Taking Information Hiding Seriously in an Object Oriented Context

J. Leslie Keedy, Gisela Menger and Christian Heinlein

Department of Computer Structures
University of Ulm
D-89069 Ulm
Federal Republic of Germany

{keedy, menger, heinlein}@informatik.uni-ulm.de

Abstract. Although information hiding is widely recognised as a key strategy for well engineered software systems, its use is not encouraged by standard object oriented programming languages. The paper explores a fundamental aspect of this issue, namely the idea that a programmer should be free to implement a module in any way which fulfils its specification. We show how taking this freedom seriously creates problems for the conventional object oriented dot notation, and we present a solution which permits client programmers to continue to use this notation without restricting the freedom of implementors.

1 Introduction

At the heart of the information hiding principle [8] is the idea that a module specification should not include "raw" data structures as part of the module interface, since these are difficult to specify and they are often subject to change (e.g. to allow optimisations to be introduced, or to reflect modifications in a system's requirements). Instead interfaces between modules should, according to the information hiding principle, be expressed exclusively procedurally, i.e. in OO terms as methods.
In contrast, it is widely accepted OO practice to allow data fields to be included in the public members of a class. Probably the main reason for this approach is that completely procedural interfaces can be both inconvenient and inefficient, especially when defining small object classes corresponding for example to records in Pascal or structures in C, or indeed to records in database systems.
One way of attempting to bridge this gap between information hiding and the OO paradigm is to view a public data field as being equivalent to a pair of methods, one which sets the value of the field, the other which returns its value. However, this widely known technique is by no means as simple as might be assumed, especially if the basic idea behind the information hiding principle is taken seriously, i.e. that it provides an implementation programmer with complete freedom to implement a module as he chooses, provided that the implementation fulfils the specification.

In this paper we show what difficulties can occur and we present the Timor solution for these. Timor is a new programming language currently under development at the University of Ulm [1-4][1].
Section 2 outlines the idea of set and get methods and section 3 presents a standard implementation for them. The client programmer's view of a module which uses set and get methods is discussed in section 4, while section 5 considers the relevance of differences in object semantics between languages such as Java (where objects are always implemented by hidden pointers) and C++ (which distinguishes between pointers and values). Section 6 presents a non-standard implementation of a trivial type which shows that similar problems arise with both forms of semantics. Then in section 7 the basic Timor solution, based on the idea of *abstract variables* at the type level, is presented. Section 8 then discusses the issue of non-standard implementations of abstract variables. Section 9 discusses how this affects the usual interpretation of the dot notation. Finally related work and concluding remarks are provided in sections 10 and 11 respectively.

2 Set and Get Methods

In the context of the idea of set and get methods, a Java field declared as

```
public int aValue;
```

can be considered as (approximately) equivalent to the method pair

```
int aValue(int aValue); // the set method
int aValue(); // the get method
```

In simple cases this creates no problems, but if the information hiding principle is taken seriously, it soon becomes evident that this alone does not completely solve the problem, especially when a class includes a public variable which is not a basic type.
Throughout this paper we illustrate the problems which arise in terms of a class `Person`, in which a public field `Date dob`, defining the `Person`'s date of birth, appears. The `Person` definition includes:

```
class PersonWithVariables {
 public String name;
 public Date dob;
 ...
}
```

and the `Date` definition includes

```
class DateWithVariables {
 public int day, month, year;
 public void reset() {...}
}
```

Reformulating these to replace concrete data fields as method pairs, and using an interface as a type definition (which might have different implementations), we have:

```
interface Person {
 String name(String name);
 String name();
 Date dob(Date dob);
 Date dob();
 ...
}
```

---

1 see http://www.timor-programming.org

```
interface Date {
 int day(int day);
 int day();
 int month(int month);
 int month();
 int year(int year);
 int year();
 void reset();
}
```

which seem quite harmless. We now consider why this is not the case. We have deliberately omitted code details so that in the following sections the above can serve as type definitions which might have different implementations.

