L6: Abstract Data Types

Today

- Abstract data types
- Representation independence
- Rep exposure
- Abstraction function & rep invariant

Required Reading (from the Java Tutorial)

- Interfaces
- Collection Interfaces (focus on Set, List, and Map)
- Collection Implementations (again, Set, List, and Map, but also Wrapper and Convenience)

In this lecture, we look at a powerful idea, abstract data types, which enable us to separate how we use a data structure in a program from the particular form of the data structure itself. Abstract data types address a particularly dangerous dependence, that of a client of a type on the type's representation. We'll see why this is dangerous and how it can be avoided. We'll also discuss the classification of operations, and some principles of good design for abstract data types.

What Abstraction Means

Abstract data types are an instance of a general principle in software engineering, which goes by many names with slightly different shades of meaning. Here are some of the names that are used for this idea:

- **Abstraction**. Omitting or hiding low-level details with a simpler, higher-level idea.
- **Modularity**. Dividing up a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system.
- **Encapsulation**. Building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can't damage its integrity.
- **Information hiding**. Hiding details of a module's implementation from the rest of the system, so that the those details can be changed later without changing the rest of the system.
- **Separation of concerns**. Making a feature (or "concern") the responsibility of a single module, rather than spreading it across multiple modules.

As a software engineer, you should know these terms, because you will run into them frequently. The fundamental purpose of all of these ideas is to help achieve the three important properties that we care about in 6.005: safety from bugs, ease of understanding, and readiness for change.

User-Defined Types

In the early days of computing, a programming language came with built-in types (such as integers, booleans, strings, etc.) and built-in procedures, eg. for input and output. Users could define their own procedures: that's how large programs were built.

These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Copyright held by the authors.

A major advance in software development was the idea of abstract types: that one could design a programming language to allow user-defined types too. This idea came out of the work of many researchers, notably Dahl (the inventor of the Simula language), Hoare (who developed many of the techniques we now use to reason about abstract types), Parnas (who coined the term *information hiding* and first articulated the idea of organizing program modules around the secrets they encapsulated), and here at MIT, Barbara Liskov and John Guttag, who did seminal work in the specification of abstract types, and in programming language support for them -- and developed 6170, the predecessor to 6.005. In 2010, Barbara Liskov earned the Turing Award, computer science's equivalent of the Nobel Prize, for her work on abstract types.

The key idea of data abstraction is that a type is characterized by the operations you can perform on it. A number is something you can add and multiply; a string is something you can concatenate and take substrings of; a boolean is something you can negate, and so on. In a sense, users could already define their own types in early programming languages: you could create a record type date, for example, with integer fields for day, month and year. But what made abstract types new and different was the focus on operations: the user of the type would not need to worry about how its values were actually stored, in the same way that a programmer can ignore how the compiler actually stores integers. All that matters is the operations.

In Java, as in many modern programming languages, the separation between built-in types and user-defined types is a bit blurry. The classes in java.lang, such as Integer and Boolean are built-in; whether you regard all the collections of java.util as built-in is less clear (and not very important anyway). Java complicates the issue by having primitive types that are not objects. The set of these types, such as `int` and `boolean`, cannot be extended by the user.

Classifying Types and Operations

Types, whether built-in or user-defined, can be classified as mutable or immutable. The objects of a mutable type can be changed: that is, they provide operations which when executed cause the results of other operations on the same object to give different results. So `Date` is mutable, because you can call `setMonth` and observe the change with the `getMonth` operation. But `String` is immutable, because its operations create new string objects rather than changing existing ones. Sometimes a type will be provided in two forms, a mutable and an immutable form. `StringBuilder`, for example, is a mutable version of `String` (although the two are certainly not the same Java type, and are not interchangeable).

The operations of an abstract type are classified as follows:

- *Creators* create new objects of the type. A constructor may take an object as an argument, but not an object of the type being constructed.
- *Producers* create new objects from old objects of the type. The `concat` method of `String`, for example, is a producer: it takes two strings and produces a new one representing their concatenation.
- *Mutators* change objects. The `add` method of `List`, for example, mutates a list by adding an element to the end.
- *Observers* take objects of the abstract type and return objects of a different type. The `size` method of `List`, for example, returns an integer.

