Abstract Data Types

6.170 Lecture 6

Spring 2004

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.

1 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.

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 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.

2 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. `StringBuffer`, 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; the terms are synonymous. 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 `ArrayList`, 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 `ArrayList`, for example, returns an integer.

We can summarize these distinctions schematically like this:

constructor: $t^* \to T$
producer: $T^+, t^* \to T$
mutator: $T^+, t^* \to$ void
observer: $T^+, t^* \to t$

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)

`CardSuit` (from PS1) represents the set of card suits: clubs, diamonds, hearts, spades. This type is not only immutable, it is *finite* — in fact, since it uses the *type-safe enumeration pattern*, only 4 `CardSuit` objects ever exist in the program. See the Bloch book for more details about typesafe enumerations, a very nice way to implement small finite types.

creators: the constants `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES`.
producers: none
observers: `getName`, `compareTo`, `toString`, etc.
mutators: none (immutable)

`Card` (from PS1) represents the set of playing cards. This type is also finite, but instead of having 52 constants, one for each card, the class has a constructor instead. This means that you can create two different Java objects representing the Ace of Spades, but both objects are the same abstract value as far as the abstract data type is concerned. This is why you should always use `equals` to compare immutable objects, not ==.

creators: `Card(CardValue, CardSuit)` constructor
producers: none
observers: `getSuit`, `getValue`, `equals`, `compareTo`, `toString`, etc.
mutators: none (immutable)

`Deck` (from PS1) represents the set of decks of cards. This type is mutable — which means it's also *infinite*. Although it's true that there are only a finite number of deck configurations (e.g., 52! orderings of full decks), you can tell two decks apart by mutating one of them. So mutable data types are theoretically infinite (but practically subject to the constraints of memory).

creators: `Deck()` constructor
producers: none
observers: `listCards`, `toString`, etc.
mutators: `addCard`, `removeCard`, `shuffle`, `dealCards`. Notice that `dealCards` is a mutator that returns something other than `void`. It returns an `Iterator` of the cards that were dealt off the deck. So the type signatures above aren't hard-and-fast rules. What really defines a mutator is whether the operation's spec includes *modifies this*.

`BasicList` (from PS2) is a slimmed-down version of Java's `List` data type, representing a mutable sequence of objects.
creators: `BasicList()` constructor
producers: none
observers: `contains`, `get`, `indexOf`, `size`
mutators: `add`, `remove`

This classification gives some useful terminology, but it's not perfect. In complicated data types, there may be operations that are producers and mutators, for example. Some people use the term *producer* to imply that no mutation occurs.

3 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 `BasicList`, for example. It might help clients who work with lists of `Integers`, but what about lists of `Strings`? `Cards`? 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, because we could apply `get` on increasing indices, 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. It wouldn't make sense to put a domain-specific method like `dealCards` into the generic type `BasicList`, for example. Conversely, `Deck` shouldn't have an `add` method that accepts arbitrary `Objects`.

- 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 `BasicList` 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.

4 Rep Exposure

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.

We'll see many interesting invariants next lecture. Let's focus on immutability for now. As a specific example, here's a class we've seen in an earlier lecture:

```
// Trans represents an immutable bank transaction.
class Trans {
    int amount;
    Date date;

    Trans(int amt, Date d) {
     // effects: makes a transaction of amt dollars on date d
        amount = amt;
        date = d;
    }
}
```

How do we guarantee that `Trans` 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:

```
Trans t = new Trans(10, new Date ());
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 `Trans` 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:

