Let tests drive or let Dijkstra derive?

Presented at ALE2014 in Krakow Poland on August 21, 2014

Author: Sander Kooijmans
Date: November 3, 2014

Why this document?

Test-driven development (TDD) is currently hyped. Many people, including TDD trainers, say that you are not a serious software developer unless you use TDD. They also claim that TDD is the **only** way to write correct programs. [professionalism] [tdd-podcast]

Uncle Bob (Robert Martin) even claims that applying TDD and some transformations may even be a formal proof of correctness. This however is still to be proven. [premise]

My experiences with TDD

Do I really need to write a test before I can create an empty class? It is hard to find the next test. And can't I wait with refactoring until my algorithm is complete so that I only refactor once? And how do I know that the production code is correct? How do I know there are no more tests to write? It turns out some people actually fail in finding the next test, see my blog post about the Potter Kata [potter-kata].

In conclusion I feel pretty restrained by TDD.

But what about tests?

Alright, I may have exaggerated a bit about TDD. But what about tests? Here are my experiences with tests:

- Tests require clean code
- Tests protect against accidental modification of existing behavior
  - When adding new functionality
  - When refactoring existing functionality
- Tests are examples of how production code must be used
- Tests **can** be specifications of the production code
  - A test can perfectly explain the expected behavior for a method in case an invalid parameter is passed.
  - Tests almost never cover all possible input values. So what is the behavior for the cases that are not covered by tests? In such cases a specification in text is more clear and concise than the tests.
- Writing tests first helps to clarify specifications
- I don't dare to deliver production code without tests!

Conclusion: I feel quite positive about tests.

The prime factors kata by Uncle Bob

Uncle Bob derived an algorithm to calculate the prime factors of an integer number, see [primes-kata]. Using just 7 tests Uncle Bob derives the following Java program:

```
package primeFactors;

import java.util.*;

public class PrimeFactors {
  public static List<Integer> generate(int n) {
    List<Integer> primes = new ArrayList<Integer>();

    for (int candidate = 2; n > 1; candidate++)
      for (; n%candidate == 0; n/=candidate)
        primes.add(candidate);

    return primes;
  }
}
```

There are infinitely many natural numbers (1, 2, 3, …). The parameter n is of type int and n is assumed to be a positive number. Therefore there are still more than 2 billion possible input values. How can 7 tests convince you that this algorithm works for all these possible input values? And do these two nested loops actually terminate for each possible input value?

I am not convinced by just seven tests at all.

Professor Edsger Wybe Dijkstra (1930-2002)

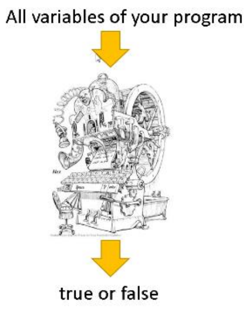
Neither would professor Dijkstra have been. He often claimed: “Program testing can be used to show presence of bugs, but never to show their absence!” You might know professor Dijkstra from:

- Semaphores to synchronize threads
- Shortest path algorithm
- Winner of the Turing Award in 1972
- Goto statement considered harmful
- His handwritten documents, called EWDs

And he was also famous for **program derivation**: to “develop proof and program hand in hand”. I will introduce you to program derivation in the next sections.

Predicates

First I need to tell you about predicates. A **predicate** is a function from the state space of your program to true or false. Or put differently: a predicate is a machine in which you insert all the variables of your program and that returns the value true of false based on the values of the variables.



For example, after execution of the statements[1]

`int i=0; int j=100;`

the following predicates hold (and many more can be conceived):

$P(i,j) \equiv i==0$
$Q(i,j) \equiv i<j$
$R(i,j) \equiv i==0 \;\&\&\; j==100$

For brevity I omit the parameters of the predicate: $R \equiv i==0 \;\&\&\; j==100$

Hoare triple

Tony Hoare invented a formal system to reason about the correctness of algorithms. It is all about Hoare triples. A **Hoare triple** looks like $\{Pre\}\ S\ \{Post\}$. Pre and $Post$ are predicates, S is a statement. This triple has the following interpretation:

When Pre holds and S is executed then if S terminates $Post$ holds. Note that the statement **must terminate** in order for de postcondition to hold!

