1 Introduction

In this lecture, we'll look at the role played by specifications of methods. Specifications are the linchpin of teamwork. It's impossible to delegate responsibility for implementing a method without a specification. The specification acts as a contract: the implementor is responsible for meeting the contract, and a client that uses the method can rely on the contract. In fact, we'll see that like real legal contracts, specifications place demands on both parties: when the specification has a precondition, the client has responsibilities too.

In the second part of this lecture we will look at a particular kind of dependence, that of a client of an abstract type on the type's representation, and see how it can be avoided. We also discuss briefly the notion of specification fields for specifying abstract types, the classification of operations, and the tradeoff of representations.

2 Why Specifications?

Many of the nastiest bugs in programs arise because of misunderstandings about behavior at interfaces. Although every programmer has specifications in mind, not all programmers write them down. As a result, different programmers on a team have different specifications in mind. When the program fails, it's hard to determine where the error is. Precise specifications in the code let you apportion blame (to code fragments, not people!), and can spare you the agony of puzzling over where a fix should go.

Specifications are good for the client of a method because they spare her the task of reading code. If you're not convinced that reading a spec is easier than reading code, take a look at some of the standard Java specs and compare them to the source code that implements them. Vector, for example, in the package java.util, has a very simple spec but its code is not at all simple.

Specifications are good for the implementor of a method because they give her freedom to change the implementation without telling clients. Specifications can make code faster too. Sometimes a weak specification makes it possible to do a much more efficient implementation. In particular, a precondition may rule out certain states in which a method might have been invoked that would have incurred an expensive check that is no longer necessary.

The contract acts as a firewall between client and implementor. It shields the client from the details of the workings of the unit -- you don't need to read the source code of the procedure if you have its specification. And it shields the implementor from the details of the usage of the unit; he doesn't have to ask every client how she plans to use the unit. This firewall results in decoupling, allowing the code of the unit and the code of a client to be changed independently, so long as the changes respect the specification -- each obeying its obligation.

Specifications actually play two subtly different roles in software. One is to catalog reusable components: this is the purpose of the specifications in the Java collections
framework, for example. The other is to regulate the connections between modules in a design. In this role, a single unit may have multiple specifications, one for each client. The specifications qualify the 'uses' relationship between modules, saying exactly how one module uses another. As the system evolves, these specifications are the part that is least affected. Consequently, perhaps more than anything else, these specifications characterize the design of the software -- they are the design. When we study design patterns, we'll see how the motivation of most design patterns is to improve the decoupling of modules, and this is usually achieved by introducing new specifications, which are weaker than the specifications used in simpler designs.

3 Behavioral Equivalence

Consider these two methods. Are they the same or different?

```java
static int findA (int [] a, int val) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == val) return i;
    }
    return a.length;
}

static int findB (int [] a, int val) {
    for (int i = a.length -1 ; i >= 0; i--) {
        if (a[i] == val) return i;
    }
    return -1;
}
```

Of course the code is different, so in that sense they are different. Our question though is whether one could substitute one implementation for the other. Not only do these methods have different code; they actually have different behavior:
- when `val` is missing, `findA` returns the length and `findB` returns -1;
- when `val` appears twice, `findA` returns the lower index and `findB` returns the higher.

But when `val` occurs at exactly one index of the array, the two methods behave the same. It may be that clients never rely on the behavior in the other cases. So the notion of equivalence is in the eye of the beholder, that is, the client. In order to make it possible to substitute one implementation for another, and to know when this is acceptable, we need a specification that states exactly what the client depends on.

In this case, our specification might be

```plaintext
requires:   val occurs in a
effects:    returns result such that a[result] = val
```

4 Specification Structure

A specification of a method consists of several clauses:

- a precondition, indicated by the keyword `requires`;
- a postcondition, indicated by the keyword `effects`;
- a frame condition, indicated by the keyword `modifies`.  

2
The precondition is an obligation on the client (i.e., the caller of the method). It's a condition over the state in which the method is invoked. If the precondition does not hold, the implementation of the method is free to do anything (including not terminating, throwing an exception, returning arbitrary results, making arbitrary modifications, etc).