3 A Standard Implementation for Set and Get Methods

We begin with an obvious implementation of the set and get methods. In Java an obvious implementation of `Person` is:

```
class PersonImpl implements Person {
 PersonImpl(...) {...}
 private String name;
 public String name(String name) {return this.name = name;}
 public String name() {return this.name;}
 private Date dob;
 public Date dob(Date dob) {return this.dob = dob;}
 public Date dob() {return this.dob;}
 ...
}
```

and of `Date` is:

```
class DateImpl implements Date {
 DateImpl(int day, int month, int year) {
  this.day = day; this.month = month; this.year = year;
 }
 private int day;
 public int day(int day) {return this.day = day;}
 public int day() {return this.day;}
 private int month;
 public int month(int month) {return this.month = month;}
 public int month() {return this.month;}
 private int year;
 public int year(int year) {return this.year = year;}
 public int year() {return this.year;}
 public void reset() {
  this.day = 0; this.month = 0; this.year = 0;
 }
}
```

We subsequently refer to this style of implementation for the set and get methods, consisting of a private variable and method implementations which modify or return the value of the variable, as a *standard implementation*.

4 The Client View

Does this technique work as a client would like it to? Ideally he would like to use the procedurally defined versions of the classes `Person` and `Date` as if they were identical

to the original versions defined to contain public variables. The following trivial piece of code shows that this is not the case.

```
Person p = new PersonImpl(...);
Date d1 = new DateImpl(1,1,2000);
p.dob = d1;
```

Because `dob` has become a private field, it cannot be directly modified. Instead the set method has to be used, i.e.

```
p.dob(d1);
```

A similar case arises when attempting to read the `dob` field. Instead of writing

```
d1 = p.dob;
```

the client writes:

```
d1 = p.dob();
```

This may appear to be syntactically less attractive, and we shall later consider how the situation can be improved with some syntactic sugar. But at this stage we pursue the semantic issues further.

The type `Date` has its own methods, which the client may wish to invoke. If it were a public field he would use the dot notation, e.g.

```
p.dob.reset();
```

But because `dob` has become a private field, it cannot be directly used by a client in a dot expression. Consequently the client would have to rephrase the last statement as:

```
p.dob().reset();
```

In this case he first "gets" the field and then invokes the required method. With the concrete fields available as public members the client could also write, e.g.:

```
p.dob.day = 3;
```

i.e. he could directly set the `day` component of `dob`. The same effect can be achieved, using the standard implementations, by two conversions to methods, i.e.:

```
p.dob().day(3);
```

It thus appears that all is well, provided that the client is prepared to use a clumsier syntax and that the implementor is prepared to write some tedious implementations. However, appearances can be deceiving.

5 Reference and Value Semantics for Objects

Java is an example of a language which uses reference semantics, in the sense that it eliminates explicit pointers, but each variable which is defined to have a structured type is implemented by a hidden pointer. However, not all OO languages have reference semantics. What would the above example mean in the context of a language which supports value semantics for objects, such as C++? Consider a standard implementation which follows the same pattern, i.e. a concrete variable and two methods which simply set and get this variable, e.g. in an implementation of class `Person`:

```
private:
  Date _dob;
public:
  Date dob(Date dob) {return _dob = dob;}
  Date dob() {return _dob;}
```

In this case there is an immediate problem with nested access using the dot notation, because an invocation of the `dob` get method returns a *copy* of the private variable `dob`. Hence an invocation of a method such as

```
p.dob().reset();
```
resets a copy of the variable, but not the actual variable which holds `p`'s date of birth. To avoid this problem the get method would have to return a reference to the internal variable. An equivalent modification would also be necessary for the set method. The changes are not particularly significant. However, this point leads us to a further, more significant issue.

The standard implementations (whether for Java or in modified form for C++) work because they include an internal variable that corresponds to the original public variable. But the idea is only useful - in the spirit of information hiding - if an implementor can provide non-standard implementations. We now consider such an example.

