noble. Simple Ownership Types for Object Containment. In *Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP'01*, LNCS 2072, pages 53–76, 2001.
- [5] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [6] M. Grand. *Patterns in Java, Volume 1*, Wiley & Sons, 1998.
- [7] H. Hakonen, V. Leppänen, and T. Salakoski. Object Integrity while Allowing Aliasing. In *Proceedings of The 16th IFIP World Computer Congress International Conference on Software: Theory and Practice, ICS'2000*, pages 91–96, 2000.

- [8] G. Kniesel, D. Theisen. JAC – Java with Transitive Readonly Access Control. In *Intercontinental Workshop on Aliasing in Object-Oriented Systems, IWAOS'99*, 1999.
- [9] G. Kniesel, D. Theisen. JAC – Access Right Based Encapsulation for Java. *Software – Practice & Experience*, 31(6):555–576, 2001.
- [10] B. Liskov, J. Guttag. *Abstraction and Specification in Program Development*, MIT Press, 1986.
- [11] B. Meyer. *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, 1997.
- [12] S. Mäkelä, V. Leppänen. Safe Aliasing for Java via a Type System Extension. 20 pages, 2006, manuscript, submitted for publication.
- [13] J. Noble. Iterators and Encapsulation. In *Proceedings of Technology of Object-Oriented Languages and Systems, TOOLS 33*, pages 431–442, 2000.
- [14] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *Proceedings of the 12th European Conference on Object-Oriented Programming, ECOOP'98*, LNCS 1445, pages 158–185, 1998.
- [15] M. Skoglund and T. Wrigstad. Alias Control with Read-Only References. In *Proceedings of 6th International Conference on Computer Science and Informatics, CSI*, pages 457–472, 2002.
- [16] Sun Microsystems. Java 2 Platform Standard Edition 5.0 API Specification. Web page, available at (<http://java.sun.com/j2se/1.5.0/docs/api/index.html>), May 2006.