The postcondition is an obligation on the implementor of the method. If the precondition holds for the invoking state, the method is obliged to obey the postcondition, by returning appropriate values, throwing specified exceptions, modifying or not modifying objects, and so on.

The frame condition is related to the postcondition. It allows more succinct specifications. Without a frame condition, it would be necessary to describe how all the reachable objects may or may not change. But usually only some small part of the state is modified. The frame condition identifies which objects may be modified. If we say modifies x, this means that the object x, which is presumed to be mutable, may be modified, but no other object may be. So in fact, the frame condition or modifies clause as it is sometimes called is really an assertion about the objects that are not mentioned. For the ones that are mentioned, a mutation is possible but not necessary; for the ones that are not mentioned, a mutation may not occur.

Omitted clauses have particular interpretations. If you omit the precondition, it is given the default value true. That means that every invoking state satisfies it, so there is no obligation on the caller. In this case, the method is said to be total. If the precondition is not true, the method is said to be partial, since it only works on some states.

If you omit the frame condition, the default is modifies nothing. In other words, the method makes no changes to any object.

Omitting the postcondition makes no sense and is never done.

### 5 Find Revisited

Roughly speaking, there are two kinds of specifications. Here is one possible specification of find:

```java
static int find (int [] a, int val)
  requires: val occurs exactly once in a
  effects: returns result such that a[result] = val
```

This specification is deterministic: when presented with a state satisfying the precondition, the outcome is determined. Both findA and findB satisfy the specification, so if this is the specification on which the clients relied, the two are equivalent and substitutable for one another. (Of course a procedure must have the name demanded by the specification; here we are using different names to allow us to talk about the two versions. To use either, you'd have to change its name to find.)

Here is a slightly different specification:

```java
static int find (int [] a, int val)
  requires: val occurs in a
  effects: returns result such that a[result] = val
```
This specification is not deterministic. Such a specification is often said to be non-deterministic, but this is a bit misleading. Non-deterministic code is code that you expect to sometimes behave one way and sometimes another. This can happen, for example, with concurrency: the scheduler chooses to run threads in different orders depending on conditions outside the program.

But a 'non-deterministic' specification doesn't call for such non-determinism in the code. The behavior specified is not non-deterministic but under-determined. In this case, the specification doesn't say which index is returned if val occurs more than once; it simply says that if you look up the entry at the index given by the returned value, you'll find val.

This specification is again satisfied by both findA and findB, each 'resolving' the underdeterminedness in its own way. A client of find can't predict which index will be returned, but should not expect the behavior to be truly non-deterministic. Of course, the specification is satisfied by a non-deterministic procedure too -- for example, one that rather improbably tosses a coin to decide whether to start searching from the top or the bottom of the array. But in almost all cases we'll encounter, non-determinism in specifications offers a choice that is made by the implementor at implementation time, and not at runtime.

So, as before, for this specification too, the two versions of find are equivalent. Finally, here's a specification that distinguishes the two

```java
static int find (int [] a, int val)
    effects: returns largest result such that
              a[result] = val or -1 if no such result
```

It is satisfied by findB but not findA.

6 Specification for a Mutating Method

Our specifications of find didn't give us the opportunity to illustrate frame conditions and the description of side effects.

Here's a specification that describes a method that mutates an object:

```java
class Vector {
    ...
    boolean addAll (Vector v)
        requires: v != null and v != this
        modifies: this
        effects: adds the elements of v to the end of this, and returns true if this changed as a result of call
}
```

We've taken this, slightly simplified, from the Java Vector class. First, look at the frame condition: it tells us that only this is modified, so in particular the argument vector v is not mutated -- likely to be a crucial property for most clients. Second, look at the
postcondition. It gives two constraints: the first telling us how this is modified, and the second telling us how the return value is determined. Finally, look at the precondition. It tells us that the behavior of the method is not constrained if you call it with a null argument, or if you attempt to add the elements of a vector to itself. You can easily imagine why the implementor of the method would want to impose the second constraint: it’s not likely to rule out any useful applications of the method, and it makes it easier to implement. The specification allows a simple implementation in which you take an element from v and add it to this, then go on to the next element of v until you get to the end. If v and this are the same vector, this algorithm will not terminate -- an outcome permitted by the specification.