6 A Non-Standard Implementation of Person

If the information hiding principle is taken seriously an implementor should have complete freedom to implement a type in any way that he sees fit, provided that his implementation fulfils the specification. This of course implies that he should be able to produce an implementation which does not contain a private variable corresponding to each set and get pair of methods. Let us take another look at the type definition for `Person`:

```
interface Person {
 ...
 Date dob(Date dob);
 Date dob();
 ...
}
```

Suppose that the implementor decides on an implementation in which the `dob` details should be compacted into a single integer (rather than using the three integers of the `Date` standard implementation). The relevant part of his Java implementation might look like this:

```
class PersonImpl2 implements Person {
 ...
 private int dateOfBirth; // holds the compacted date of birth value
 public Date dob(Date dob) {
  year(dob.year()); month(dob.month()); day(dob.day()); return dob;
 }
 public Date dob() {
  return new DateImpl(day(), month(), year());
 }
 // private methods for format conversion
 private void day (int day) { // sets internal day from int format
  dateOfBirth = dateOfBirth - dateOfBirth % 100 + day;
 }
 private int day() { // returns internal day in int format
  return dateOfBirth % 100;
 }
 private void month(int month) {// sets internal month from int format
  dateOfBirth =
   dateOfBirth - dateOfBirth % 10000 + month * 100 + dateOfBirth %100;
  }
 private int month() { // returns internal month in int format
  return (dateOfBirth % 10000 - dateOfBirth % 100) / 100;
 }
 private void year(int year) { // sets internal year from int format
```

```
    dateOfBirth = year * 10000 + dateOfBirth % 10000;
  }
  private int year() { // returns internal year in int format
    return dateOfBirth / 10000;
  }
}
```

This (partial) implementation of `Person` fully accords with the information hiding principle, which is about being able to hide implementation details (especially of data structures) behind a procedural interface. Yet semantically it does not fulfil the expectations of client programmers, as discussed above. The problem is that even in this Java example - despite the use of reference semantics for objects - the client programmer cannot invoke methods of the "correct" `Date` object, because such an object does not even exist.

Attempts to do so, e.g. by method invocations such as

```
p.dob().reset();
```

or

```
p.dob().day(3);
```

are not recognised as errors by the compiler or the run-time system, since these are valid method invocations on the `Date` object returned by the get method `dob`, but semantically this is not the original object which contains the actual date of birth in this implementation of `Person`. Hence resetting the date using the `reset` method or changing the day using the `day` set method of the nested "variable" have no effect on the date of birth as it is stored in the `Person` object.

In Java it is in fact possible to implement a semantically correct non-standard implementation based on the idea that in an invocation such as

```
Date d = p.dob();
```

a proxy object is returned. This holds a back reference to `p` (either implicitly using an inner class or explicitly using a nested static class). Hence when `d.reset()` is invoked, this can execute the required algorithm.

This solution relies on the fact that Java uses reference semantics for all variables. However, it cannot simply be carried over to produce a solution in C++ (or other languages which use value semantics and pointers), because the value returned by a method invocation such as `p.dob()` involves returning a value, not a pointer. Hence the only way to produce a semantically correct implementation in C++ is to include an actual variable of type `Date` in the implementation of `Person` and to pass a reference to this back to clients in the get method of `dob`.

Thus in languages which support value semantics the object oriented paradigm restricts the implementation programmer's freedom to implement a type as he sees fit.

7 Abstract Variables

One of the main aims in the design of Timor [1-4] is to provide a suitable framework for developing fully modular software systems, and in this respect the information hiding principle plays a key role. As a consequence the conventional class construct has been abandoned in Timor and instead a rigorous distinction is drawn between types and their (potentially multiple) implementations, based on a rigorous application of the information hiding principle. Nevertheless the language shares many of the aims of the object oriented paradigm.

