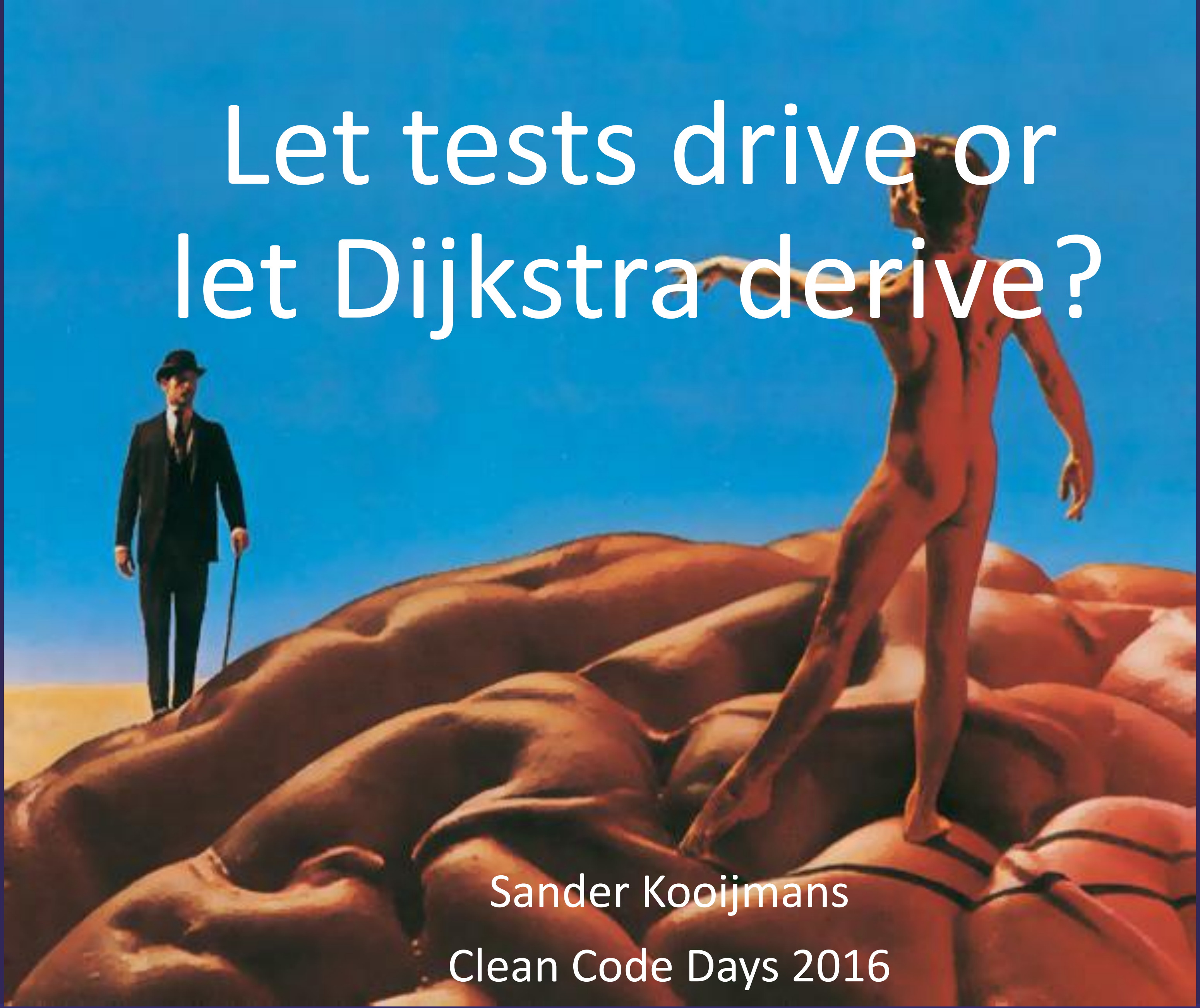
Let tests drive or let Dijkstra derive?