7 Declarative Specification

Roughly speaking, there are two kinds of specifications. Operational specifications give a series of steps that the method performs; pseudocode descriptions are operational. Declarative specifications don’t give details of intermediate steps. Instead, they just give properties of the final outcome, and how it’s related to the initial state.

Almost always, declarative specifications are preferable. They’re usually shorter, easier to understand, and most importantly, they don’t expose implementation details inadvertently that a client may rely on (and then find no longer hold when the implementation is changed). For example, if we want to allow either implementation of find, we would not want to say in the spec that the method ‘goes down the array until it finds val’, since aside from being rather vague, this spec suggests that the search proceeds from lower to higher indices and that the lowest will be returned, which perhaps the specifier did not intend.

Here are some examples of declarative specification, starting with one from String. The startsWith method tests whether a string starts with a particular substring:

```java
public boolean startsWith(String prefix)
    effects:
        if (prefix == null) throws NullPointerException
        else returns true iff there exists a sequence s such that (prefix.seq ^ s = this.seq)
```

We have assumed that String objects have a specification field that models the sequence of characters. The caret is the concatenation operator, so the postcondition says that the method returns true if there is some suffix which, when concatenated to the argument, gives the character sequence of the string. The absence of a modifies clause indicates that no object is mutated. Since String is an immutable type, none of its methods will have modifies clauses.

Another example from String:

```java
public String substring(int i)
    effects:
        if i < 0 or i > length (this.seq) throws IndexOutOfBoundsException else returns r such that some sequence s | length(s) = i && s ^ r.seq = this.seq
```

5
This specification shows how a rather mathematical postcondition can sometimes be easier to understand than an informal description. Rather than talking about whether \( i \) is the starting index, whether it comes just before the substring returned, etc, we simply decompose the string into a prefix of length \( i \) and the returned string.

## 8 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 precursor to 6.005!).

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.

## 9 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 *Vector* is mutable, because you can call `add` and observe the change with the `size` 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:
· **Constructors** create new objects of the type. A constructor may take an object as an argument.

· **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 `Vector`, for example, mutates a vector by adding an element to its high end.

· **Observers** take objects of the abstract type and return objects of a different type. The `size` method of `Vector`, for example, returns an integer.

This classification gives some useful terminology, but it’s not perfect. In complex 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.

Another term you should know is **iterator**. An iterator usually means a special kind of method (not available in Java) that returns a collection of objects one at a time -- the elements of a set, for example. In Java, an iterator is a **class** that provides methods that can then be used to obtain a collection of objects one at a time. Most collection classes provide a method with the name `iterator` that returns an iterator.

### 10 Example: List

Let’s look at an example of an abstract type: the list. A list, in Java, is like an array. It provides methods to extract the element at a particular index, and to replace the element at a particular index. But unlike an array, it also has methods to insert or remove an element at a particular index. In Java, **List** is an interface with many methods, but for now, let’s imagine it’s a simple class with the following methods:

```java
class List {
    public List ();
    public void add (int i, Object e);
    public void set (int i, Object e);
    public void remove (int i);
    public int size ();
    public Object get (int i);
}
```

The `add`, `set`, and `remove` methods are mutators; the `size` and `get` methods are observers. It’s common for a mutable type to have no producers (and an immutable type certainly cannot have mutators).

To specify these methods, we’ll need some way to talk about what a list looks like. We do this with the notion of **specification fields**. You can think of an object of the type as if it had these fields, but remember that they don’t actually need to be fields in the implementation, and there is no requirement that a specification field’s value be obtainable by some method. In this case, we’ll describe lists with a single specification field,

```java
    seq [Object]    elesms;
```

where for a list `l`, the expression `l.elems` will denote the sequence of objects stored in the list, indexed from zero. Now we can specify some methods:

```java
    public void get (int i);
```
 effects: throws IndexOutOfBoundsException, if i < 0 or i >
       length (this.elems), and returns this.elems [i]
public void add (int i, Object e);
 modiﬁes: this
 effects: throws IndexOutOfBoundsException if i < 0 or i >
       length (this.elems), else this.elems’ =
       this.elems [0 .. i - 1] ^ <e> ^ this.elems [i ..]
public void set (int i, Object e);
 modiﬁes: this
 effects: throws IndexOutOfBoundsException if i < 0 or i >=
       length (this.elems), else this.elems’ [i] =
       e and this.elems unchanged elsewhere

