Specification and validation

[LG Ch. 9] Design patterns are a useful way to describe program structure. They provide a guide as to **how** a program fits together.

Another dimension is the responsibilities of the classes that make up the program. We should be able to specify these responsibilities and enforce them.

*Design by contract* views the relationship between classes and their clients as a formal agreement.

Each party’s rights and obligations are spelled out.

Violations of the contract are *exceptions*, and are usually handled by special language constructs.

**Software correctness**

How can we know that software is correct?

It is only correct if it does what it is supposed to do.

But how do we know what it is supposed to do? We need a *specification*.

For example, is

\[ x = y + 5 \]

a correct statement? You need to know what the statement is supposed to do.

If the specification is,
“Make sure \( x \) and \( y \) have different values,”

then the statement is

But if the specification is, “Make sure that \( x \) has a negative value,”

then the statement is

Specifications are expressed through assertions. Not only do assertions help us determine whether the program is correct, they also—

- help document the program, and
- provide a basis for systematic testing and debugging.

A correctness formula is an expression

\[
\{P\} S \{Q\}
\]

It means that any execution of \( S \), starting in a state where \( P \) holds, will terminate in a state where \( Q \) holds.

In this formula,

- \( P \) is called the precondition, and
- \( Q \) is called the postcondition.

This sounds simple, but the implications take some getting used to.

Here is a trivial correctness formula, which holds as long as \( x \) is an integer:

\[
\{x \geq 9\} x = x + 5 \{x \geq 13\}
\]

It looks like this is a typo. After all, we could also strengthen the postcondition to read,
This postcondition is *stronger* than the original postcondition, because there are values of \( x \) which satisfy the original postcondition but do not satisfy it.

Given the original precondition, this is the strongest postcondition we can give and still have the program be correct.

Similarly, given the original postcondition, we can also weaken the precondition. What is the weakest precondition we can give involving \( x \) to have the program be correct?

From a formula that holds, you can always get another one that holds by strengthening the ____________ or weakening the ____________.

Preconditions and postconditions can be introduced into programs by using *assertions*.

**Specifications**

In this lecture, we will list preconditions and postconditions as *requirements* and *effects* of a program.

Here’s an example from the text.

```java
static int p(int y)
// Requires: y > 0
// Effects: Returns x such that x > y
```

The specification is satisfied by any procedure named \( p \) that, when _________________, returns a value greater than its argument.

What are some programs satisfying this specification?
The collection of programs satisfying a specification is called its *specificand set*.

A specification should be both—

- *sufficiently restrictive*, so that it rules out all implementations that it rules out all implementations that are unacceptable to an abstraction’s clients, and
- *sufficiently general*, so that it does not preclude acceptable implementations.

**Sufficiently restrictive**

One way in which a specification can fail to be restrictive enough is if some requirements are left out.

L&G give the example of a specification for a Java class that is like Smalltalk Bags.

Let us take a look at an *iterator* over this collection.

The *Iterator* interface is defined in *java.util*.

It is like *Enumeration*, except that it allows elements to be removed from the collection during the iteration.

Here is a specification for an *elems* iterator for a bag of integers.

```java
public Iterator elems()
// Effects:  Returns a generator
// that produces every element of this
// (as Integers).
```

This specification does not say what happens if the bag is changed while the iteration is in progress.

So implementations of it could exhibit quite diverse behavior.

If we want to constrain the implementation to a particular behavior, we might require that the bag not be changed while the generator is in use.
This yields the following specification.

```java
public Iterator elems()
// Effects: Returns a generator
// that produces every element of this
// (as Integers).
// Requires: this not be modified while
// the generator is in use.
```

There are still at least two potential sources of confusion here.

- It does not specify the order in which the elements are to be returned. Should it?

- It does not say what to do when an element occurs

Here is a revised specification that addresses these issues.

```java
public Iterator elems()
// Effects: Returns a generator
// that produces every element of this
// (as Integers), in arbitrary order.
// Each element is produced exactly the
// number of times it occurs in this.
// Requires: this not be modified while
// the generator is in use.
```

A good specification should—

- specify exceptions that are to be raised, and
- say what happens in boundary cases.