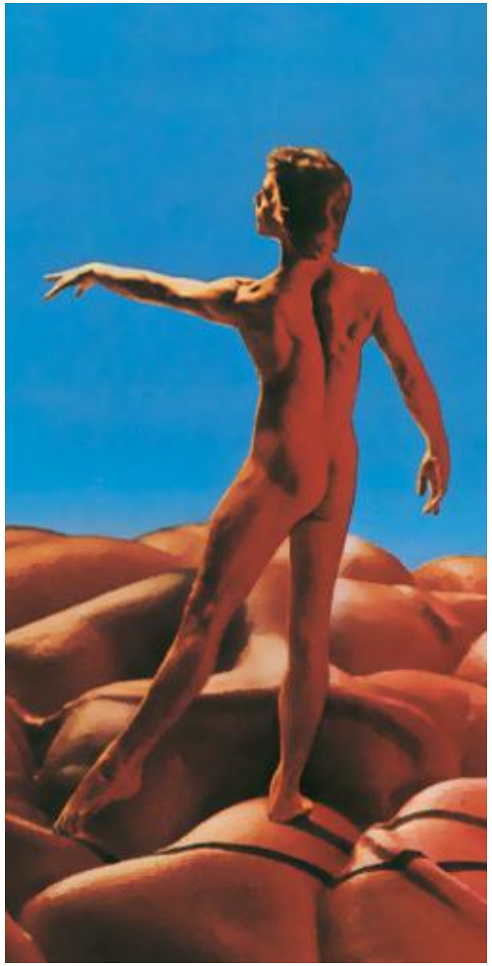
Sander Kooijmans

Clean Code Days 2016

high tech ICT

Apollo

Dionysos



Cygnus

Who is Sander Kooijmans?

www.gogognome.nl

Why this presentation?

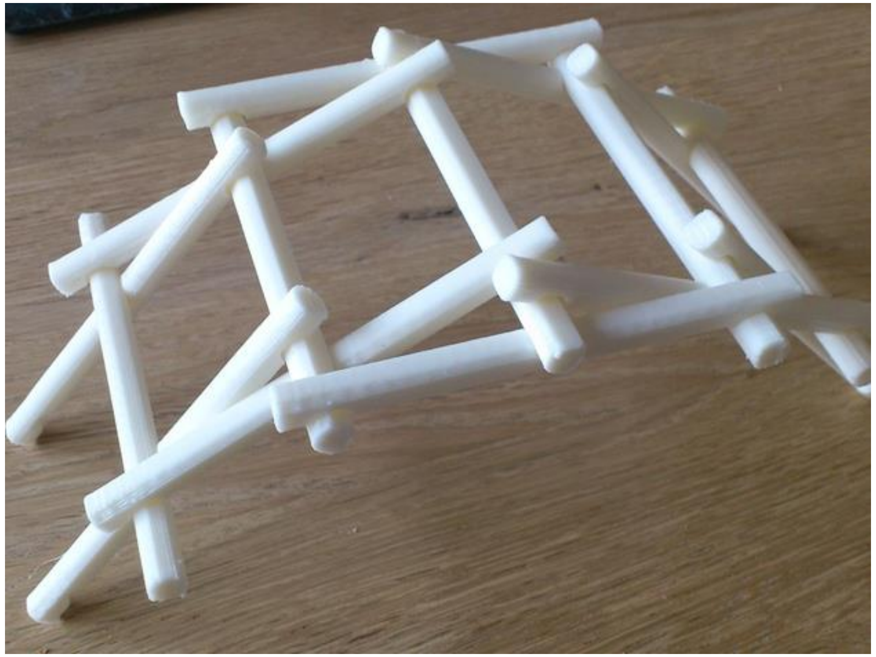
- You are not a serious programmer if you do not use TDD
- The only way to write correct programs is using TDD
- Applying TDD and transformations may be a formal proof of correctness (still to be proven)
- TDD gives an empirical evidence for correctness (April 2016)

What to expect?

