Representation Invariants
6.170, Lecture 8
Fall 2005

8.1 Context

What you'll learn: How to find representation invariants and avoid representation exposure.

Why you should learn this: An understanding of the theory of abstract types helps you avoid whole classes of nasty, subtle bugs – or at minimum alerts you to their existence.

What I assume you already know: How to write a simple ADT.

In the next two lectures, we describe two tools for understanding abstract data types: the representation invariant and the abstraction function. The representation invariant describes whether an instance of a type is well formed; the abstraction function tells us how to interpret it.

8.2 A Tale of Two Spaces: Rep and Abstract

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. But 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 we wish to create an abstract type to represent a set of characters, with the following specification:

> *CharSet represents a set of characters*
> **class** CharSet
> spec (set **char**) cs;

CharSet()
  returns new result such that no result.cs

void add(**char** ch)
  modifies this.cs
  effects cs = \old(cs) + ch

void remove(**char** ch)
  modifies this.cs
  effects cs = \old(cs) - ch

boolean member(**char** ch)
  returns ch in cs

This specification uses a spec field `cs` with is a (mathematical) set of `char`s. Suppose we chose to implement this specification using a `StringBuffer` as follows:

```
public class CharSet {
    private StringBuffer s;
    public CharSet() {
        s = new StringBuffer();
    }
    public void add(char ch) {
        if (!member(ch)) s.append(ch);
    }
    public void remove(char ch) {
        int index = s.indexOf(String.valueOf(ch));
        if (index>=0) {
            s.deleteCharAt(index);
        }
    }
    public boolean member(char ch) {
        return s.indexOf(String.valueOf(ch))!=-1;
    }
}
```

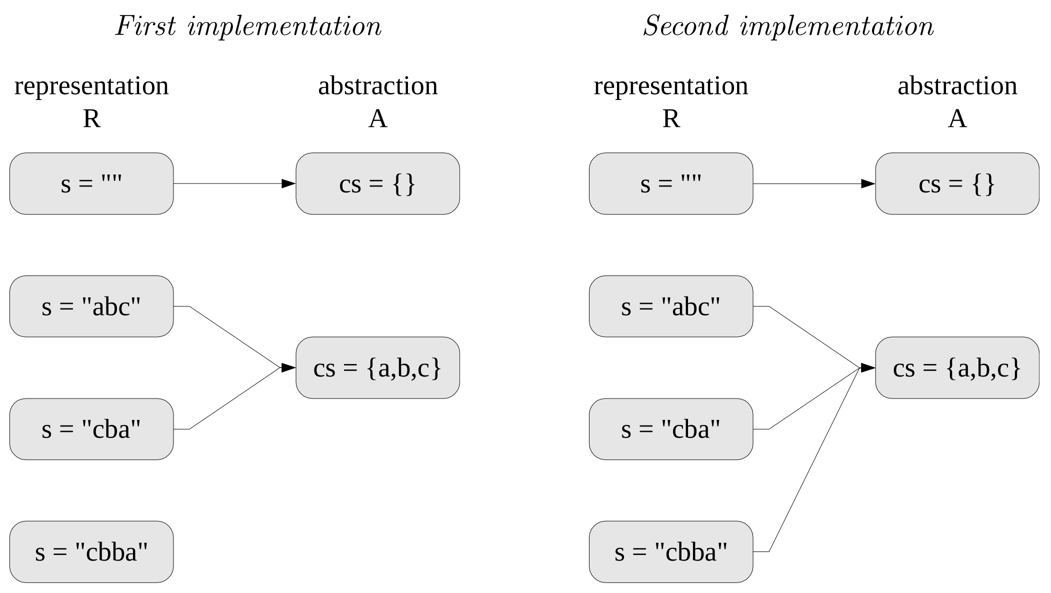
This works just fine. What are the rep values and the abstract values of this type? The abstract values are easy, and are given purely by the specification – they can be described by the possible values of the spec field `cs`: any set of characters, such as {} or {a,b,c}. The rep values are given by examining the implementation. They can be described by the possible values of the rep field `s` in the implementation. If you examine the code, you will discover that there are constraints on the possible values of `s`. The logic of the `add()` method is such that only one instance of each character can appear in `s`; "adb" is possible but not "adbd", for example. This is a useful fact, and vital to the correct functioning of the `remove()` method.