Timor also introduces a new programming paradigm which allows an object to be qualified by objects of other types (known as qualifying types [3, 4]). This paradigm makes it essential - to clarify which variables "belong to" an object for purposes of qualification and which are other objects to which it refers - to distinguish between value semantics and reference semantics. Hence, with respect to value semantics, the proxy pattern discussed in the previous section cannot be carried over to Timor, just as it cannot be carried over to C++. In the rest of the paper we describe how Timor tackles the problem set out in the previous sections.

7.1 Abstract Variables in Timor

Timor supports an idea known as "abstract variables". On the one hand abstract variables provide implementation programmers with complete freedom to implement a type in any way they choose, while on the other hand client programmers have the convenience of (apparently) being able to use exported variables, as if they were public fields in the object oriented paradigm.
Abstract variables appear in type definitions, e.g.

```
type Person {
 ...
 Date dob;        // an abstract variable
 ...
}
type Date {
 ...
 int day, month, year; // three abstract variables
 op void reset();      // a normal method
}
```

In an implementation an abstract variable is formally considered to be a set and get method pair. For `dob` in the above example these are defined as follows:

```
final op Date dob(Date dob); // the "set" method
    // an operation (keyword op) can modify the state of an instance
final enq Date dob();        // the "get" method
    // an enquiry (keyword enq) cannot modify the state of an instance
```

If an abstract variable is declared as `final`, it only has a get method.
As indicated above, Timor has value semantics, hence the enquiry `dob` returns a value, rather than a reference.

7.2 Standard Implementations

The standard implementation for an abstract variable consists of a private (concrete) variable and a standard implementation of the method pair, e.g.

```
state:
 Date dob;
instance:
 op Date dob(Date dob) {return this.dob = dob;}
 enq Date dob() {return this.dob;}
```

Such code is automatically included in implementations of a type in which an abstract variable is declared, unless the programmer provides a non-standard implementation (see below). If a type consists only of abstract variables (i.e. it corresponds to a record

or structure) there is an automatic implementation, which consists of the standard implementations of the individual abstract variables together with a parameterless constructor (known in Timor as a *maker*).

7.3 Client Access

A client accesses an abstract variable as if it were a concrete variable, just as he accesses a public field in the standard OO paradigm. The compiler transforms statements along the following lines:

```
p.dob = d1          =>          p.dob(d1)
d1 = p.dob          =>          d1 = p.dob()
```

The dot notation is used in the conventional way, e.g.

```
p.dob.reset();
```

This means that the `reset` method is applied to the `dob` element within `p`, as would be expected. An address operator (such as `&` in C++) has been deliberately avoided in Timor, because references are intended only to express logical relationships between complete objects, and pointers to their internal values would violate the information hiding principle. Thus a value within an object cannot be referenced externally, which means that the language deliberately excludes the splitting of the above statement into two parts such as:

```
Date* d = &p.dob; // invalid in Timor
d.reset();
```

This has the effect that the expressions

```
p.dob.reset();
```

and

```
(p.dob).reset();
```

have different meanings. The first resets the internal date in `p`, while the second copies the internal date (as in `d1 = p.dob` above) and then resets the copy. Hence in a dot expression the appearance of an abstract value name at the end of an expression returns a value, whereas its use in a non-terminating situation has the conventional object oriented meaning, corresponding to a pointer (which is not directly accessible to clients) that defines the context for the next element in the expression.

8 Non-standard Implementations of Abstract Variables

Timor gives an implementation programmer complete freedom to implement a type however he chooses. Consequently, if the type definition includes an abstract variable, the programmer is not required to include in his implementation a concrete variable with the same type and name as those of the abstract variable, although he can do so if he wishes.

To program a non-standard implementation of an abstract variable the programmer explicitly includes method implementations for the public set and get methods, plus any further internal methods and/or variables which these require.