Using Hoare triples we can now define rules to prove statements.

Assignment

For the assignment the rule is:

$\{Pre\}\ x = E\ \{Post\}$ is equivalent to $Pre \Rightarrow Post(x{:=}E)$

The := operator is the substitution operator. As an example I will proof the following Hoare triple:

$\{x == 5\}\ x = x+3\ \{x > 7\}$

$$
\begin{array}{ll}
 & (x > 7)(x{:=}x+3) \\
\equiv & \{\ \text{substitution}\ \} \\
 & x+3 > 7 \\
\equiv & \{\ \text{calculus}\ \} \\
 & x > 4 \\
\Leftarrow & \{\ \text{precondition}\ \} \\
 & x == 5
\end{array}
$$

Now read from bottom to top and you will see it states $Pre \Rightarrow Post(x{:=}E)$. This is the only example of a formal proof I will show you, because I don't want to limit the audience to university graduates.

[1] Dijkstra invented a special language, called the Guarded Command Language (GCL), which was well suited for his way of program derivation. I will stick to Java because most readers understand Java. This saves me the trouble of explaining the GCL.

If-statement

For the if-statement the rule is:

`{Pre} if (B) S else T {Post}` is equivalent to:
`{Pre && B} S {Post} and {Pre && !B} T {Post}`

So, to prove an if-statement the “then path” and the “else path” have to be proven. For the “then path” the precondition is strengthened by `B`. For the “else path” the precondition is strengthened by `!B`.

As an example let us prove: `{true} if (y>=x) u=y else u=x {u == max(x,y)}`

As promised I will not give a formal proof here. First of all, notice the precondition. It does not mention any variables. Therefore, the precondition is satisfied no matter which values the variables `u`, `x` and `y` have. The postcondition states that `u` is equal to the maximum value of `x` and `y`.

First I prove the “then path”. That is the path for which `y>=x` holds. If `y>=x` then `x` is not larger than `y`. And thus `y` is the maximum value of `x` and `y`. So after the assignment of `y` to `u` the postcondition `u == max(x,y)` is valid.

Next I prove the “else path”. That is the path for which `y>=x` is false. If `y>=x` is false then `x>y` must be true. And if `x` is larger than `y` then for sure it is the maximum value of `x` and `y`. Thus after assigning `x` to `u` the postcondition `u == max(x,y)` is valid.

Since both the “then path” as the “else path” have been proved the if-statement has been proved.

While-loop

`{Inv} while (B) S {Post}` follows from the following three parts:

1. `{Inv && B} S {Inv}`
2. `Inv && !B ⇒ Post`
3. The loop must terminate

Note that I used `Inv` instead of `Pre`. `Inv` is short for invariant. The invariant should hold before the while-loop and it should hold after each iteration, because of part 1. The result of this is that when the loop terminates, the invariant still holds. We can thus use the invariant and `!B` to prove the postcondition, as shown in part 2.

What about part 3? How do you prove that a loop terminates? We use a bound function. A bound function takes the variables of your algorithm and returns an integer value. You must prove that after each iteration the bound function is decreased by at least one. You also need to prove that the bound function cannot get below a lower bound.

As an example I prove an algorithm to calculate 2^n for the given number `n`.

```
// n >= 0 && originalN == n
int p=1;
// invariant: p * 2^n == 2^originalN
// bound function: n >= 0
while (n != 0) {
   p = 2*p; n = n-1;
}
// p == 2^original
```

The first line is the precondition of the algorithm. It states that n is non-negative. It also claims that `n` is equal to `originalN`. You can consider `originalN` as an extra variable that stores the value of n at the beginning of the algorithm. Since the value of `originalN` is only read and not changed by an assignment, there is no need to actually create this variable. We only use it to reason about the algorithm.

Next there is the while-loop. First I have to prove that the invariant is valid right before the while-loop. At that point p is 1 and n is equal to `originalN`. So I fill in `p == 1` in the invariant and that results in `2^originalN == 2^originalN`. That is true, thus the invariant holds before the while-loop.

Does the invariant hold after an iteration of the loop? The first assignment doubles the expression `p * 2^n` and the second assignment halves it again. Thus after execution of the two assignments the invariant holds again.