Contrast the following alternative implementation of `CharSet`:

```
public class CharSet {
    // everything else the same, except the following methods...
    public void add(char ch) {
        s.append(ch);
    }
    public void remove(char ch) {
        int index = s.indexOf(String.valueOf(ch));
        while (index>=0) {
            s.deleteCharAt(index);
            index = s.indexOf(String.valueOf(ch));
        }
    }
}
```

This works just fine too. But now the `add` method doesn't check for duplicates, so there is no constraint on `s`. The `remove` method needs to be changed to dovetail with the new `add` behavior.

We can look at the rep (R) and abstract (A) value spaces for the two implementations graphically, drawing arcs from each rep value to the abstract value it represents:



The graphs here are obviously not complete, and just show a few samples from infinite sets of values. There are several things to note about the graphs:

1. Every abstract value is mapped to from something. The purpose of implementing the abstract type is to support operations on abstract values. So we will need to be able to create and manipulate representations for all the possible abstract values we care about.

2. 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.

3. Not all rep values are mapped. In this case, the string "cbba" is not mapped for the first implementation. If the type of the rep field(s) are 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 the first implementation, the implementer decided that the string `s` should not contain duplicates. This made it possible to terminate the `remove` method when it hits the first instance of a particular character, since we know there can be at most one. This optimization wasn't possible in the second implementation.

8.3 Rep Invariants and Abstraction Functions

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 -> A
  ```

  The arcs in the previous diagram denote the abstraction function. In the terminology of functions, the properties 1-3 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 -> 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.

We will look at abstraction functions next day. Today, let's look at the rep invariant, `RI`. For our first implementation of `CharSet`, it was:

```
RI(r) = r.s != null &&
        r.s contains no duplicates
```

Like specifications, rep invariants can be expressed in various levels of (in)formality. For example, `r.s contains no duplicates` could be written as something like `i!=j => r.s[i]!=r.s[j]`, but it is probably clearer as is.

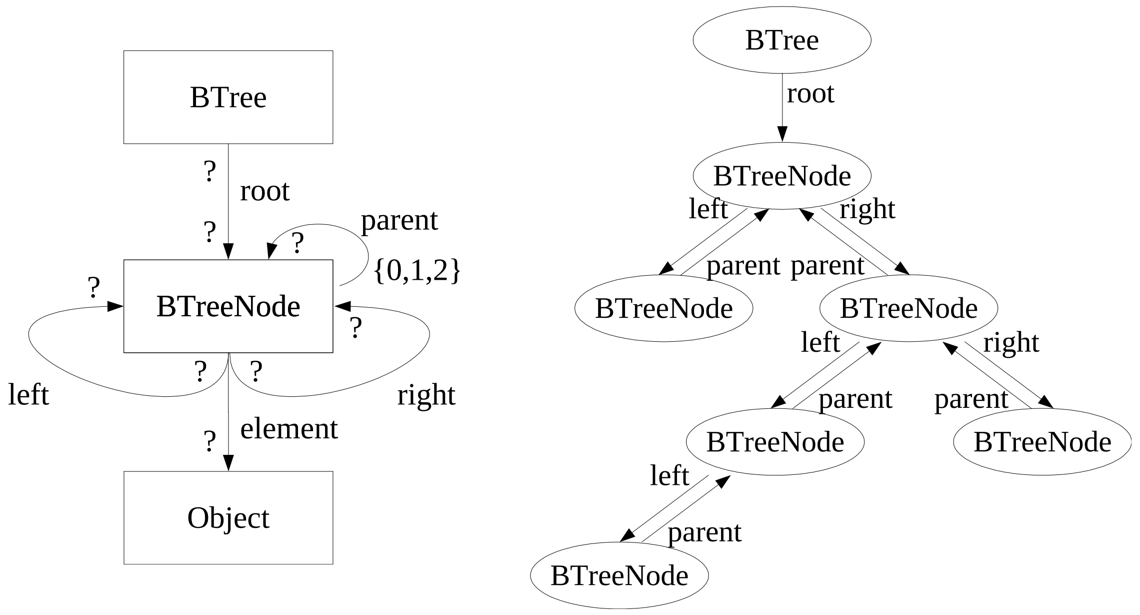
For our second implementation of `CharSet`, the rep invariant was weaker:

