Object-Oriented Design Pattern: 
Access Protector

Harri Hakonen and Ville Leppänen
Department of Information Technology, University of Turku, 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.

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

1. Introduction

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. The implementation can be made so that the granted permissions cannot be loosened when the object reference is forwarded. Also, the object can be referred via multiple views of different access level at the same time depending on what kind of object integrity protection is required toward different clients.

Access Protector is a compile time solution (all checking is reduced to type checking) and it is intended for designing a safe public interface for a class library. The pattern can also be used, for example, in the following situations.

- For efficiency reasons the internal structure of an object must be shared with the client objects. For this case we can define what are considered as illicit changes to the aliased objects and to disallow them. Because the pattern introduces interfaces the aliasing does not break the information hiding principle.
- We want to prevent side-effects to a routine argument comprising a structure of objects. In this case the pattern can be used for implementing different immutability policies for the subobjects. In other words, we do not need deep cloning of objects to guarantee immutability of passed arguments.

2. Example

Context

Consider an implementation of an AVL tree, which is a balanced binary search tree with two invariant conditions: the ordinary key ordering condition and with a balancing condition (the height of the left and right subtree differ at most by one). Figure 1 illustrates a recursive implementation of the AVL tree to also support hierarchical access 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) (which is a pair: a totally orderable key (key) and a satellite data (someData)) and two children (leftChild and rightChild) which are AVL trees.

Note that the recursive definition of the AVL tree unifies each (sub)node to its corresponding AVL (sub)tree.

Problem

The AVL tree implicitly assumes that the client accessing a subtree s of a tree t (e.g. via s = t.LeftChild()) does not mutate the element keys nor the node ordering/balancing in the subtree s, because changes can violate invariant conditions of t. In other words, the definition of the AvlTree class omits the issue of object sharing (or aliasing) with different access rights simultaneously to the same object and, at most, only documents the usage assumptions.

Supporting such different views should be efficient – preferably all the checking should happen at compile time.

Solution

A solution is to support simultaneously two kinds of views on subtrees: (a) with full access rights (internally) and (b) partially immutable views (for external clients).

By a partially immutable interface we ensure that the state of subtrees cannot be changed and thereby both the ordering and balancing invariants cannot be violated. We only allow changing the satellite date (someData) of node elements. Thus, we define two classes, PartiallyImmutableAvlTree and MutableAvlTree so that MutableAvlTree extends PartiallyImmutableAvlTree by inheritance. The MutableAvlTree corre-
The idea is to group the query routines, i.e. function \texttt{Get(key)}, into class \texttt{PartiallyImmutableAvlTree} and the others, i.e. procedure \texttt{Put()}, into \texttt{MutableAvlTree}. Furthermore, to prevent outside assignments, \texttt{keyed}, \texttt{leftChild}, \texttt{rightChild} and \texttt{height} are defined as non-public attributes in \texttt{MutableAvlTree}. These attributes can be accessed only through functions \texttt{Keyed()}, \texttt{LeftChild()}, \texttt{RightChild()} and \texttt{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 \texttt{PartiallyImmutableAvlTree} (see Figure 2).

For the element we also need a partially immutable view – besides the full view – that denies changes of the fixed key value but allows changing the (free) satellite data. The class \texttt{Element} is divided into classes \texttt{PartiallyImmutableElement} and \texttt{MutableElement} as shown in Figure 2.

The solution can be further refined by preventing the possible side effects in the argument passing. Instead of \texttt{PartiallyImmutableAvlTree Get(key)}, one should require the interface to be \texttt{PartiallyImmutableAvlTree Get(ImmutableOrderable key)}. Essentially, the same consideration applies also to \texttt{Put()}. Figure 2 defines the static structure of the Access Protector for the AVL tree example. 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.

3. Access Protector pattern

\textbf{Context}

In designing the public interface of a class library, our pattern enables an alias reference to a subobject from an object context that restricts the state values of the subobject by some invariant conditions.

\textbf{Problem statement}

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

\textbf{Forces}

- Aiming at higher software reliability by guaranteeing that a client using a subobject cannot violate invariant conditions of the containing object.
- The solution should be efficient – requiring little or no runtime overhead.
- The resulting design of classes should be understandable, maintainable, and fast to create.

\textbf{Solution}

The solution regroups the routines into multiple classes or interfaces according to the required access protection hierarchy. The access protection hierarchy is obtained by considering all the (invariant) constrained contexts, where objects of given type are used. 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. The most restrictive view is a deep immutable view, whereas the most allowing is the mutable view. Between the two extremes can be a set of partially immutable views.

A partial immutable view 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 – whereas the others are free for changes. 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 immutable view is realized with a class system it induces a set of classes that represent all the needed protective views.

**Solution structure**

Suppose we have initially class Unprotected, shown in Figure 3, 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, for example, into three access-protection roles, as illustrated in Figure 4.

![Solution structure](image)

**Figure 3: An unprotected class.**

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

- **DeepImmutable classes/interfaces** (i) are the extreme in protection providing a common base for various access levels. Those classes 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

![DeepImmutable](image)

**Figure 4: Three access-protection layers of Unprotected.**
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.

4. 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 4 depicts a direct implementation by interfaces DeepImmutable, PartlyImmutable, and ImmutableOther, and by classes Mutable and Other. However, in this implementation there is no way to prevent downcasting the protection away on the client code.

To prevent downcasts we can utilize private inner classes to produce runtime objects with hidden types. As an example, Figure 5 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.

5. Discussion

5.1 Other ways to approach the problem

Traditionally, the aliasing related issues 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 these kinds of mechanisms, see [2], [13]. To motivate the solution given in Section 3 let us compare shortly what properties the copying and wrapping techniques have with respect to aliasing.

• **Copying:** To handle the aliasing problems the referred object \( r \) can be copied to/from the target object. The advantage of copying is that it prevents aliasing, but it has several severe drawbacks: (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 (shallow copy / deep copy / something else). (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. 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. This has the advantage of removing the need to have reference to subobjects. It also have several drawbacks: 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. Thus, in our AVL example, it is impossible to protect the

```java
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

class PlanarPoint {
    // PlanarPoint
    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 PlanarPointView getView() { return new PlanarPointView(); }
}
```

Figure 5: An immutable planar point with a private inner class in Java language.
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 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 traversal order into the iterator interface. 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.

Both techniques provide 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 the 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.

### 5.2 Inheritance relationships

Initially, we have a hierarchy of unprotected classes shown in Figure 6. Figure 7 illustrates the schema of 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 7. 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.

### 6. Consequences

Access Protector pattern has the following benefits:

- **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 usage context. At the same time, because the accesses are defined through interfaces, the underlying implementation can still be changed1. Simply put, why not 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?

- **Efficient and 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.

- **Granulated 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.

---

1In fact, Access Protector could be described more generally with subinterfaces instead of subobjects. However, we find the pattern more approachable in the concrete context.
• **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.

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 and the member routines are redistributed (see Figure 4). 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 built on non-concrete classes which 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.

• **All implementations do not prevent casting away the access protection** Using only interfaces to define the access levels is light, but does not prevent downcasting. Implementations based on private inner class prevent downcasts but are more expensive in terms of design and runtime costs, and are programming language dependent.

• **Selection of the intermediate layers** The top and the bottom access-protection layers are often the 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.

7. Known Uses

The underlaying ideas behind Access Protector 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’s substitution principle).

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 the 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.

8. See Also

Access Protector is the opposite of the Adapter pattern [5]. The Proxy pattern [5] 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 Jouni Smed and Timo Knuutila for their valuable comments and support.

References