We can summarize these distinctions schematically like this:

creator: $t^* \rightarrow T$
producer: $T^+, t^* \rightarrow T$
mutator: $T^+, t^* \rightarrow void$

observer: $T^+, t^* \rightarrow t$

These show informally the shape of the signatures of operations in the various classes. Each T is the abstract type itself; each t is some other type. In general, when a type is shown on the left, it can occur more than once. For example, a producer may take two values of the abstract type; string concat takes two strings. The occurrences of t on the left may also be omitted; some observers take no non-abstract arguments (e.g., `size`), and some take several.

Here are some examples of abstract data types, along with their operations:

`int` is Java's primitive integer type. `int` is immutable, so it has no mutators.

creators: the numeric literals 0, 1, 2,
producers: arithmetic operators +, −, ×, ÷
observers: comparison operators ==, !=, <, >
mutators: none (it's immutable)

List is Java's list interface. List is mutable. List is also an *interface*, which means that other classes provide the actual implementation of the data type. These classes include ArrayList and Linked List.

creators: ArrayList constructor, LinkedList constructor, Collections.singletonList()
producers: Collections.unmodifiableList()
observers: size(), get()
mutators: add(), remove(), addAll(), Collections.sort()

String is Java's string interface. String is immutable.

creators: String(), String(char[]) constructors
producers: concat(), substring(), toUpperCase()
observers: length(), charAt()
mutators: none (it's immutable)

This classification gives some useful terminology, but it's not perfect. In complicated data types, there may be an operation that is both a producer and a mutator, for example. Some people use the term *producer* to imply that no mutation occurs.

Designing an Abstract Type

Designing an abstract type involves choosing good operations and determining how they should behave. A few rules of thumb.

It's better to have a few, simple operations that can be combined in powerful ways than lots of complex operations.

Each operation should have a well-defined purpose, and should have a coherent behavior rather than a panoply of special cases. We probably shouldn't add a `sum` operation to `List`, for example. It might help clients who work with lists of `Integers`, but what about lists of `Strings`? Or nested lists? All these special cases would make `sum` a hard operation to understand and use.

The set of operations should be *adequate*; there must be enough to do the kinds of computations clients are likely to want to do. A good test is to check that every property of an object of the type can be extracted. For example, if there were no `get` operation, we would not be able to find out what the elements of a list are. Basic information should not be inordinately difficult to obtain. The `size` method is not strictly necessary for List, because we could apply `get` on increasing indices until we get a failure, but this is inefficient and inconvenient.

The type may be generic: a list or a set, or a graph, for example. Or it may be domain-specific: a street map, an employee database, a phone book, etc. But it should not mix generic and domain-specific features. A `Deck` type intended to represent a sequence of playing cards shouldn't have a generic `add` method that accepts arbitrary objects (like integers or strings). Conversely, it wouldn't make sense to put a domain-specific method like `dealCards` into the generic type `List`.

Representation Independence

A good abstract data type should be *representation independent*. This means that the use of an abstract type is independent of its representation (the actual data structure or data fields used to implement it), so that changes in representation have no effect on code outside the abstract type itself. For example, the operations offered by `List` are independent of whether the list is represented as a linked list or as an array.

You won't be able to change the representation of an ADT at all unless its operations are fully specified with preconditions (requires), postconditions (effects), and frame conditions (modifies), so that clients know what to depend on, and you know what you can safely change.

Preserving Invariants

Finally, and perhaps most important, a good abstract data type should preserve its own invariants. An *invariant* is a property of a program that is always true. Immutability is one crucial invariant that we've already encountered: once created, an immutable object should always represent the same value, for its entire lifetime.

When an ADT preserves its own invariants, reasoning about the code becomes much easier. If you can count on the fact that `Strings` never change, you can rule out that possibility when you're debugging code that uses `Strings` — or when you're trying to establish an invariant for another ADT. Contrast that with a string class that guarantees that it will be immutable only if its clients promise not to change it. Then you'd have to check all the places in the code where the string might be used.

Immutability

We'll see many interesting invariants. Let's focus on immutability for now. Here's a specific example:

```
public class Transaction {
    public int amount;
    public Calendar date;

    public Transaction(int amount, Date date) {
        this.amount = amount;
        this.date = date;
    }
}
```

How do we guarantee that `Transaction` objects are immutable — that, once a transaction is created, its date and amount can never be changed?