```
RI(r) = r.s != null
```

The rep invariant acts as a kind of *convention* for using the rep space. For example, once we decide that in our implementation of `CharSet` we will not allow duplicates in `r.s`, then we can write the `add()` and `remove()` method more independently. The abstraction function also helps with this, as we will see next day.

8.4 Another example

Consider a binary tree ADT, whose implementation has an object model as follows:



(object model on left, snapshot of a typical `BTree` on the right, omitting the `element` relation for clarity). We would like to write a rep invariante RI that is always true for our chosen representation:

```
all tree: BTree | RI(tree)
```

We can write RI recursively as:

```
RI(tree) = (tree.root!=null) =>
               (tree.root.parent = null && RI(tree,tree.root))

RI(tree,node) = ((node.left!=null) => (node.left!=node.right)) &&
                ((node.left!=null) =>
                    (node.left.parent = node && RI(tree,node.left))) &&
                ((node.right!=null) =>
                    (node.right.parent = node && RI(tree,node.right)))
```

This makes sure that the `parent`, `left` and `right` fields are consistent throughout the tree. Written this way, it is easy to imagine converting the `RI` to test code. You could also write the RI in many other ways; for example, the recursive construction is not necessary. The constraints in a tree are very similar to the file system example in a previous lecture. For example, to ensure that all nodes trace back to the root through their parent fields, we could write:

```
RI(tree) = all node: tree.root.*(left+right) | tree.root in n.*parent
           // ... other conditions, this is not complete ...
```

8.5 Rep Invariants for Modular Reasoning

The rep invariant makes *modular reasoning* possible. To check whether an operation is implemented in a way that is consistent with the rep invariant, we don't need to look at any other methods.

Instead, we appeal to the principle of induction. We ensure that every constructor creates an object that satisfies the invariant, and that every mutator and producer *preserves* the invariant: that is, if given an object that satisfies it, it produces one that also satisfies it. Now we can argue that every object of the type satisfies the rep invariant, since it must have been produced by a constructor and some sequence of mutator or producer applications.

Rep invariants can often be translated to code, and used to periodically check whether an object is in a valid state. A valid state is a rep value for which there is some corresponding abstract value – in other words, those rep values for which the rep invariant holds. If we detect that the rep invariant is false, then we know that something has gone wrong – the object could not possibly represent an abstract value. If the rep invariant is true, we know that the object does indeed represent an abstract value (though not necessarily the right one; this is not a panacea for detecting all bugs). Here is what we do:

- Create a method `checkRep()` which checks that the rep invariant holds (and fails immediately if it does not).
- Place calls to `checkRep()` at the end of every constructor, to confirm that the rep invariant holds when an object is first created.
- Place calls to `checkRep()` at the beginning and end of every public method: mutators, producers, and observers too (they may change the rep through *benevolent side-effects*, and anyway, do you really trust them?)

If we do all this, we can apply the following *structural induction*: if an invariant of an abstract data type is (1) established by creators; (2) preserved by producers, mutators, and observers; and (3) the representation of the type is never exposed, then the invariant is true of all instances of the abstract data type.

Notice the caveat about the representation being exposed (called *rep exposure*). How can this happen?

8.6 Rep Exposure

The notion of rep invariants that we have espoused offers a systematic method for checking the correctness of abstract data type implementations. (We haven't yet considered how to ensure that a creator, mutator or producer generates the right instance of a type, only that it produces a well-formed instance. When we study abstraction functions in the next lecture, we'll look at that issue.)

Our method says that we can consider the operations one by one, and then appeal to induction to show that every instance will be well-formed. A crucial aspect of this method is local reasoning: we can examine the operations individually, and certainly don't need to look at client code.

But this method is not always sound. It has a proviso: that the representation must not be exposed. Representation exposure is a nasty problem, because it can arise unexpectedly, and have disastrous effects that are hard to pin down.

The simplest form of rep exposure involves allowing client code to manipulate the representation directly. We saw that this is easy to rule out by making all fields private, so we don't usually even regard it as a kind of exposure. Instead, the rep exposure we'll be concerned with arises because a mutable object inside the representation is accessible from the outside, through a different path. Two common ways in which this happens are:

- when a reference to a mutable object is passed in and made part of the rep, despite being accessible as an existing reference from the outside.
- when a reference to a mutable object in the rep is returned as the result of an operation.

Consider the following class representing an interval of time between two `Date`s:

