Read-only by specification

K. Rustan M. Leino and Greg Nelson
11 September 1995

Digital's Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, U.S.A.
{rustan,gnelson}@pa.dec.com

When programming with classes in an object-oriented language, a data field declared in one class can be made *publicly* available. That is, any scope that imports the class declaration gets access to the data field. It is often desirable to make some data fields of a class accessible only to the class itself, a commonly provided feature among programming languages (in C++ [0], this access control is called *private*). Yet another common paradigm is to make some data fields accessible only to the class implementation itself and the implementations of its subclasses (called *protected* in C++).

Different programming languages handle access control in different ways. C++, for example, hardwires the three kinds of access control described above, whereas Modula-3 [4] permits an arbitrary number of access control kinds. In either case, access control is about full control, *i.e.*, a scope can either both read and write a field or it can do neither. In this note, we discuss how to make some fields read-only to some scopes. We present a beautiful solution to declaring some fields read-only to some scopes and to formally reasoning about read-only fields.

0 Warm-up: Read-only to clients and subclasses

We begin by showing some common and simple mechanisms by which a class can declare fields that are private to the implementation and read-only to any scope outside the implementation. The idea is to make the data field itself private and to furnish a different way to access its value.

Here's an example written in something like C++.

```
class T
{
private:
  int x;
public:
  int QueryX();
};

int T::QueryX()    { return x; }
```

The `class` construct declares a new class T. This class has a data field x and a method $QueryX$. The example also shows the implementation of method $QueryX$, which simply returns the x data field. Although x is directly accessible only to the implementation of class T (because it is declared as private), the value of x is publicly accessible via method $QueryX$. In effect, what has been achieved is to make x read-only.

The works but is not free of nuisance. As part of this solution, a client accesses the value of x for a T object t not by $t.x$ but by $t.QueryX()$. (Eiffel [3] made the syntax of invoking parameter-less functions such that clients need not be aware of whether they're invoking a method or accessing the field directly.) The implementor, too, has more typing to do than, say, just marking x as being "write private" and "read public".

Efficiency is also of possible concern, since instead of just dereferencing a pointer, this solution requires a procedure call. C++ fixes that by allowing $QueryX$ to be declared as *inline*. There is also an opportunity, in general, to inline these short procedures at link-time.

An example similar to the one above but in which x is readable only by the implementation of T and its subclasses is easy to construct (change "`public:`" to "`protected:`").

To summarize, in common programming languages, having a data field in scope means being able to read *and* write it. The ability to make a field read-only comes from providing a public procedure that retrieves the value of the field, which itself is declared to be private.

1 Trouble: Privately read-only, writable by subclasses

In the previous section, we mentioned the case where the T, the class that declares x, can read and write x, and the subclasses of T can but read x. What about doing it the other way around, *i.e.*, letting subclasses read and write the field but restricting the class that declares x to only read the field?

Such a question is interesting academically, but has practical merit only if this is a useful programming paradigm. Therefore, let us describe a real-life example of the use of such a paradigm.

Readers

A *reader* implements an input stream. Different readers may have different sources (*e.g.*, a disk, a terminal, a text string in memory). Common to all readers is the way they buffer characters between the source and the $GetChar$ procedure. The SRC Modula-3 library [1], for example, features such an abstraction of readers. The reader base class reveals to its subclasses that it uses a buffer and various pointers into that buffer. Four of those pointers are st, lo, cur, hi, which are to satisfy, for any reader rd, the inequalities

$$rd.lo \leq rd.cur \leq rd.hi \quad . \tag{0}$$

Furthermore, there is a condition that states that

$$rd.st + rd.hi - rd.lo \leq \text{the size of the buffer} \quad . \tag{1}$$

In the interface that reveals these fields to reader subclasses, there is an English comment that says

> The class-independent code modifies $rd.cur$, but no other variables revealed in this interface [. . .].

This is of great importance to subclasses, because it allows a subclass to monarchically control the exact values of these other fields, subject to maintaining the invariants (0) and (1).