If the loop terminates, then `n` is zero. If I apply this to the invariant I get `p * 2^0 == 2^originalN`. 2^0 is one so this results in `p == 2^originalN`, which is the postcondition of the algorithm.

The only thing I still need to prove is that the loop terminates. The bound function I choose is `n`. It is decreased by exactly one each iteration. Initially `n` is at least zero and the loop terminates when `n` is zero. Thus the bound function cannot get below zero.

This concludes the proof of the while-loop.

Deriving a solution to the prime factors kata

So far I discussed how to prove an algorithm after the algorithm was written. Now I will use Dijkstra's style to derive an algorithm. I will derive a solution to the prime factors kata of Uncle Bob. I will do this again informally because I do not wish to limit my audience to university graduates.

For those who forgot: a prime number is an integer number that can only be divided by 1 and the number itself (that is: with remainder zero). The number one is not considered to be a prime number. Hence, the smallest prime number is two.

Each integer number except zero consists of a unique product of zero or more prime factors. For example: 10 = 2 * 5, 8 = 2 * 2 * 2 and 1 has zero prime factors.

Let me start with a specification of the algorithm:

```
public static List<Integer> generate(int n) {
    List<Integer> factors = new ArrayList<Integer>();
    // n >= 1 && originalN == n
    "do the work";
    // factors contains all prime factors of originalN
    return factors;
}
```

The method defines a variable called `factors`, which is returned. The pseudo-statement "`do the work`" is the part that has to be derived. Its precondition is that `n` is positive. I use the trick with `originalN` again to store the value of `n`, since I plan on changing `n`. The postcondition of "`do the work`" is that the list factors contains all prime factors of `originalN`.

First attempt

Look at this table:

n	factors
originalN == 140 == 2 * 2 * 5 * 7	[]
2 * 5 * 7	[2]
5 * 7	[2, 2]
7	[2, 2, 5]
1	[2, 2, 5, 7]

My first idea for the algorithm is to make a loop. Each iteration one prime factor of `n` is removed from `n` and added to `factors`. Removing a factor from `n` is performed by dividing `n` by the factor. The loop terminates when there are no more prime factors, that is, when `n` equals 1.

This inspires me to introduce these two invariants:

1. `factors` contains only primes
2. `n * "product of factors" == originalN`

The bound function is `n`. It decreases by at least one each iteration and it cannot get below one. This leads to the following algorithm:

```
// n >= 1 && n == originalN
while (n != 1) {
    int prime = "a prime factor of n";
    n = n / prime;
    factors.add(prime);
}
// n == 1 and invariants imply
// that factors contains all prime factors of originalN.
```

Now I have to find a way to implement "`a prime factor of n`". I will try to remove the prime factors in ascending order. Hey, I can add a third invariant to express this idea:

3. `n` has no prime factors less than `candidate`

candidate is a new variable. How should I initialize it? The smallest prime number is 2. So there are no prime factors smaller than 2. If I initialize candidate to 2 then invariant 3 holds before the while-loop.

What about the statements inside the loop? I'd like to increase candidate, because if candidate is larger than n, then all prime factors have been removed from n. If candidate is not a prime number or if it is not a factor of n, then I can safely increase candidate by one. If candidate is a prime factor and is a factor of n, then I cannot increase candidate. Instead I can remove this prime factor from n. This leads to the following algorithm:

```
// n >= 1 && n == originalN
int candidate = 2;
while (n != 1) {
   if (n % candidate == 0 && "candidate is prime") {
        n = n / candidate;
        factors.add(candidate);
    } else {
        candidate++;
    }
}
// n == 1 and invariants imply that factors contains
// all prime factors of originalN.
```

The expression `n % candidate == 0` is true if and only if candidate is a factor of n.

The bound function n is not correct anymore. In case candidate is not a prime factor of n, the bound function n does not decrease. The bound function `originalN + n – candidate` works. The "then path" of the if-statement decreases n, the "else path" increases candidate thus `originalN + n – candidate` decreases each iteration. n is at least one and candidate is at most n+1. If originalN is a prime number, then candidate will be equal to `originalN + 1` at the end. That is highest value candidate can get. Together with `n >= 1` this results in the lower bound for the bound function: `originalN + n – candidate` is at least zero.