We now consider various possible non-standard implementations of the type `Person`, as it appears in section 7.1, beginning with an implementation which includes a corresponding concrete variable for `Date dob`:

```
impl PersonImpl1 of Person {
 ...
state:
 Date dob;                      // a concrete variable
 Log log;
instance:
 op Date dob(Date dob) { // the set method implementation
  log.write("dob field modified");
  return this.dob = dob;
 }
 enq Date dob() {
  log.write("dob field read");
  return this.dob;
 }
 ...
}
```

This implementation is unproblematic, because the concrete variable `Date dob` provides the compiler with a starting point for interpreting the dot notation in expressions such as

```
p.dob.day = 3;
```

and the concrete variable `dob` by definition provides a method, for example, for setting the abstract variable `day`.

But suppose that the programmer decides on an implementation in which the `dob` details should be compacted into a single integer (equivalent to that discussed in section 6), which does not include a concrete variable corresponding to the abstract variable. Part of this implementation could be equivalent to that proposed for the Java version, e.g.

```
impl PersonImpl2 of Person {
 ...
state:
 int dateOfBirth; // holds the compacted date of birth value
instance:
 op Date dob(Date dob) { // the set method implementation
  year(dob.year()); month(dob.month()); day(dob.day()); return dob;
 }
 enq Date dob() { // the get method implementation
  return Date.init(day(), month(), year());
 }
 ...
// internal methods for format conversion
 op void day (int day) { // sets internal day from int format
  dateOfBirth = dateOfBirth - dateOfBirth % 100 + day;
 }
 enq int day() { // returns internal day in int format
  return dateOfBirth % 100;
 }
 op void month(int month) {// sets internal month from int format
  dateOfBirth =
   dateOfBirth - dateOfBirth % 10000 + month * 100 + dateOfBirth %100;
  }
 enq int month() { // returns internal month in int format
  return (dateOfBirth % 10000 - dateOfBirth % 100) / 100;
 }
 op void year(int year) { // sets internal year from int format
  dateOfBirth = year * 10000 + dateOfBirth % 10000;
 }
```

```
  enq int year() { // returns internal year in int format
   return dateOfBirth / 10000;
  }
 }
```

However, as this stands, there is still a problem when the client attempts to access a method (or abstract variable) of the nested abstract variable `Date dob`, because this cannot be located from a concrete variable, and in fact there is no equivalent to these methods. Consequently, if an abstract variable which has a structured type is given a non-standard implementation, this must include not only the set and get methods, but also an implementation of each of the public methods of the nested type.

Timor allows these methods to be implemented by allowing the dot notation to appear within a method name in an implementation which includes non-standard implementations of an abstract variable without a corresponding concrete variable. Thus an implementation of `Person` which includes a non-standard implementation of the abstract variable `Date dob` must also include implementations for the set and get methods `dob.day`, `dob.month`, `dob.year` as well as for the method `dob.reset`, e.g.

```
impl PersonImpl2 of Person {
... // as above plus:
instance:
 op int dob.day(int day) {day(day); return day;}
 enq int dob.day() {return day();}
 op int dob.month(int month) {month(month); return month;}
 enq int dob.month() {return month();}
 op int dob.year(int year) {year(year); return year;}
 enq int dob.year() {return year();}
 op void dob.reset() {day(0); month(0); year(0);}
}
```

In the case of several levels of nesting, the dot symbol appears more than once in method names.

9. Interpreting Dot Expressions

The alternatives outlined above for implementing nested abstract variables imply that elements in a dot expression can in practice be interpreted either as concrete variables (values and references) or as the names of methods. The latter can appear without the usual parameter brackets (if they correspond to set or get methods of an abstract variable) and they may include compound method names (such as `dob.day` in the last example).

For the case where no implementations of types are used which provide implementations of compound method names, the interpretation of a dot expression is equivalent to that for a conventional class based OO language (and can be optimised accordingly). However, the interpretation of dot expressions which potentially contain compound names is somewhat more complicated. To help clarify what this involves we now sketch out a mental model, while emphasizing that this need not be the technique actually implemented in a compiler.