For example, the class of *message readers* keeps st to the size of a message header. In other words, message readers use the first part of the buffer for storing the header of received messages. This header is not part of the source of the reader, but its information is used by the message reader and must therefore not be clobbered by the class-independent code.

As another example, the initialization of a *text reader* takes a text string as an argument and makes the text string into the source of the reader. It does so by copying the source directly into the reader's buffer, after which it no longer keeps track of the given text string. Hence, unless the buffer is kept intact, the source of this reader is destroyed. Furthermore, the st and lo fields of text readers are set to 0 during initialization. The correctness of the text reader methods relies on the fact that these fields are still 0 upon entry to the methods.

The question is thus: How can the English comment be recognized mechanically?

A clunky solution

The problem is not without solution in C++ and Modula-3, but the solution is at best suboptimal. The solution is to declare the actual data fields in the subclasses rather than in the base class, despite the fact that all readers contain these attributes. So that the base class can read the values of these fields, it provides methods like $QuerySt$, $QueryLo$, and $QueryHi$, which are replaced by the subclasses. Each subclass thus returns its respective value of st, lo, hi,

This time, we're not just trading a pointer dereference for a procedure call, but we're trading it for an indirect procedure call. Such calls cannot be inlined by an optimizing linker.

We encountered this problem by trying to write a formal specification of readers. At first, that might seem an even more difficult problem, since one may need to formally state what it means for a field to be read-only to "other" scopes. As it turns out, we found a beautiful and simple solution using only the specification constructs with which we were already equipped.

2 Enter specifications

We continue in the context of the reader classes presented above. A reader is said to be *valid* just when its class invariant holds. In the public interface of readers, we declare $valid$ to be an *abstract* field, that is, a field that can be used only in specifications and cannot be directly accessed by the code. It is to be $true$ just when a reader is valid. The mapping from the (*concrete*) state of a reader to its abstract state is captured in the *representation invariant* (or "*rep*" for short) of an abstract field.

The rep of $valid$ is not given in the public interface, because those details are unimportant to reader clients. Instead, the public interface simply states that each operation on a reader requires (as a precondition) that the reader is valid. Hence, it is in the client's best interest to keep a reader valid. This is easy for most casual clients, because the only public operation on readers that destroys validity is $Close$; all other operations maintain validity.

Let's now view readers from the perspective of a reader *friend, i.e.,* the perspective of those clients or subclass implementations that need to know more about the details of reader implementations. (C++ buffs may think of this view as the "protected" view.)

In this friends interface, the fields st, lo, cur, hi and the buffer are revealed. Furthermore, a new abstract field, $svalid$, is declared. The idea is that $svalid$ is $true$ just when the class-specific part of a reader's class invariant holds.

The friends interface gives the rep for $valid$, which states that for all readers rd,

$$rd.valid \equiv (0) \wedge (1) \wedge rd.svalid \quad . \tag{2}$$

To cater for sound modular verification, the appropriate *representation dependencies* must be declared before one is permitted to write down this rep (see [2]). That involves declaring that

$rd.valid$ *depends* on every field given in (2). For example, $rd.valid$ depends on $rd.svalid$. This means that if the value of $rd.svalid$ changes, the value of $rd.valid$ may also change.

At first sight, declaring these dependencies seems unnecessary because one can infer them from the rep. However, the rep is not always in scope. For example, the rep of $svalid$ is different for each subclass. The rep for $svalid$ declared by text writers will say something like

$$rd.svalid \equiv rd.st = 0 \wedge rd.lo = 0 \quad ,$$

whereas the rep for $svalid$ declared by message readers will be quite different. In order to allow the subclasses to write their respective reps for $svalid$, we must declare the appropriate dependencies. And those dependencies must be declared in this friends interface (see [2]).

We thus declare that $rd.svalid$ depends on $rd.st$, $rd.lo$, and $rd.hi$. Again, this allows subclasses to mention these fields in their respective reps for $svalid$. Note, however, that it does not require subclasses to declare reps for $svalid$ that actually depend on these fields—it simplify allows them to.

There should not be a dependency of $rd.svalid$ on $rd.cur$, because the precise value of $rd.cur$ is something a subclass is not allowed to rely on. All a subclass knows about a valid reader is that $rd.cur$ satisfies (0).