In the postcondition of add, we have used s[i..j] to mean the subsequence of s from
indices i to j, and s[i..] to mean the suﬃx from i onwards. The caret means
sequence concatenation. So the postcondition says that, when the index is in bounds or
one above, the new element is ‘spliced in’ at the given index.

11 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.
· 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 the 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.

12 Choice of Representations

So far, we have focused on the characterization of abstract types by their operations. In
the code, a class that implements an abstract type provides a representation: the actual
data structure that supports the operations. The representation will be a collection of
fields each of which has some other Java type; in a recursive implementation, a field
may have the abstract type but this is rarely done in Java.

Linked lists are a common representation of lists, for example. The list object has a ﬁeld
header that references an Entry object. An Entry object is a record with three ﬁelds:
next and prev which may hold references to other Entry objects (or be null), and
element, which holds a reference to an element object. The next and prev ﬁelds are
links that point forwards and backwards along the list. In the middle of the list, following
next and then prev will bring you back to the object you started with. Let’s assume that the linked list does not store null references as elements. There is always a dummy Entry at the beginning of the list whose element field is null, but this is not interpreted as an element.

Another, different representation of lists uses an array. The list has a field elementData that is an array of elements.

These representations have different merits. The linked list representation will be more efficient when there are many insertions at the front of the list, since it can splice an element in and just change a couple of pointers. The array representation has to bubble all the elements above the inserted element to the top, and if the array is too small, it may need to allocate a fresh, larger array and copy all the references over. If there are many get and set operations, however, the array list representation is better, since it provides random access, in constant time, while the linked list has to perform a sequential search.

We may not know when we write code that uses lists which operations are going to predominate. The crucial question, then, is how we can ensure that it’s easy to change representations later.

13 Representation Independence

Representation independence means ensuring that the use of an abstract type is independent of its representation, so that changes in representation have no effect on code outside the abstract type itself. Let’s examine what goes wrong if there is no independence, and then look at some language mechanisms for helping ensure it.

Suppose we know that our list is implemented as an array of elements. We’re trying to make use of some code that creates a sequence of objects, but unfortunately, it creates a Vector and not a List. Our List type doesn’t offer a constructor that does the conversion. We discover that Vector has a method copyInto that copies the elements of the vector into an array. Here’s what we now write:

```java
List l = new List();
v.copyInto(l.elementData);
```

What a clever hack! Like many hacks it works for a little while. Suppose the implementor of the List class now changes the representation from the array version to the linked list version. Now the list l won’t have a field elementData at all, and the compiler will reject the program. This is a failure of representation independence: we’ll have to change all the places in the code where we did this.

Having the compilation fail is not such a disaster. It’s much worse if it succeeds and the change has still broken the program. Here’s how this might happen.

In general, the size of the array will have to be greater than the number of elements in the list, since otherwise it would be necessary to create a fresh array every time an element is added or removed. So there must be some way of marking the end of the segment of the array containing the elements. Suppose the implementor of the list has designed it with the convention that the segment runs to the first null reference, or to the end of the array, whichever is first. Luckily (or actually unluckily), our hack works under these circumstances.
Now our implementor realizes that this was a bad decision, since determining the size of the list requires a linear search to find the first null reference. So he adds a size field and updates it when any operation is performed that changes the list. This is much better, because finding the size now takes constant time. It also naturally handles null references as list elements (and that’s why it’s what the Java LinkedList implementation does).

Now our clever hack is likely to produce some buggy behaviors whose cause is hard to track down. The list we created has a bad size field: it will hold zero however many elements there are in the list (since we updated the array alone). Get and set operations will appear to work, but the first call to size will fail mysteriously.