To evaluate such a dot expression, an abstract variable `x` whose type `X` contains methods `m1`, `m2`, etc. is treated as a pair of set and get methods (cf. section 7.1):

```
op X x(X x);
enq X x();
```

plus a set of *nested methods* `x.m1`, `x.m2`, etc. having the same parameter and result types as the original methods `m1`, `m2`, etc. In a standard implementation of `x` (cf. section 7.2), these nested methods simply call the original methods on the concrete variable `x`, while in a non-standard implementation these methods are explicitly provided by the implementor. If the abstract variable's type `X` itself contains abstract variables `y`, `z`, etc. (whose types `Y` and `Z` might again contain abstract variables, and so on), these variables are recursively "expanded" into corresponding sets of method pairs before the abstract variable `x` is expanded.

To give a concrete example, the type `Date` defined in section 7.1 would be treated as having method pairs `day`, `month`, and `year` as well as a single method `reset`. Given that, the type `Person` defined there would be treated as having a method pair `dob` plus nested method pairs `dob.day`, `dob.month`, and `dob.year` as well as a single nested method `dob.reset`. Finally, if another type `Schizo` were defined as

```
type Schizo {
 ...
 Person p1, p2;
}
```

this would be treated as having method pairs `p1` and `p2` (which might be called *level 1 methods*) plus nested method pairs `p1.dob` and `p2.dob` (*level 2 methods*) plus doubly nested method pairs `p1.dob.day`, `p1.dob.month`, `p1.dob.year`, etc. (*level 3 methods*).

Based on this mental model, the interpretation of arbitrary dot expressions can be described as follows.

Using Java terminology, a *dot expression* in Timor is a *primary expression* whose *primary prefix* denotes an entity `e` of some type `T`, e.g., a variable, an explicit method call, or a nested dot expression surrounded by parentheses, each denoting either a value of type `T` or a reference to an object of type `T`. The corresponding *primary suffix* is a sequence of identifiers, each preceded by a dot, optionally followed by an argument list or an assignment operator and an arbitrary expression (the RHS of the assignment). In the latter case, the assignment operator is removed and the RHS of the assignment is transformed into a one-element argument list. If the identifier sequence is neither followed by an explicit argument list nor by an argument list resulting from such a transformation, an empty argument list is appended.

To evaluate such a dot expression, the *longest* method name found in `T` that is a prefix of the given identifier sequence is chosen (where the *length* of a method name is determined by the number of dots it contains). This method is invoked on the entity `e`, either with an empty argument list if its name is a *proper* prefix of the identifier sequence, or with the argument list following the identifier sequence if the method name is equal to the *complete* identifier sequence. In the latter case, the method's return value constitutes the value of the dot expression. Otherwise, it is treated as a new primary prefix, while the remaining sequence of identifiers becomes the corresponding new primary suffix; for these, the evaluation just described is performed recursively.

For the case where an implementation of a type contains compound names and this is the only implementation of the type in question, related dot expressions can be fully analysed at compile time. However, if a single program uses different implementations of the same type with different implementation techniques (e.g. one uses compound names, one uses concrete variables to implement the methods of nested abstract variables) then the interpretation of a dot expression can only be fully carried out by run-time code. This is likely to occur only in extremely unusual circumstances.

We anticipate that in almost all practical cases a full compile time analysis will be possible, since the technique which we have proposed for implementing abstract variables is likely to be used only in unusual situations (such as the implementation of a `Person` database, as discussed in the next section).

