Correctness

Last updated: March 11th 2019, at 11.30am

Contents

1 Why Show a Programme Correct?

A few examples. Some of these examples are well documented, some may be apocryphal. They all serve as good examples of what could go wrong...

- Airline booking system

  Northwest Airlines had an online booking/information system that, if you asked it for a list of flights, returned the results supposedly sorted by departure time of the outbound flight. For example, a request for flights from San Francisco (SFO) to Minneapolis (MSP) returned the following listing (return flight and other non-relevant data discarded):

Departs	Arrives	Flight Number
6:25am	12:04pm	NW928
7:50am	1:28pm	NW344
10:15am	3:47pm	NW350
11:30am	5:16pm	NW588
12:40am	6:09am	NW360
3:25pm	9:01pm	NW354
5:00pm	10:31pm	NW358

The risk? Assuming that because 11:30am is later than 10:15 am it follows that 12:40am is later than 11:30am.

- Gym changing room access

  Dr. Louise Selby joined "Pure Gym" in Cambridge, and was given a PIN that would give her access to the gym, and to the changing rooms. Access to the gym worked fine. However, when she tried to enter the changing room her PIN wouldn't work. The reason — the PIN system automatically classified anyone with the title "Dr." as male (despite a "M/F" tick box being part of the application form), and she, of course, was trying to enter the women's changing room.

  The gym's response to the problem:

  > "The system is a product that we license and is therefore not something we built this way, but we are working on it.
  >
  > "In the meantime, we have removed the option of 'Dr' as a title to choose from when new members sign up, and are urging anyone who signed up to Pure Gym as a doctor recently to contact our membership team to prevent any issues entering changing rooms."

  They also stated "we have found a bug in the membership system..." It is arguable whether they found it, or Dr. Selby found it.

  See: Cambridge Evening News, 18/3/15 (`http://www.cambridge-news.co.uk/Cambridge-paediatrician-8217-s-outrage-Pure-Gym/story-26188693-detail/story.html`)

- Mars Climate Orbiter

  Mars Climate Orbiter was launched by NASA in December 1998. It was due to enter into orbit around Mars in September 1999. However, communication was lost as it tried to do so, and was never re-established. The original loss of communication *was* expected, as the orbiter went behind Mars (as seen from Earth). However, this took place 49 seconds earlier than expected, and the orbiter never resumed contact.

  Investigation showed that the orbiter had almost certainly descended too close to Mars, and disintegrated in the Martian atmosphere. The reason

for this, investigation showed, was that a piece of ground software was programmed in US units (pounds, feet, etc.), while the orbiter itself, conforming to the Software Interface Specification provided by NASA, was programmed in SI (metric) units.

The total cost of the mission was $327.6 million.

- Banking mail shot

  A well known high street bank wanted to send a mail shot to its wealthiest customers advertising its investment service. A programmer was employed to write a programme that would trawl through the bank's accounts system identifying the wealthiest 5% of its customers. The bank didn't want to risk its database during development of the programme, so the programmer constructed a test data base with customers called "Joe Average", "Rich Bastard", "Stony Broke", and so on. The programme was completed and successfully selected "Rich Bastard" from the test data base.

  The bank then let it loose on the world, at which point its wealthiest customers all received a letter that started

  > Dear Rich Bastard, ...

  See: Snopes.com urban legends (`http://www.snopes.com/business/consumer/bastard.asp`)

- Torpedo system

  A torpedo fired from a submarine will usually home in on the largest piece of metal it can find in the vicinity. Normally this will be the ship that it was fired at. It sometimes happens though that the torpedo homes in on the submarine that fired it. Many torpedoes also have some form on inertial guidance system. A new torpedo was developed that used the inertial guidance system to protect the submarine.

  If the torpedo detected that it had turned through 180° it would immediately explode — the idea being that it would then be heading back to the submarine, but still at a safe distance from it.

  The torpedo was taken out for a field test. It was armed, and the order to fire was given. Unfortunately the firing mechanism failed to work, and the torpedo refused to leave the firing tube. After repeated unsuccessful attempts the commander of the submarine finally gave up, and gave the order to turn for home.

  The submarine, with the armed torpedo still in the firing tube, turned through 180°...

2 How to Show a Programme Correct

2.1 Example: Calculating Squares

I claim that the following procedure will calculate the square of n.

```
public int square(int n) {
   int i = 0, square = 0, twoN = 0;
   while (i != n) {
      square = square + twoN + 1;
      twoN = twoN + 2;
      i++;
   }
   return square;
}
```

How can we show that this programme is correct?