Now I still have to deal with the part "candidate is prime" in case candidate is a factor of n. Here mathematics comes to the rescue! Assume that candidate is not prime. In that case candidate consists of the product of prime factors. Each of these prime factors is smaller than candidate and is a prime factor of n too. But wait, invariant 3 says that that is not possible! This is a contradiction. Therefore I can conclude that if candidate is a factor of n then candidate must be a prime factor. (This proof technique is called a "proof by contradiction".)

So here is the completed algorithm:

```
// n >= 1 && n == originalN
int candidate = 2;
while (n != 1) {
   if (n % candidate == 0) {
        n = n / candidate;
        factors.add(candidate);
    } else {
        candidate++;
    }
}
// n == 1 and invariants imply that factors contains
// all prime factors of originalN.
```

Isn't that great? I derived an algorithm to calculate prime factors without even using Eratosthenes' sieve to find out which number is prime or not.

Second attempt

The algorithm I derived previously either removes a prime factor from n or it increases `candidate`. Now I aim for a solution where I can increase `candidate` each iteration.

Look at this table:

n	factors
originalN == 180 == 2 * 2 * 3 * 3 * 5	[]
3 * 3 * 5	[2, 2]
5	[2, 2, 3, 3]
1	[2, 2, 3, 3, 5]

Again I try to remove prime factors in each iteration. But this time I remove all "occurrences" of a prime factor instead of one occurrence at a time. This leads me to writing this algorithm:

```
// n >= 1 && n == originalN
int candidate = 2;
while (n != 1) {
    "if candidate is prime then move all occurrences of candidate
from n to factors";
    // candidate is not a prime factor of n
    candidate++;
}
// n == 1 and invariants imply that factors contains
// all prime factors of originalN.
```

I use the same three invariants and bound functions as before:

Invariant 1: `factors` contains only primes
Invariant 2: `n * "product of factors" == originalN`
Invariant 3: n has no prime factors less than `candidate`

Bound function: `originalN + n - candidate >= 0`

Now I have to derive code for “`if candidate is prime then move all occurrences of candidate from n to factors`”. This turns out to be quite easy:

```
// n >= 1 && n == originalN
int candidate = 2;
while (n != 1) {
    while (n % candidate == 0 && “candidate is prime”) {
        n = n / candidate;
        factors.add(candidate);
    }
    // candidate is not a factor of n
    candidate++;
}
// n == 1 and invariants imply that factors contains
// all prime factors of originalN.
```

Now I still have to deal with the part “`candidate is prime`” in case `candidate` is a factor of `n`. It turns out the proof given by attempt 1 is applicable here too. Therefore I can conclude that if `candidate` is a factor of `n` then `candidate` must be a prime factor.

So here is the completed algorithm:

```
// n >= 1 && n == originalN
int candidate = 2;
while (n != 1) {
    while (n % candidate == 0) {
        n = n / candidate;
        factors.add(candidate);
    }
    // candidate is not a factor of n
    candidate++;
}
// n == 1 and invariants imply that factors contains
// all prime factors of originalN.
```

Let me apply some syntactic sugar: if I use for-loops instead of while-loops, the algorithm looks like this:

```
    for (int candidate = 2; n > 1; candidate++)
      for (; n%candidate == 0; n/=candidate)
        primes.add(candidate);
```

And this is exactly the same algorithm that Uncle Bob derived using TDD. Now I know it is correct, because I proved it.

Did you see the invariants and mathematics I needed to prove the algorithm? Do you understand now why I am not convinced of the correctness by the seven tests Uncle Bob came up with?

Conclusions

There is no guarantee that TDD results in correct code.

Deriving and proving an algorithm can go hand in hand.

TDD and formal proofs are tools, not goals. The goal is to create correct code. Tests are the best way to protect programmers against introducing bugs in existing code.

References

[premise] http://blog.8thlight.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html

[professionalism] http://blog.8thlight.com/uncle-bob/2014/05/02/ProfessionalismAndTDD.html

[tdd-podcast] http://cleancoders.com/episode/clean-code-episode-6-p2/show

[potter-kata] http://gogognome.nl/2013/01/potter-kata-solution/

[primes-kata] http://butunclebob.com/files/downloads/Prime%20Factors%20Kata.ppt