- My experiences with tests and TDD
- Uncle Bob’s solution to the primes kata
- How to prove algorithms are correct
- Deriving a solution for the primes kata (Dijkstra style)
- My conclusions

My experience with TDD

My experience with TDD



My experience with tests

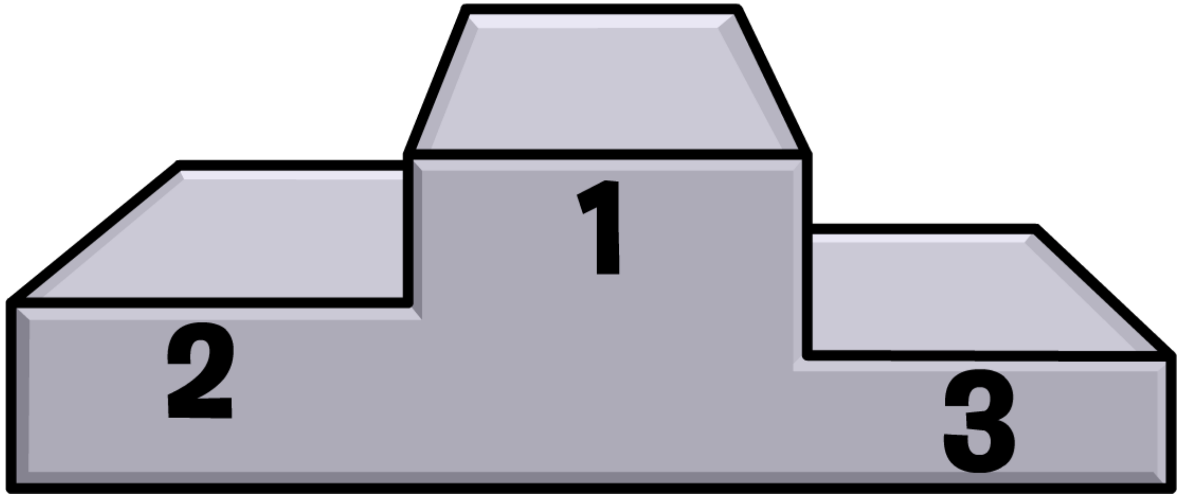
My experience with tests

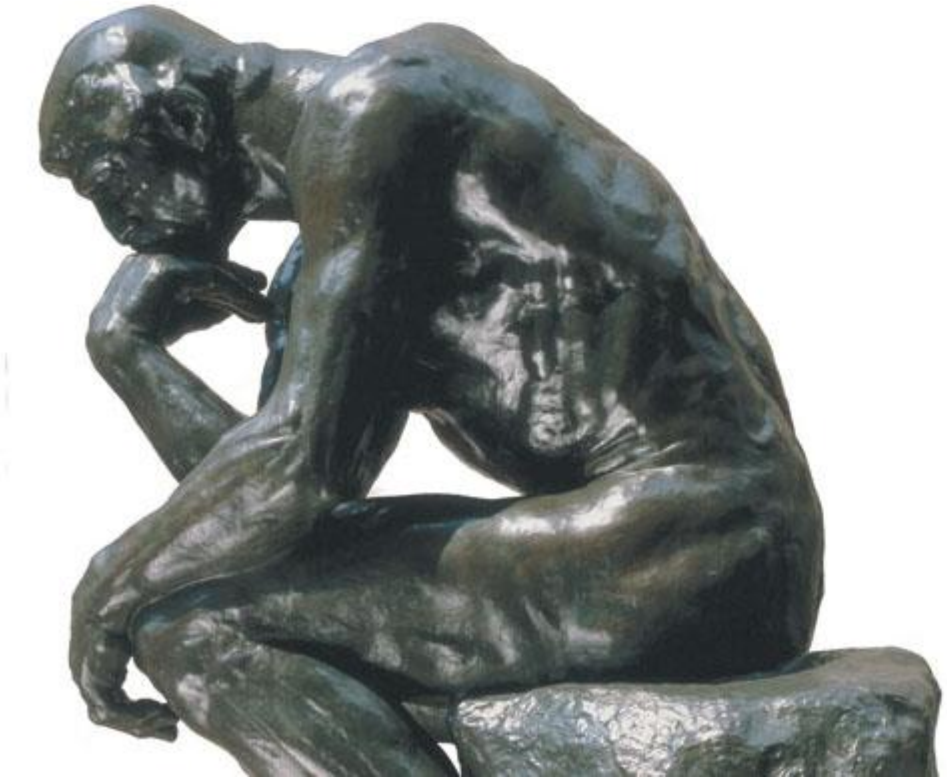
My experience with tests

My experience with tests

```
@Test
public void testCountOccurrences() {
    assertEquals(0, countNrOccurrences("A", ""));
    assertEquals(0, countNrOccurrences("A", “BC"));
    assertEquals(1, countNrOccurrences("A", "ABC"));
    assertEquals(3, countNrOccurrences("A", “BACABAB"));
}
```