The first threat to immutability comes from the fact that clients can (in fact, must!) directly access its fields. So nothing's stopping us from writing code like this:

```
Transaction t = new Transaction(10, new Calendar ());
t.amount += 10;
```

This is a trivial example of *representation exposure*, meaning that code outside the class can modify the representation directly. Rep exposure like this threatens not only invariants, but also representation independence. We can't change the implementation of `Transaction` without affecting all the clients who are directly accessing those fields.

Fortunately, Java gives us language mechanisms to deal with this kind of rep exposure:

```
public class Transaction {
    private final int amount;
    private final Calendar date;

    public Transaction(int amount, Calendar date) {
        this.amount = amount;
        this.date = date;
    }

    public int getAmount() {
        return amount;
    }

    public Calendar getDate() {
        return date;
    }
}
```

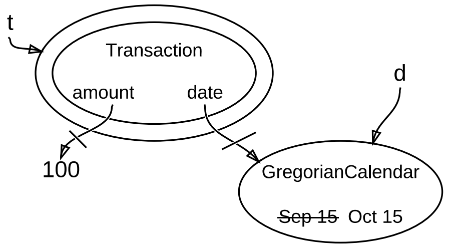
The *private* and *public* keywords indicate which fields and methods are accessible only within the class and which can be accessed from outside the class. The *final* keyword also helps by guaranteeing that the fields of this immutable type won't be reassigned after the object is constructed.

But that's not the end of the story: the rep is still exposed! Consider this (perfectly reasonable) client code that uses `Transaction`:

```
/** @return a transaction of same amount as t, one month later */
public static Transaction makeNextPayment(Transaction t) {
    Calendar d = t.getDate();
    d.add(Calendar.MONTH, 1);
    return new Transaction (t.getAmount(), d);
}
```

`makeNextPayment` takes a transaction and should return another transaction for the same amount but dated a month later. The `makeNextPayment` method might be part of a system that schedules recurring payments.

What's the problem here? The `getDate` call returns a reference to the *same* calendar object referenced by transaction `t`. So when the calendar object is mutated by add(), this affects the date in `t` as well:



`Transaction`'s immutability invariant has been broken. The problem is that `Transaction` leaked out a reference to a mutable object that its invariant depended on. We *exposed the rep*, in such a way that `Transaction` can no longer guarantee that its objects are immutable. Perfectly reasonable client code created a subtle bug.

We can patch this kind of rep exposure by *defensive copying*: making a copy of a mutable object to avoid leaking out references to the rep. Here's the code:

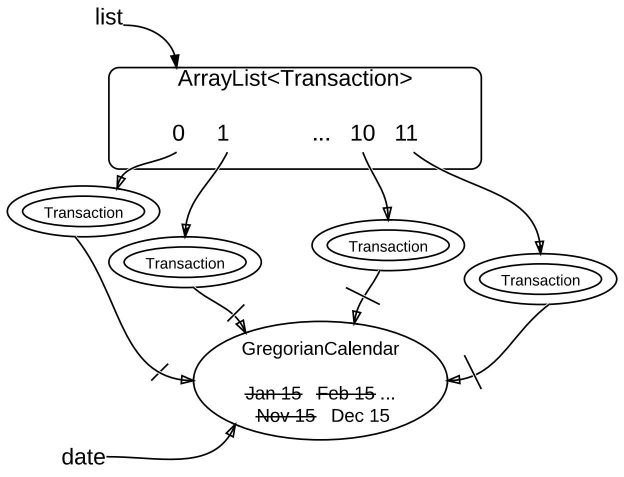
```
    public Calendar getDate() {
        return (Calendar)date.clone();
    }
```

clone() is probably the best way to do this with Calendar (despite the unfortunate problems with clone() in general – see Josh Bloch, *Effective Java*, item 10). Other classes offer a copy constructor, like StringBuilder(String).