```
public class Interval {
    private final Date start, stop;
    private final long duration;
    public Interval(Date start, Date stop) {
        this.start = start;
        this.stop = stop;
        duration = stop.getTime()-start.getTime();
    }
    public Date getStart() { return start; }
    public Date getStop() { return stop; }
    public long getDuration() { return duration; }
}
```

The rep invariant will capture the need for consistency between the duration, start, and stop fields. But unfortunately this implementation leaks like a sieve because the Java `Date` class is mutable, and it is easy for a client to end up having a reference to the `Date` objects stored in `start` and `stop` – and hence being free to change these without updating duration.

To avoid this rep exposure, we need to make *defensive copies*, using copy constructors, clone methods, or whatever is available:

```
public class Interval {
    //...
    public Interval(Date start, Date stop) {
        this.start = new Date(start.getTime());
        this.stop = new Date(stop.getTime());
        duration = stop.getTime() - start.getTime();
    }
    public Date getStart() { return new Date(start.getTime()); }
    public Date getStop() { return new Date(stop.getTime()); }
}
```

Another common example of rep exposure can occur when a method returns a collection. When the representation already contains a collection object of the appropriate type, it is quite tempting to return it directly. For example, in Java `List`s must have a method `toArray()` that returns an array of elements corresponding to the elements of the list. Suppose we implement a class `VeryExclusiveList` which only accepts `HighSociety` objects. If in our implementation we use an array, we might be tempted to just return this array when implementing the `toArray()` method. But this leaves us wide open:

```
List posh = new VeryExclusiveList();
posh.add(new HighSociety("Lord Vandersnoot"));
posh.add(new HighSociety("Lady Bassington-Bassington"));
//...
Object[] whosWho = posh.toArray();
whosWho[0] = new Ruffian();  // ouch! who left the door open...
```

A more subtle variant of this problem arises with iterators. Many Java classes have a method that returns an iterator. It is often a good design decision to provide a method that returns an iterator over a collection, rather than the collection itself, to save the cost of making defensive copies. But building an iterator is a lot of work for the implementor, so we might be tempted to use one that's already provided by the Java library. Suppose our representation includes an `ArrayList` field called `list` that holds a collection of elements, and we want to implement a method:

```
public Iterator elements()
```

that returns an iterator over those elements. Noticing that the `ArrayList` interface provides its own method that returns an iterator, we implement our method like this:

```
public Iterator elements() {
    return list.iterator();
}
```

Unfortunately, this too is a rep exposure! The `Iterator` interface in Java includes an optional `remove` operation that allows the client to remove elements from the underlying collection. `ArrayList` implements this operation, so the result of this method is an iterator object that can actually affect the list collection, outside the abstract type.

Fortunately there's a solution to this problem:

```
public Iterator elements() {
    return Collections.unmodifiableList(list).iterator();
}
```

The `unmodifiableList` method in `Collections` creates a wrapper around the list you give it, which forbids all mutations through the wrapper.

8.7 Summary

Why use rep invariants? Recording the invariant can actually save work:

- It makes modular reasoning possible. Without the rep invariant documented, you might have to read all the methods to understand what's going on before you can confidently add a new method.
- It helps catch errors. By implementing the invariant as a runtime assertion, you can find bugs that are hard to track down by other means.