2.2 Prove the Programme

2.2.1 Writing Test Cases

Rigorous testing of software is always required for any software to be used in earnest and automated testing is a good way to do this. As the code is developed the same tests frequently have to be run again and again. In fact, it can be a good idea to write the test cases before writing the actual code. One advantage of doing this is that if you have worked out the test cases then then you have worked out *what* it is that your code must do and of course understanding that is an essential pre-requisite to being able to write the code in the first place.

```
public class SquareTest extends junit.framework.TestCase {
   public void test0() {
      assertEquals(0, new Square.square(0));
   }
   public void test1() {
      assertEquals(1, new Square.square(1));
   }
   ⋮
}
```

2.3 Prove the Programme Correct

Add *assertions* (true statements) and prove that each assertion follows from the preceding assertion(s).

2.3.1 A Simple Example

E.g.

$\{\mathsf{twoN} = 2\mathsf{i}\}$
`twoN = twoN + 2;`
$\{\mathsf{twoN} = 2(\mathsf{i}+1)\}$

or, in more detail, with a temporary variable `tmp`:

$\{\mathsf{twoN} = 2\mathsf{i}\}$
`tmp = twoN + 2;`
$\{\mathsf{twoN} = 2\mathsf{i}, \mathsf{tmp} = \mathsf{twoN}+2 = 2\mathsf{i}+2 = 2(\mathsf{i}+1)\}$
`twoN = tmp;`
$\{\mathsf{twoN} = 2(\mathsf{i}+1)\}$

2.3.2 The Proof

The aim We can try to prove this by "proof by wishful thinking". I.e. we assume that we will be able to prove what we want, and see what needs to be true in order to be able to prove it.

We want to prove that the result is the desired value — i.e. $\mathtt{n}^2$, hence assertion $\boxed{6}$.

```
public int square(int n) {
   int i = 0, square = 0, twoN = 0;
   while (i != n) {
      square = square + twoN + 1;
      twoN = twoN + 2;
      i++;
   }
   {6} {square = n^2}
   return square;
}
```

Introduce the loop variable We rephrase the assertions to make use of the loop variable i, as this is the variable used in most of the assignments. At the end of the loop `i` will be equal to `n` (because the **while** test fails), so we now want `square` to be $\mathtt{i}^2$, as shown in the updated assertion $\boxed{6}$.

```
public int square(int n) {
   int i = 0, square = 0, twoN = 0;
   while (i != n) {
      square = square + twoN + 1;
      twoN = twoN + 2;
```

```
        i++;
    }
    6 {i = n, square = i² = n²}
    return square;
}
```

Add necessary preconditions In order for `square` $= \texttt{i}^2$ to be true after the **while** loop it needs to be true:

- before the **while** loop (because it's possible the code might skip over the whole loop),
- and at the end of the loop (because it's from here that the code loops back to the test, at which point it could exit the loop).

So we add assertions 1 and 5.

```
1  public int square(int n) {
2      int i = 0, square = 0, twoN = 0;
3      1 {square = i²}
4      while (i != n) {
5          square = square + twoN + 1;
6          twoN = twoN + 2;
7          i++;
8          5 {square = i²}
9      }
10     6 {i = n, square = i² = n²}
11     return square;
12 }
13
```

Reality check: is the first assertion true? Yes, `i` $= 0$ and `square` $= 0$, from the initial assignments on line 2, so `square` $= 0 = 0^2 = \texttt{i}^2$.

A free assertion If `square` $= \texttt{i}^2$ in assertion 1, and also in assertion 5, it must also be true at the start of the **while** loop, so we add assertion 2.

```
1  public int square(int n) {
2      int i = 0, square = 0, twoN = 0;
3      1 {square = i²}
4      while (i != n) {
5          2 {square = i²}
6          square = square + twoN + 1;
7          twoN = twoN + 2;
8          i++;
```

```
9         5 {square = i²}
10    }
11    6 {i = n, square = i² = n²}
12    return square;
13 }
14
```

Add more preconditions In order for `square` to be $= \texttt{i}^2$ in assertion 5, `square` must be $(\texttt{i}+1)^2$ before the the increment to `i` on line 9 (because the "new" `i` in assertion 5, *after* the `i++` instruction, where `square` $= \texttt{i}^2$, will be one more than the "old" `i`). So we add assertion 4 (and hope that it is true).

```
1  public int square(int n) {
2     int i = 0, square = 0, twoN = 0;
3     1 {square = i²}
4     while (i != n) {
5        2 {square = i²}
6        square = square + twoN + 1;
7        twoN = twoN + 2;
8        4 {square = (i + 1)²}
9        i++;
10       5 {square = i²}
11    }
12    6 {i = n, square = i² = n²}
13    return square;
14 }
15
```