```
// Trans represents an immutable bank transaction.
class Trans {
    private int amount;
    private Date date;

    public Trans(int amt, Date d) {
     // effects: makes a transaction of amt dollars on date d
        amount = amt;
        date = d;
    }

    public int getAmount {
     // effects: returns amount of this transaction in dollars
        return amount;
    }

    public Date getDate {
     // effects: returns date of this transaction
        return date;
    }
}
```

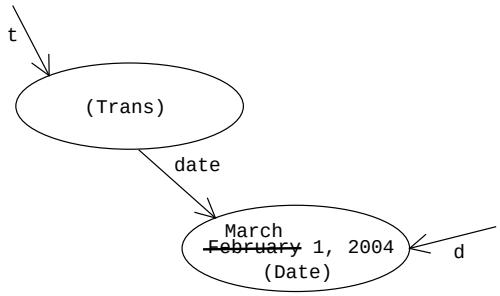
The private and public keywords indicate which fields and methods are accessible only within the class and which can be accessed from outside the class.

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

```
1.  Trans makeNextPayment(Trans t) {
2.   // effects: returns a transaction of same amount as t, one month later
3.      Date d = t.getDate();
4.      d.setMonth (d.getMonth() + 1);
5.      // ... deal with December and day 28/30/31 problem ...
6.      return new Trans (t.getAmount(), d);
7.  }
```

We've left some code out for simplicity, but the idea here is that `makeNextPayment` takes a transaction and should return another transaction for the same amount but dated a month later. (The code we left out concerns rolling December over to January, and dealing with days that don't exist in the next month, like January 31.) The `makeNextPayment` method might be part of a system that schedules recurring payments.

What's the problem here? The `getDate` call in Statement 3 returns a reference to the *same* date object referenced by transaction `t`. So when the date object is mutated in Statement 4, this affects the date in `t` as well:



`Trans`'s immutability invariant has been broken. The problem is that `Trans` leaked out a reference to a mutable object that its invariant depended on. We *exposed the rep*, in such a way that `Trans` 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:

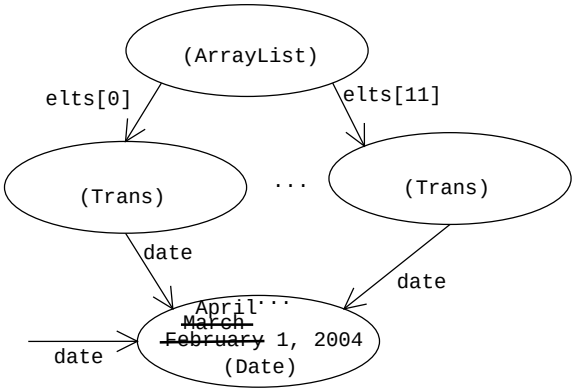
```
// Trans represents an immutable bank transaction.
class Trans {
    private int amount;
    private Date date;

    public Trans(int amt, Date d) {
     // effects: makes a transaction of amt dollars on date d
        amount = amt;
        date = d;
    }

    public int getAmount {
     // effects: returns amount of this transaction in dollars
        return amount;
    }

    public Date getDate {
     // effects: returns date of this transaction
        return new Date (date);
    }
}
```

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

```
1.  List makeYearOfPayments (int amount) {
```

```
2.   // effects: returns a list of 12 monthly payments of identical amounts
3.      List list = new ArrayList ();
4.      Date date = new Date ();
5.      for (int i = 0; i < 12; i++) {
6.          list.add (new Trans (amount, date));
7.          date.setMonth (date.getMonth () + 1);
8.          // ... deal with December and day 28/30/31 problem ...
9.      }
10.     return list;
11. }
```

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



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

```
// Trans represents an immutable bank transaction.
class Trans {
    private int amount;
    private Date date;

    public Trans(int amt, Date d) {
     // effects: makes a transaction of amt dollars on date d
        amount = amt;
        date = new Date (d);
    }

    public int getAmount {
     // effects: returns amount of this transaction in dollars
        return amount;
    }

    public Date getDate {
     // effects: returns date of this transaction
        return new Date (date);
```

```
    }
}
```

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:

```
public Trans (int amount, Date date) {
 // requires: caller must never mutate date again!
 // effects: returns date of this transaction
    return date;
}
...
public Date getDate {
 // effects: returns date of this transaction
 // requires: caller must never mutate the date returned!
    return 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 `Date` had been immutable, then we would have ended this section after talking about `public` and `private`. No rep exposure would have been possible.

5 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.