But we're not done yet! There's still rep exposure. Consider this (again perfectly reasonable) client code:

```
    /** @return a list of 12 monthly payments of identical amounts */
    public static List<Transaction> makeYearOfPayments (int amount) {
        List<Transaction> list = new ArrayList<Transaction> ();
        Calendar date = new GregorianCalendar ();
        for (int i=0; i < 12; i++) {
            list.add (new Transaction (amount, date));
            date.add (Calendar.MONTH, 1);
        }
        return list;
    }
```

This code intends to advance a single Calendar object through 12 months, creating a transaction for each date. But notice that the constructor of `Transaction` saves the reference that was passed in, so all 12 transaction objects end up pointing to the same date:



Again, the immutability of `Transaction` has been violated. We can fix this problem too by judicious defensive copying, this time in the constructor:

```
public Transaction(int amount, Calendar date) {
    this.amount = amount;
    this.date = (Calendar)date.clone();
}
```

In general, you should carefully inspect the argument types and return types of all your ADT operations. If any of the types are mutable, make sure your implementation doesn't return direct references to its representation.

You may object that this seems wasteful. Why make all these copies of dates? Why can't we just solve this problem by careful specification:

```
/**
 * ...
 * @param date Date of transaction.  Caller must never mutate date again!
 */
public Transaction(int amount, Calendar date) { ...
```

This approach is sometimes taken when there isn't any other reasonable alternative – for example, when the mutable object is too large to copy efficiently. But the cost in your ability to reason about the program, and your ability to *avoid bugs*, is enormous. In the absence of compelling arguments to the contrary, it's almost always worth it for an abstract data type to guarantee its own invariants, and preventing rep exposure is essential to that.

An even better solution is to prefer immutable types. If we had used an immutable date object instead of the mutable `Calendar`, then we would have ended this section after talking about `public` and `private`. No rep exposure would have been possible.

The Java Collections classes offer an interesting compromise: *immutable wrappers*. Collections.unmodifiableList() takes a (mutable) List and wraps it with an object that looks like a List, but whose mutators are disabled – set(), add(), remove() throw exceptions. So you can construct a list using mutators, then seal it up in an unmodifiable wrapper (and throw away your reference to the original mutable list), and get an immutable list. The downside here is that you get immutability at runtime, but not at compile time – Java won't warn you at compile time if you try to sort() this unmodifiable list. You'll just get an exception at runtime.

How to establish invariants

An invariant is a property that is true for the entire program – which in the case of an invariant about an object, reduces to the entire lifetime of the object.

If the object is a state machine, then we need to:

- establish invariant in the initial state
- ensure that all state transitions preserve the invariant

So your creators and producers must establish the invariant for new instances, and all mutators (and observers, too, but particularly mutators) must preserve it.

Immutable types are simpler, because they have only one state to reason about.

The risk of rep exposure makes the situation more complicated. So the full rule for proving invariants is:

Structural induction: If an invariant of an abstract data type is

(1) established by creators;

(2) preserved by producers, mutators, and observers;

and (3) no rep exposure occurs,

then the invariant is true of all instances of the abstract data type.

Rep Invariant and Abstraction Function

We now take a deeper look at the theory underlying abstract data types. This theory is not only elegant and interesting in its own right; it also has immediate practical application to the design and implementation of abstract types. If you understand the theory deeply, you'll be able to build better abstract types, and will be less likely to fall into subtle traps.

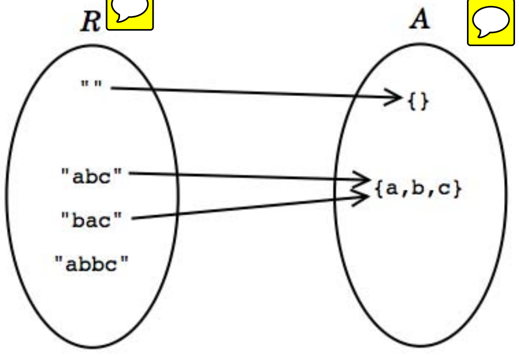
In thinking about an abstract type, it helps to consider the relationship between two spaces of values.

The space of *rep* or *representation* values consists of the values of the actual implementation entities. In simple cases, an abstract type will be implemented as a single object, but more commonly a small network of objects is needed, so this value is actually often something rather complicated. For now, though, it will suffice to view it simply as a mathematical value.