But there it is! We have just specified that the class-independent code is not allowed to change the values of st, lo, hi. Why is that? Since all class-independent procedures (except $Close$) require and maintain validity, they can assume $valid$ as a precondition and must ensure that $valid$ is also a postcondition. The class-independent code will have the friends interface in scope (so as to get access to the data fields on readers), so it has information that $valid$ depends on $svalid$. That means that any change to $svalid$ may cause a change of $valid$. In fact, since the rep of $valid$, (2), is also in scope, it is known exactly how a change of $svalid$ affects the value of $valid$.

Moreover, the friends interface shows the dependencies, for every reader rd, of $rd.svalid$ on $rd.st, rd.lo, rd.hi$. Hence, a change in any of those fields may cause a change in the value of $rd.svalid$. But to know the exact effect on $rd.svalid$ of such changes, the rep of $rd.svalid$ must be in scope. Without knowing the rep of $rd.svalid$, it is impossible to prove that a change of $rd.st$, $rd.lo$, or $rd.hi$ does *not* alter the value of $rd.svalid$. Remember that the reps for $rd.svalid$ for different subclasses are given by the subclasses themselves, so the class-independent code does not have access to these reps, by definition of “class-independent”.

Note that the class-independent code can modify $rd.cur$ and still hope to maintain validity, because $rd.svalid$ does not depend on $rd.cur$. $rd.valid$ does depend on $rd.cur$, but the rep of $rd.valid$ is in scope.

Summary

The solution is the following, written in a C++-like notation.

```
class Reader
{
public :
  abstract boolean valid;
  char GetChar();
    requires valid;
  ⋮
protected :
  int st, lo, cur, hi;
  Buffer buff ;
  abstract boolean svalid;
  depends valid on st, lo, cur, hi, buff , svalid;
  rep valid  ≡  lo ≤ cur ≤ hi  ∧  st + hi − lo ≤ size of buff  ∧  svalid;
  depends svalid on st, lo, hi, buff ;
⋮
};
```

This declaration indicates that procedures like *GetChar* require *valid* and do not change it. Any change by *GetChar* of any of *svalid* 's dependencies must then be accompanied by a proof that the value of *svalid* remains unchanged (otherwise, as the rep of *valid* reveals, *valid* would change, too). Since the class-independent code has no rep for *svalid* , it can prove that it leaves *svalid* unchanged only by not changing any of *svalid* 's dependencies. This makes *svalid* read-only for the class-independent code.

It is important to realize that it is a reader's reps for *valid* and *svalid* that determine what exactly makes a reader valid. However, it is the dependencies that provide a mechanism for detecting violations of design and implementation errors.

Note, by the way, that any (permitted) read or write of the fields st, lo, hi is compiled directly into a pointer dereference.

The solution also works for fields that are declared as public and that are read-only except to the class implementation (and/or the implementation of the subclasses). In fact, the solution very clearly tells what scopes are allowed to modify the fields: If a is declared to depend on c , a scope can prove that a modification of c does not affect the value of a only if it has the rep of a in scope.

3 Conclusion

The key idea in object-oriented programming is to organize data into classes and subclasses in such a way that the data and behavior of a class are those that are common to all its subclasses. Sometimes, some data fields of a class are read by the class but updates to these fields are done only in the subclasses. In this note, we have described a novel mechanism by which a class can declare some fields and give away its right to modify those fields.

Not only is the solution beautiful, but it also draws only from abstraction, specifications, and dependencies—three elements that already are crucial to reasoning about the correctness of modular programs. We have for this problem no solution that does not use dependencies. Without dependencies, the problem seems extremely difficult, but with dependencies, the solution falls easily and delightfully into place.

References

[0] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1990.

[1] J.J. Horning, B. Kalsow, P. McJones, and G. Nelson. Some useful modula-3 interfaces. Research Report 113, DEC SRC, 130 Lytton Ave., Palo Alto, CA 94301, U.S.A., December 1993.

[2] K.R.M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[3] B. Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.

[4] G. Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ, 1991.