10. Discussion

In terms of the trivial example which we have used throughout the paper it may appear that the issue addressed is not a serious one. After all this example could have been much more easily managed simply by providing an implementation of `Date` that uses an integer representation internally. (In fact one of the strengths of Timor is that it allows a type to have multiple implementations in a straightforward way.) However, there are cases where it may be appropriate not to use such implementations in the implementation of an enclosing type. For example if a database of `Person` objects were to be implemented in a write-through file in such a way that the only representation of each object in the memory were a key which allows the intended database object to be located and accessed, then there would be no representation whatsoever for the abstract variables except in the file. Such examples can be implemented in Java using the proxy with back reference technique, and they can be implemented in Timor, despite its value semantics, using the technique described.
It may also appear that the approach leads to scalability problems, but in fact its scalability properties are comparable to those in Java, where for each non-standard implementation of an abstract variable that does not include a concrete variable of the same name an inner class (or static nested class) is required which implements all the same routines. (In the case that a Timor module includes several abstract variables of the same type which also use the same implementation, this can of course be shared.) On the other hand the approach is in principle more efficient than using Java proxies, as the overhead of creating and using extra proxy objects is avoided in Timor.
Finally we point out that the issue which has been discussed is completely orthogonal to other aspects of implementing types in a non-standard way. For example in Timor there is no direct relationship to the issue of implementing complex types such as a stack of stack of persons in a non-standard way: a type can be defined (generically or not) and it can have many implementations. There is no problem in Timor in providing a special implementation of such a type, and this does not raise any of the issues discussed in this paper.

11. Related Work

Although information hiding is a widely acknowledged software engineering technique it is normally not cleanly incorporated into the object orientation paradigm, which has preferred the simpler class construct (which integrates a type and a single implementation) to a more rigorous approach which separates types and their (potentially multiple) implementations. Techniques which are sometimes recommended as a means of providing multiple implementations for a type (such as abstract classes or Java interfaces as type definitions with their subclasses as implementations) fail to ensure that the full rigour of information hiding is achieved, and they have the disad-

vantage that the implementation classes themselves also introduce new types, which can for example tempt the programmer to introduce additional public methods and fields. But even when a programmer uses an abstract class or an interface as a type definition he must decide whether to export a variable or provide set and get methods. With Java interfaces he does not even have this choice: variables cannot be defined as part of an interface. This harmonises with the information hiding principle but not with the client's simple requirements of treating certain aspects of the interface in a natural way as exported data items.

Because in Eiffel a parameterless function and an *attribute* (variable) are accessed in the same way (according to the "principle of uniform reference" [6]) it is possible to redefine a parameterless function declared in a superclass as an attribute in a subclass. Notice however that the reverse is not possible, since it would only be possible fully to implement an attribute declared in a superclass by means of two methods (i.e. a set and a get method) in a subclass. Hence the principle of uniform reference does not produce the same effect as a Timor abstract variable.

In the language Sather abstract classes are used as type definitions which specify method signatures without implementations. In [7] this is motivated by the idea that a class may have different implementations in concrete classes. However, concrete classes which form the leaf nodes of the type graph can also be used as types. So called *attributes* are defined similarly to abstract variables in Timor, as a pair of set and get methods. If clients use a concrete class as a type, they may call the accessor routines, but attributes may also be used as if they were variables. In this case no problems arise with nested attributes or the dot notation because attributes may only appear in concrete classes, i.e. where the implementations are fixed as so called reference classes. As it is not possible to define attributes in abstract classes, the level of flexibility reached in Timor, where abstract variables can be defined in (nested) type definitions which can have multiple implementations, is not attained.

In the programming language Theta [5] a more convincing attempt has been made to support the information hiding principle, by strictly distinguishing between types and their implementations. But types must be defined in terms of methods, i.e. there are no abstract variables. In implementations (which are not types), a shorthand notation is provided to implement simple set and get methods by declaring appropriate variables with implements clauses. This is convenient for the implementor but not for the client of a type.

The programming language Tau [9] also provides rigorous support for the information hiding principle, allowing an interface (i.e. a type definition) to include "abstract instance fields" as members. As in Timor these correspond to get and set methods. In an implementation (which is called a class, but which in contrast to Java does not introduce a new type) this can be realised either as a concrete field or as a pair of methods. A client of the interface can access an abstract field syntactically as if it were a concrete field. As in Timor exceptions can be associated with the definition of abstract fields.