The space of *abstract* values consists of the values that the type is designed to support. These are a figment of our imagination. They're platonic entities that don't exist as described, but they are the way we want to view the elements of the abstract type, as clients of the type. For example, an abstract type for unbounded integers might have the mathematical integers as its abstract value space; the fact that it might be implemented as an array of primitive (bounded) integers, say, is not relevant to the user of the type.

Now of course the implementor of the abstract type must be interested in the representation values, since it is the implementor's job to achieve the illusion of the abstract value space using the rep value space.

Suppose, for example, that we choose to use a string to represent a set of characters. Then these form our two value spaces. We can show the two value spaces graphically, with an arc from a rep value to the abstract value it represents:



There are several things to note about this graph:

- Every abstract value is mapped to. The purpose of implementing the abstract type is to support operations on abstract values. Presumably, then, we will need to be able to create and manipulate all possible abstract values, and they must therefore be representable.
- Some abstract values are mapped to by more than one rep value. This happens because the representation isn't a tight encoding. There's more than one way to represent an unordered set of characters as a string.

Not all rep values are mapped. In this case, the string `"abbc"` is not mapped. If the type of threp is nontrivial, it will not make sense to give an interpretation for all rep values. A doubly-linked list representation, for example, can be twisted into all kinds of pretzel configurations that won't correspond to simple sequences, and for which we won't want to write special cases in the code. Or sometimes we will want to impose certain properties on the rep to make the code of the operations more efficient or easier to write. In this case, we have decided that the array should not contain duplicates. This will allow us to terminate the `remove` method when we hit the first instance of a particular character, since we know there can be at most one.

In practice, we can only illustrate a few elements of the two spaces and their relationships; the graph as a whole is infinite. So we describe it by giving two things:

An *abstraction function* that maps rep values to the abstract values they represent:

$$AF : R \rightarrow A$$

The arcs in the diagram show the abstraction function. In the terminology of functions, the properties we discussed above can be expressed by saying that the function is onto, not necessarily one-to-one, and often partial.

A *rep invariant* that maps rep values to boolean:

$$RI : R \rightarrow \text{boolean}$$

For a rep value *r*, *RI r* is true if and only if *r* is mapped by *AF*. In other words, *RI* tells us whether a given rep value is well-formed. Alternatively, you can think of *RI* as a set: it's the subset of rep values on which *AF* is defined.

A common confusion students have about abstraction functions and rep invariants is that they imagine that they are determined by the choice of rep and abstract value spaces, or even by the abstract value space alone. If this were the case, they would be of little use, since they would be saying something redundant that's already available elsewhere.

It's easy to see why the abstract value space alone doesn't determine *AF* or *RI*: there can be several representations for the same abstract type. A set of characters could equally be represented as a string, as above, or as a bit vector, with one bit for each possible character. Clearly we need two separate functions to map these two different rep value spaces.

It's less obvious why the choice of both spaces doesn't determine *AF* and *RI*. The key point is that defining a type for the rep, and thus choosing the values for the space of rep values, does not determine which of the rep values will be deemed to be legal, and of those that are legal, how they will be interpreted. Rather than deciding, as we did above, that the strings have no duplicates, we could instead allow duplicates, but at the same time require that the characters be sorted, appearing in nondecreasing order. This would allow us to perform a binary search on the string and thus check membership in logarithmic rather than linear time. Same rep value space — different rep invariant.

Even with the same type for the rep value space and the same rep invariant *RI*, we might still have different interpretations *AF*. Suppose *RI* admits any string of characters. Then we could define *AF*, as above, to interpret the array's elements as the elements of the set. But there's no *a priori* reason to let the rep decide the interpretation. Perhaps we'll interpret consecutive pairs of characters as subranges, so that the string `"acgg"` represents the set `{a,b,c,g}`.

The essential point is that designing an abstract type means not only choosing the two spaces — the abstract value space for the specification and the rep value space for the implementation — but also deciding what rep values to use and how to interpret them.

Example: Rational Numbers

Here's an example of an abstract data type for rational numbers. Look closely at its rep invariant and abstraction function.