Here’s another example. Suppose we have the linked list implementation, and we include an operation that returns the Entry object corresponding to a particular index.

```java
public Entry getEntry (int i)
```

Our rationale is that if there are many calls to set on the same index, this will save the linear search of repeatedly obtaining the element. Instead of

```java
l.set (i, x); ... ; l.set (i, y)
```

we can now write

```java
Entry e = l.getEntry (i);
e.element = x;
...
e.element = y;
```

This also violates representation independence, because when we switch to the array representation, there will no longer be Entry objects.

There should only be a dependence of the client type Client on the List class (and on the class of the element type, in this case Object, of course). The dependence of Client on Entry is the cause of our problems. Returning to our object model for this representation, we want to view the Entry class and its associations as internal to List. We can indicate this informally by coloring the parts that should be inaccessible to a client red (if you’re reading a black and white printout, that’s Entry and all its incoming and outgoing arcs), and by adding a specification field elems that hides the representation:

In the Entry example we have exposed the representation. A more plausible exposure, which is quite common, arises from implementing a method that returns a collection. When the representation already contains a collection object of the appropriate type, it is tempting to return it directly. For example, suppose that List has a method toArray that returns an array of elements corresponding to the elements of the list. If we had implemented the list itself as an array, we might just return the array itself. If the size field was based on the index at which a null reference first appears) a modification to this array may break the computation of size.

```java
a = l.toArray ();    // exposes the rep
a[i] = null;        //ouch!!
...
elm = l.get (i);    // now behaves unpredictably
```
Once size is computed wrongly, all hell breaks loose: subsequent operations may behave in arbitrary ways.

14 Language Mechanisms

To prevent access to the representation, we can make the fields private. This eliminates the array hack; the statement

```java
v.copyInto (l.elementData);
```

would be rejected by the compiler because the expression `l.elementData` would illegally reference a private field from outside its class.

The `Entry` problem is not so easily solved. There is no direct access to the representation. Instead, the `List` class returns an `Entry` object that belongs to the representation. This is called `representation exposure`, and it cannot be prevented by language mechanisms alone. We need to check that references to mutable components of the representation are not passed out to clients, and that the representation is not built from mutable objects that are passed in. In the `array` representation for example, we can't allow a constructor that takes an array and assigns it to the internal field.

Interfaces provide another method for achieving representation independence. In the Java standard library, the two representations of lists that we discussed are actually distinct classes, `ArrayList` and `LinkedList`. Both are declared to extend the `List` interface. `List` declares only the operations and doesn't give representations or code:

```java
public interface List {
    void add (int i, Object e);
    void set (int i, Object e);
    void remove (int i);
    int size ();
    Object get (int i);
}
```

and the two classes are declared as implementations of `List`:

```java
public class LinkedList implements List {
    private Entry header;
    ...
}

public class ArrayList implements List {
    private Object[] eltData;
    ...
}
```
Whenever possible, clients refer only to the List interface, so the classes containing the representations are not accessible. Here’s a standard idiom for creating and manipulating objects:

```java
List l = new LinkedList();
...
l.add(i, e);
```

Note that the interface can’t be used to construct the object; an interface has no constructors, and it is at the point of creation that we need to specify the implementation. But we have carefully declared the result of the constructor call as a List and not a LinkedList. A subsequent reference to l.header would now be illegal, even if the field were declared public.

The dependences on the concrete classes due to constructor calls are localized as much as possible, but sometimes we would like to mitigate them further. The Factory design pattern, which we will discuss later in the course, addresses this particular problem.

### 15 Summary

A specification acts as a crucial firewall between the implementor of a procedure and its client. It makes separate development possible: the client is free to write code that uses the procedure without seeing its source code, and the implementor is free to write the code that implements the procedure without knowing how it will be used. Declarative specifications are the most useful in practice. Preconditions make life hard for the client, but, applied judiciously, are a vital tool in the software designer’s repertoire.

Abstract types are characterized by their operations. Representation independence makes it possible to change the representation of a type without its clients being changed. In Java, access control mechanisms and interfaces can help ensure independence. Representation exposure is trickier though, and needs to be handled by careful programmer discipline.