In the languages mentioned above the dot notation is interpreted in the context of reference semantics. They all support the concept of abstract type definitions which may have multiple and easily interchangeable implementations. In this sense they share the information hiding aim of Timor. None of these languages provides a mechanism for re-implementing nested methods or abstract variables in a non-standard implementation using the technique described in this paper, which is particularly relevant for languages which support value semantics.

Variables at the class interface defined as a pair of access methods which are automatically available to manipulate the state of an object can be found in other languages too. For example in CLOS [10] such variables are known as slots. As these slots appear within a class definition they are concrete variables with a standard implementation for their setters and getters, and they are by no means abstract in the sense of Timor. There is no dot notation because syntactically the functional style of LISP is used.

12. Concluding Remarks

The paper has shown how Timor takes the final step in reconciling a rigorous interpretation of the information hiding principle with unrestricted user convenience in the context of a language based on value semantics. On the one hand Timor fully supports the information principle, allowing any type to have multiple implementations (which are not themselves types). On the other hand, type definitions can, for the convenience of client programmers, apparently include (or consist entirely of) exported variables, which can be accessed syntactically as if they are concrete variables. Yet even in such cases the information hiding principle is not violated, and the implementation programmer is given complete freedom to implement a type in any way which he sees fit (provided that it accords with the type specification). This remains true even if a type is itself defined in terms of nested abstract variables, which the client programmer apparently reaches using the dot notation and even if there are no real concrete variables corresponding to the steps in a dot expression.
The basic technique is well known: a variable can be considered as equivalent to a pair of methods for setting and getting a value (or a reference). However, we have seen that when instances of such types are nested the reconciliation of the two concepts is by no means as simple as might at first be thought, especially with respect to the interpretation of dot expressions.

Acknowledgements

Special thanks are due to Dr. Mark Evered and Dr. Axel Schmolitzky for their invaluable contributions to discussions of Timor and its predecessor projects. Without their ideas and comments Timor would not have been possible. We also thank the referees for their helpful comments, which have resulted in considerable improvements in the presentation of the paper.

References

[1] J. L. Keedy, G. Menger, and C. Heinlein, "Support for Subtyping and Code Re-use in Timor," 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia, 2002, Conferences in Research and Practice in Information Technology, vol. 10, pp. 35-43.

[2] J. L. Keedy, G. Menger, and C. Heinlein, "Inheriting from a Common Abstract Ancestor in Timor," *Journal of Object Technology (www.jot.fm)*, vol. 1, no. 1, pp. 81-106, 2002.

[3] J. L. Keedy, G. Menger, C. Heinlein, and F. Henskens, "Qualifying Types Illustrated by Synchronisation Examples," in *Objects, Components, Architectures, Services and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany*, vol. LNCS 2591, M. Aksit, M. Mezini, and R. Unland, Eds.: Springer, 2003, pp. 330-344.

[4] J. L. Keedy, K. Espenlaub, G. Menger, and C. Heinlein, "Qualifying Types with Bracket Methods in Timor," *Journal of Object Technology* (to appear 2004).

[5] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A. C. Myers, "Theta Reference Manual," MIT Laboratory for Computer Science, Cambridge, MA, Programming Methodology Group Memo 88, February 1994.

[6] B. Meyer, *Object-oriented Software Construction*. New York. Prentice-Hall, 1988.

[7] N. Nemec, B. Gomes, D. Stoutamire, B. Vaysman, and H. Klawitter, "Sather - A Language Tutorial," GNU Sather Package.

[8] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, 1972.

[9] A. Schmolitzky, "Ein Modell zur Trennung von Vererbung und Typabstraktion in objektorientierten Sprachen (A Model for Separating Inheritance and Type Abstraction in Object Oriented Languages)," *Ph.D. Thesis, Dept. of Computer Structures*: University of Ulm, Germany, 1999.

[10] P. H. Winston and B. K. P. Horn, *Lisp*, 3rd ed. Addison-Wesley, Reading, MA, 1989.