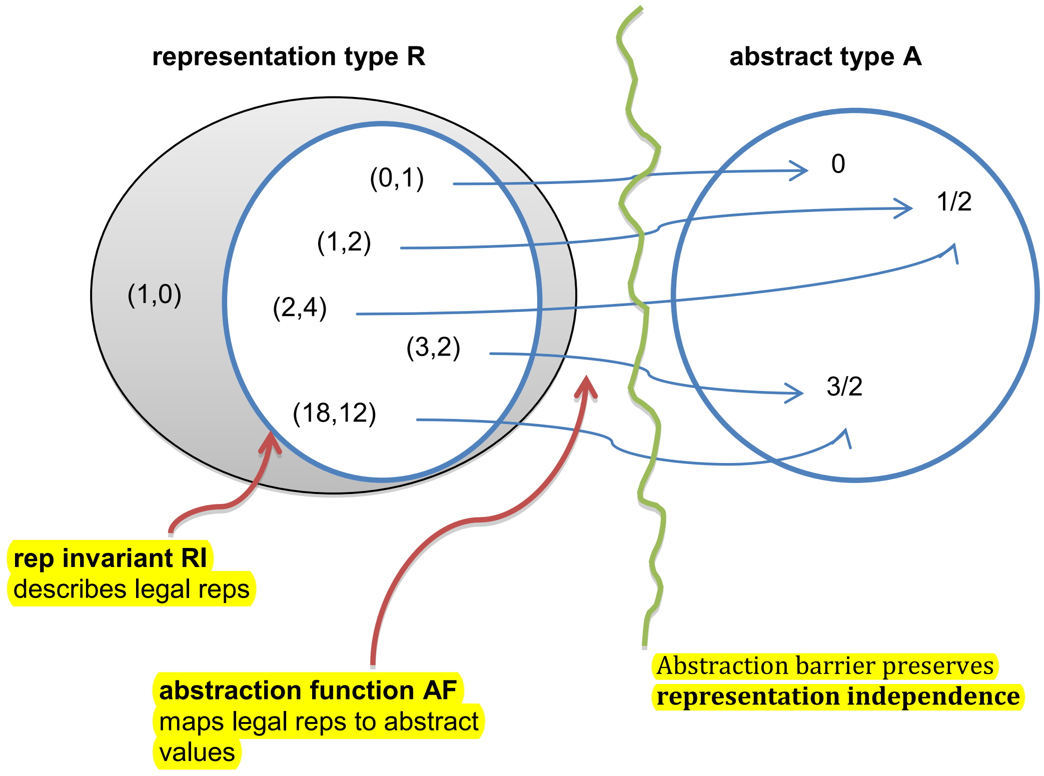
```
public class RatNum {
    private final int numer;
    private final int denom;

    // Rep invariant:
    //   denom > 0
    //   numer/denom is in reduced form

    // Abstraction Function:
    //   represents the rational number numer / denom

    /** Make a new Ratnum == n. */
    public RatNum(int n) {
        numer = n;
        denom = 1;
        checkRep();
    }

    /**
     * Make a new RatNum == (n / d).
     * @param n numerator
     * @param d denominator
     * @throws ArithmeticException if d == 0
     */
    public RatNum(int n, int d) throws ArithmeticException {
        // reduce ratio to lowest terms
        int g = gcd(n, d);
        n = n / g;
        d = d / g;
        // make denominator positive
        if (d < 0) {
            numer = -n;
            denom = -d;
        } else {
            numer = n;
            denom = d;
        }
        checkRep();
    }
}
```



Checking Rep Invariants and Implementing Abstraction Functions

The rep invariant isn't just a neat mathematical idea. If your implementation asserts the rep invariant at run time, then you can catch bugs early.

```
    // Check that the rep invariant is true
    // *** Warning: this does nothing unless you turn on assertion checking
    // by passing -enableassertions to Java
    private void checkRep() {
        assert denom > 0;
        assert gcd(numer, denom) == 1;
    }
```

toString() is a useful place to implement the abstraction function:

```
    /**
     * @return a string representation of this rational number
     */
    // This effectively implements the abstraction function
    public String toString() {
        return (denom > 1) ? (numer + "/" + denom) : (numer + "");
    }
```

Summary

Abstract data types are characterized by their operations. Representation independence makes it possible to change the representation of a type without its clients being changed. An abstract data type that preserves its own invariants is easier and safer to use. Java language mechanisms like access control help ensure rep independence and invariants, but representation exposure is a trickier issue, and needs to be handled by careful programmer discipline.