Another free assertion The assignment on line 8 doesn't affect the value of `square`, so if `square` $= (\texttt{i}+1)^2$ is true (in assertion 4) after the assignment it must be true before it, so we add assertion 3.

```
1  public int square(int n) {
2     int i = 0, square = 0, twoN = 0;
3     1 {square = i²}
4     while (i != n) {
5        2 {square = i²}
6        square = square + twoN + 1;
7        3 {square = (i + 1)²}
8        twoN = twoN + 2;
9        4 {square = (i + 1)²}
10       i++;
11       5 {square = i²}
12    }
```

```
13      6 {i = n, square = i² = n²}
14      return square;
15  }
16
```

Rewrite $(\texttt{i}+1)^2$ We can expand $(\texttt{i}+1)^2$:

```
public int square(int n) {
    int i = 0, square = 0, twoN = 0;
    1 {square = i²}
    while (i != n) {
        2 {square = i²}
        square = square + twoN + 1;
        3 {square = i² + 2i + 1 = (i + 1)²}
        twoN = twoN + 2;
        4 {square = (i + 1)²}
        i++;
        5 {square = i²}
    }
    6 {i = n, square = i² = n²}
    return square;
}
```

Some wishful thinking Before the assignment on line 6 we have, from assertion 2, `square` $= \texttt{i}^2$ (and also, trivially, `1` $= 1$), and after it we have, in assertion 3, `square` $= \texttt{i}^2 + 2\texttt{i} + 1$, so, if `twoN` $= 2\texttt{i}$, we *can* conclude assertion 3 from assertion 2.

`square`	=	`square`	+	`twoN`	+	`1`
		$\texttt{i}^2$		`2i?`		`1`

So let's assume, in wishful thinking, that, before the assignment, `twoN` $= 2\texttt{i}$, and add this to assertion 2.

```
1   public int square(int n) {
2       int i = 0, square = 0, twoN = 0;
3       1 {square = i²}
4       while (i != n) {
5           2 {square = i², twoN = 2i}
6           square = square + twoN + 1;
7           3 {square = i² + 2i + 1 = (i + 1)²}
8           twoN = twoN + 2;
9           4 {square = (i + 1)²}
10          i++;
```

```
11          5 {square = i²}
12      }
13      6 {i = n, square = i² = n²}
14      return square;
15  }
16
```

Add some assertions for `twoN` In the same way that we needed to add some preconditions for the value of `square`, we need to add some for `twoN`. If `twoN` $= 2$`i` in assertion 2 it must also be the case that `twoN` $= 2$`i` in assertions 1, 3 (the assignment on line 6 doesn't change the value of `twoN`), 5 (so that it will be true at assignment 2 again when we loop round), and in 6 (though we don't actually need this for the proof). In assertion 4, prior to the increment of `i` on line 10, we need `twoN` $= 2($`i` $+ 1)$, for exactly the same reason that we need `square` $= ($`i` $+ 1)^2$.

We therefore add these conditions to the relevant assertions, giving:

```
1   public int square(int n) {
2       int i = 0, square = 0, twoN = 0;
3       1 {square = i², twoN = 2i}
4       while (i != n) {
5           2 {square = i², twoN = 2i}
6           square = square + twoN + 1;
7           3 {square = i² + 2i + 1 = (i + 1)², twoN = 2i}
8           twoN = twoN + 2;
9           4 {square = (i + 1)², twoN = 2(i + 1)}
10          i++;
11          5 {square = i², twoN = 2i}
12      }
13      6 {i = n, square = i² = n², twoN = 2i}
14      return square;
15  }
16
```

Reality check for the assertions about `twoN`:

- Assertion 1: `twoN` $= 0 = 2 \times 0 =$ `2i`, so this is OK.
- Assertion 2: Certainly true on first entry to the loop because of the previous assertion. If we can prove assertion 5, then this will complete the proof of this assertion.
- Assertion 3: The assignment to `square` hasn't changed the value of `twoN`, so this follows from assertion 2.