Consider this specification for `indexString`:

```java
public static int indexString(String s1, String s2)
// Effects: If s1 occurs as a substring in s2, returns the index
// which s1 occurs; else returns -1.
// Examples:
// indexString("bc", "abcabc") = 1
// indexString("b", "a") = -1
```
What does this *not* tell us?

Let’s correct these omissions.

```java
public static int indexString(String s1, String s2)
    // Effects:
    // if s1 occurs as a substring in s2 returns the ___ index at which
    // s1 occurs; else returns -1.
    // Examples:
    // indexString("bc", "abcabc") = 1
    // indexString("b", "a") = -1
```

*Sufficiently general*

Why is it important for specifications to be as general as possible?

Here is a specification for a search function.

```java
public static int search(int[] a, int x)
    throws NotFoundException, NullPointerException
    // Effects: If a is null, throws NullPointerException; else
    // examines a[0], a[1], ... in turn. If one of the elements
    // is equal to x before a zero value is reached, the index of
    // the element equal to x is returned. Signals
    // notFoundException if none equals x.
```

How is this specification too restrictive?

We can remove these unnecessary restrictions.

```java
public static int search(int[] a, int x)
    throws NotFoundException, NullPointerException
    // Effects: If a is null, throws NullPointerException; else
```
This is a *definitional* specification, not an *operational* specification like the first one.

Why are definitional specifications usually better?

Why are operational specifications often written?

To turn an operational specification into a definitional specification, consider every property and decide whether it is really needed.

If not, it should be eliminated or weakened.

*Exercise:* Think of another example of an overly restrictive specification and how you might weaken it.

*Clarity*

Specifications should be clear. It is hard to say precisely what this means, but it does mean that

- specifications should not be too short,

- specifications should not be too long,

Which of the following is clearer?

```java
static boolean subset(IntSet s1, IntSet s2)
    throws NullPointerException
//  Effects: If s1 or s2 is null, throws NullPointerException
//  otherwise returns true if s1 is a subset of s2 else returns false
```

or

```java
static boolean subset(IntSet s1, IntSet s2)
    throws NullPointerException
//  Effects: If s1 or s2 is null, throws NullPointerException
```
A lot depends on the understanding of readers.

For example,

```c
static float pv(float inc, float r, int n)
// Requires: inc > 0 && r > 0 && n > 0
// Effects: Returns the present value of an annual income
// of inc for n years at an interest rate of r,
// i.e., pv(inc, r, n) =
// inc + (inc/(1+r) + ... + (inc/(1+r)^n-1)
// e.g., pv(100, .1, 3) = 100 + 100/1.1 + 100/1.21
```

Why is it good to use both the “i.e.” and the “e.g.”?

To keep readers from wasting time understanding a specification, it is good to clearly identify redundant information. How can we indicate it?

By associating “Requires” and “Effects” clauses with a routine \( r \), the class tells its clients

“If you promise to call \( r \) with Requires satisfied then I, in return, promise to deliver a final state in which Effects is satisfied.

As in human relationships, any good contract between classes entails obligations as well as benefits for both parties (the “client” and the “supplier”).

- The precondition binds the ____________. It defines the conditions under which a call to the routine is legitimate.
  - It is an obligation for the ______. and a benefit for the ______.
• The postcondition binds the _________. It defines the conditions that must be ensured by the routine on return. It is an obligation for the ________ and a benefit for the ________.

Let us take a look at one of the contracts we specified above.

<table>
<thead>
<tr>
<th>pv</th>
<th>Obligations</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>(Satisfy precondition)</td>
<td>(From postcond.)</td>
</tr>
<tr>
<td>Supplier</td>
<td>(Satisfy postcond.)</td>
<td>(From precond.)</td>
</tr>
</tbody>
</table>

Where in the software development process should specifications be evolved?