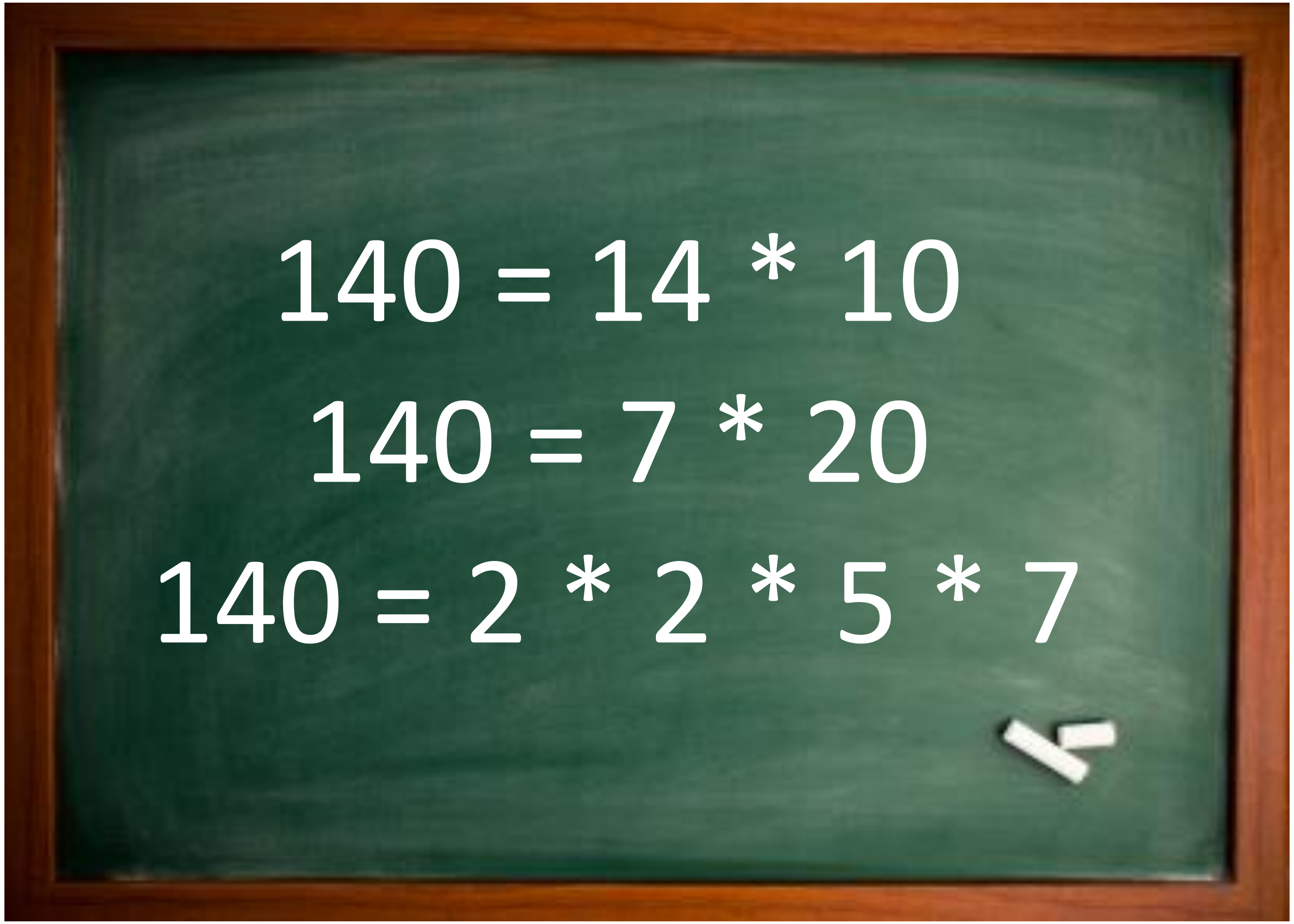
My experience with tests





My experience with tests

The Prime Factors Kata by Uncle Bob



The Prime Factors Kata by Uncle Bob

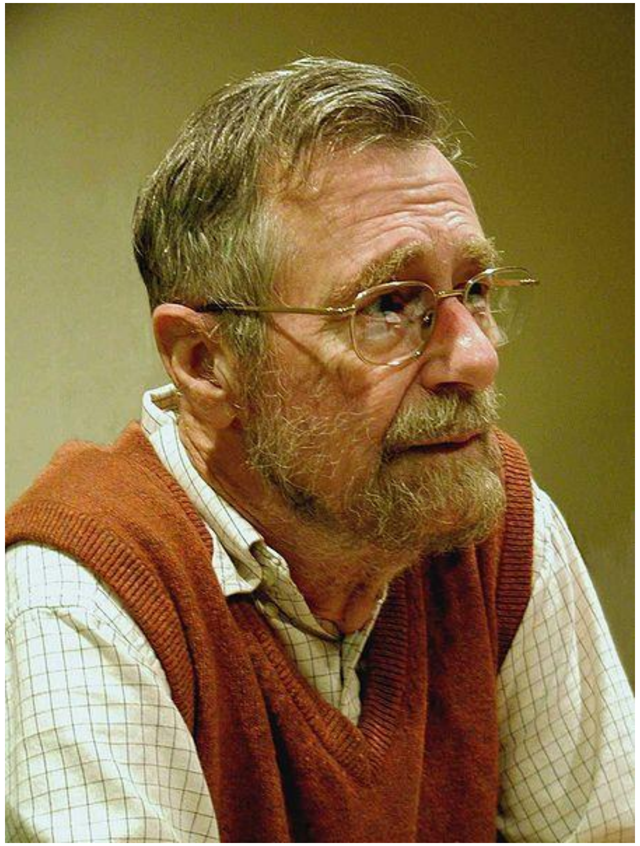
```java
package primeFactors;

import static primeFactors.PrimeFactors.generate;
import junit.framework.TestCase;
import java.util.*;

public class PrimeFactorsTest extends TestCase {
  private List<Integer> list(int... ints) {
    List<Integer> list = new ArrayList<Integer>();
    for (int i : ints)
      list.add(i);
    return list;
  }

  public void testOne() throws Exception {
    assertEquals(list(),generate(1));
  }

  public void testTwo() throws Exception {
    assertEquals(list(2),generate(2));