- Assertion $\boxed{4}$: This follows from assertion $\boxed{3}$ ($\texttt{twoN} = 2\texttt{i}$), and from the assignment. After the assignment we have $\texttt{twoN} = 2\texttt{i} + 2 = 2(\texttt{i} + 1)$.
- Assertion $\boxed{5}$: This follows from assertion $\boxed{4}$, and from the increment in the value of `i`.
- Assertion $\boxed{6}$: Follows from assertions $\boxed{1}$ and $\boxed{5}$.

So all the assertions are justified, and the proof is complete.

The final proof So the final proof is:

```
public int square(int n) {
   int i = 0, square = 0, twoN = 0;
   1 {square = i^2, twoN = 2i}
   while (i != n) {
      2 {square = i^2, twoN = 2i}
      square = square + twoN + 1;
      3 {square = i^2 + 2i + 1 = (i + 1)^2, twoN = 2i}
      twoN = twoN + 2;
      4 {square = (i + 1)^2, twoN = 2i + 2 = 2(i + 1)}
      i++;
      5 {square = i^2, twoN = 2i}
   }
   6 {i = n, square = i^2 = n^2, twoN = 2i}
   return square;
}
```

2.3.3 Assertions in Java

You can add assertions to your Java code using either the **assert** keyword, or by calling the `assert(...)` method: [1]

```
public int square(int n) {
   int square = 0, twoN = 0;
   for (int i = 0; i < n; i++) {
      assert square == i*i : square; assert twoN == 2*i : twoN;
      square = square + twoN + 1;
      assert square == (i+1)*(i+1); assert twoN == 2*i;
      twoN = twoN + 2;
      assert(square == (i+1)*(i+1)); assert(twoN == 2*(i + 1));
   }
   assert square == n*n : square; assert twoN == 2*n;
```

[1] In this, and the previous example I would normally put the assertions on separate lines, after the instructions. I have had to put some of them after the instruction on the same line to get the code to fit on one slide.

```
    return square;
}
```

This code demonstrates three different ways of putting assertions into your code:

- The first pair of assertions use the **assert** keyword with just one argument, which must be a boolean expression. The assertion will succeed if this expression is true, and fail otherwise, in which case an `AssertionError` exception will be thrown;
- The second pair also uses the **assert** keyword, but this time with two arguments, separated by a colon. The first argument must again be a boolean expression. The second must be an expression having a non-**void** value. This value will be reported by any `AssertionError` thrown by the assertion;
- The third pair of assertions uses the `assert(...)` method. This method takes one **boolean** argument, and works just like the first version of the **assert** keyword.

Note that including assertions in your code does not guarantee that your programme is correct. Java does not check the mathematical correctness of the proof, it merely tests the assertions at run time. I.e. Java assertions are a form of testing, not of proof. Also assertions must be enabled before they are tested at all. The default is for assertions to be disabled. See the tutorial notes for information on how to enable assertions.

2.4 Improve a Correct Programme

Start with a simple but correct programme and, by a series of *semantically equivalent transformations*, develop it into a more complex but still correct programme.

2.4.1 A Simple Example: Recursion Elimination

```
public SomeType f(int n) {
    if (n == 0) {
        return aResult;
    } else {
        return g(f(n-1));
    }
}
```

Note: we have used **int** as the type of the parameter of the method `f`. It could have been any type (on which we could define some "simplifying action", and some "terminal value" for implementing the recursion). The **int** type is used here only to make the example simpler.

From this definition it is fairly easy to see, especially if we think of `f` and `g` as functions, that:

- $\texttt{f}(0) = \texttt{aResult}$
- $\texttt{f}(1) = \texttt{g}(\texttt{aResult})$
- $\texttt{f}(2) = \texttt{g}(\texttt{g}(\texttt{aResult}))$
- ...
- $\texttt{f}(\texttt{n}) = \texttt{g}^{\texttt{n}}(\texttt{aResult})$
- ...

And from this it is relatively easy to construct an equivalent iterative version of the method:

```
public SomeType f(int n) {
   SomeType result = aResult;
   int i = 0;
   while (i != n) {
      result = g(result);
      i++;
   }
   return result;
}
```

2.4.2 The transformation

First attempt The aim is to take the definition of the function/method, and to rewrite to create a recursive definition. I.e. we will start by defining the method `sqr` to simply be $\texttt{n}^2$, and then aim to rewrite so that we have a definition in terms of $\texttt{square}(\texttt{n}-1)$ and constants only.

Begin with a simple programme

```
public int square(int n) {
   int square;
   square = n²;
   return square;
}
```

It's hard to argue that this programme is incorrect!