Specifications are critical for suppliers, who otherwise would not know what they could rely on in implementing their modules.

There would be only the code, and the code can change over time.

Testing

[L&G, Ch. 10] One of the major purposes of specifications is to give us a way to determine when our program is correct.

Validation refers to a process to increase our confidence that a program will function as it is supposed to.

There are two ways to validate a program.
• **Verification:** Provide a proof that the program will work on all possible inputs. This involves careful reasoning about each line in the program.

• **Testing:** Run a program on a “representative set” of inputs and make sure it works for them.

Usually there are too many combinations of inputs for exhaustive testing.

So we must choose test cases carefully.

**Black-box testing**

In black-box testing, we generate test data from the specification alone.

We do not consider the internal structure of the module being tested.

What are some advantages of black-box testing?

Black-box tests need not be changed even when major changes are made to the program.

**Testing paths through the specification:** We might explore alternate paths through the specification.

These paths can be through both the requires and effects clauses.

Consider this specification:

```c
static float sqrt(float x, float ε)
// Requires: x • 0 && .00001 < ε < .001
// Effects: Returns sq Ǝ x-ε • sq*sq • x+ε
```
The *requires* clause is the conjunction of two terms:

- \( x \cdot 0 \)
- \( .00001 < \varepsilon < .001 \)

How might each of these be satisfied?

The first can be satisfied in two ways:

We should at least test these two cases:

- \( \ldots && .00001 < \varepsilon < .001 \)
- \( \ldots && .00001 < \varepsilon < .001 \)

Similarly, in the specification

```java
static boolean isPrime(int x)
// Effects: If x is a prime returns true; else returns false
```

what two cases should we test?

Also, we should test all exceptional conditions …

```java
public static int search(int[] a, int x)
    throws NotFoundException, NullPointerException
// Effects: If a is null, throws NullPointerException; else
// returns i such that a[i] = x. Signals
// notFoundException if none equals x.
```

*Testing boundary conditions:* A program should be tested on “typical” inputs, but also on “atypical” inputs.

Testing all paths causes some boundary conditions to be covered, e.g., a test for finding the square root of 0.

But others may be missed, e.g.,
• logical errors, in which a path to handle a special case presented by a boundary condition is omitted.

• checking for conditions that may cause the underlying VM to raise an exception, e.g., arithmetic overflow.

To cover the latter kind of error, it is good to have test data cover the largest and smallest possible values for all bounded numerical arguments.

Aliasing errors: Another kind of error that frequently occurs is where a single data item is referred to by two names.

An example is passing the same value twice to a method such as append:

```java
static void append(Vector v1, Vector v2)
    throws NullPointerException
```

Glass-box testing

Black-box testing is rarely sufficient.

Without looking at the internal structure of a program, it is impossible to know which test cases are likely to give new information.

For example, if a program does a table lookup for some values, but computes others, we need to test both.

So we need to take into consideration the structure of a program.

The goal is to make sure that each path is tested. Such a test set is path complete.

Consider this program.

```java
static int maxOfThree(int x, int y, int z) {
    if (x > y)
        if (x > z) return x; else return z;
    if (y > z) return y; else return z; }
```
Despite the fact that the number of inputs is nearly infinite, there are only 4 paths through the program.

What data would be representative of these four cases?

But path-completeness is not a panacea:

```java
static int maxOfThree(int x, int y, int z) {
    return x;
}
```

can be tested in a path-complete way with this input:

This is a case where some paths are missing.

Another problem is that there may be too many paths to test.

A program that iterates over a large set of inputs may have thousands or millions of paths, or more.

We can come up with some general rules that usually help us here, though.

- For loops with a fixed maximum of iterations, we can test the maximum number of iterations.
- For loops with a variable amount of iteration, we test with 0, 1, and 2 iterations.
  In addition, we consider test cases for all possible ways of terminating the loop.
- For recursive procedures, we include test cases that cause the procedure to return without any recursive calls, and test cases that cause exactly one recursive call.