Introduce recursion

```
public int square(int n) {
   int square;
   if (n == 0) {
      square = 0;
   } else {
      square = square(n-1) - square(n-1) + n²;
   }
   return square;
}
```

Obviously, adding square(n − 1) − square(n − 1) to the result has no effect.

Replace a method call by its value We replace the second call of square(n − 1) by its value — $(n-1)^2$.

```
public int square(int n) {
   int square;
   if (n == 0) {
      square = 0;
   } else {
      square = square(n-1) - (n − 1)² + n²;
   }
   return square;
}
```

Expand We expand the $(n-1)^2$.

```
public int square(int n) {
   int square;
   if (n = 0) {
      square = 0;
   } else {
      square = square(n-1) - n² + 2n - 1 + n²;
   }
   return square;
}
```

In a little more detail:

$$\begin{aligned} \texttt{square}(\texttt{n}-1) - (\texttt{n}-1)^2 + \texttt{n}^2 &= \texttt{square}(\texttt{n}-1) - (\texttt{n}^2 - 2\texttt{n} + 1) + \texttt{n}^2 \\ &= \texttt{square}(\texttt{n}-1) - \texttt{n}^2 + 2\texttt{n} - 1 + \texttt{n}^2 \end{aligned}$$

Simplify Simplify the expression — this gets rid of the n^2 for us.

```
public int square(int n) {
   int square;
   if (n == 0) {
      square = 0;
   } else {
      square = square(n-1) + 2n - 1;
   }
   return square;
}
```

Standardise Rewrite $2n$ in terms of $2(n-1)$...

```
public int square(int n) {
   int square;
   if (n == 0) {
      square = 0;
   } else {
      square = square(n-1) + 2(n - 1) + 1;
   }
   return square;
}
```

In a little more detail:

$$\begin{aligned} \texttt{square(n}-1)+2\texttt{n}-1 &= \texttt{square(n}-1)+2\texttt{n}-2+2-1 \\ &= \texttt{square(n}-1)+2(\texttt{n}-1)+2-1 \\ &= \texttt{square(n}-1)+2(\texttt{n}-1)+1 \end{aligned}$$

We're still left with an $2(n-1)$ in the expression. We can apply the same method to give a function for $2n$. Unsurprisingly:

```
public int twoN(int n) {
   int twoN;
   if (n == 0) {
      twoN = 0;
   } else {
      twoN = twoN(n-1)+2;
   }
}
```

Return a Pair of Values Rewrite the programme to return a pair of values:[2]

```
public (int square,int twoN) square(int n) {
   (int,int) result;
   if (n == 0) {
      result = (0,0);
   }  else {
      result = (
         square(n-1)→ square + square(n-1)→ twoN + 1,
         square(n-1)→ twoN + 2);
   }
   return square;
}
```

Eliminating the recursion Here g is: $\mathsf{g}(square, twoN) = (square + twoN + 1, twoN + 2)$, so:

```
public (int square,int twoN) square(int n) {
   (int square,int twoN) result = (0,0);
   int i = 0;
   while (i != n) {
      result= (
         result→ square + result→ twoN + 1,
         result→ twoN + 2);
      i++;
   }
   return result;
}
```

Final version We don't need to keep the `square` and `twoN` value in a pair. Also, we're only interested in the final value of the `square`.

```
public int square(int n) {
   int i= 0, square = 0, twoN = 0;
   while (i != n) {
      square = square + twoN + 1;
      twoN = twoN + 2;
```

[2] We could do this in Java by defining a `Pair` class:

```
protected class Pair<A,B> {
   protected A x;
   protected B y;
}
```

but the code in our example is clearer if we juct assume that pairs are a valid data type.

```
        i++;
    }
    return square;
}
```

3 Summary

3.1 The three tenets

- Prove a programme

  Can only prove that a programme is incorrect, not that it is correct.

- Prove a programme correct

  Construct the proof *after* the programme has been written. Strongly reliant on the programme being written in a style conducive to proof.

- Improve a correct programme

  The development of the programme *is* the proof of its correctness

3.2 W?W?W?

- Why do it?

  Would you rather have the local nuclear power station run on correct or incorrect code?

- Would anybody do it?

  No. It's too much like hard work.

- Will anybody do it?

  Maybe. Automated support is being developed, especially in transformational development.

  But proving a programme correct, or improving a correct programme will (always?) be more work than "Well I've not found any bugs" (yet!).

  Until customers demand real guarantees it will be cheaper to issue "patches".

For now, "prove" the programme — i.e. rigorously test the programme using e.g. JUnit tests.