  }

  public void testThree() throws Exception {
    assertEquals(list(3),generate(3));
  }

  public void testFour() throws Exception {
    assertEquals(list(2,2),generate(4));
  }

  public void testSix() throws Exception {
    assertEquals(list(2,3),generate(6));
  }

  public void testEight() throws Exception {
    assertEquals(list(2,2,2),generate(8));
  }

  public void testNine() throws Exception {
    assertEquals(list(3,3),generate(9));
  }
}
```

```java
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;
  }
}
```

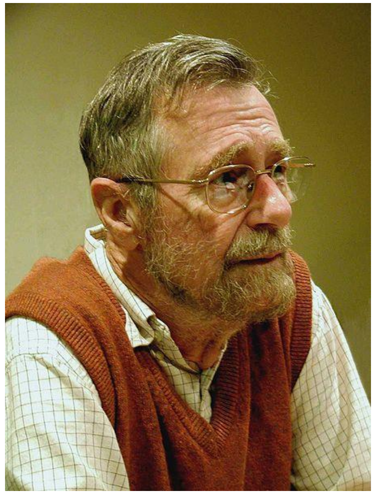
Why is this solution correct?
Does it terminate for all possible inputs?

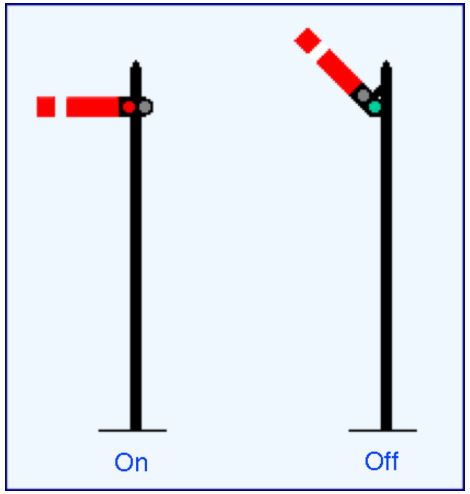
http://butunclebob.com/files/downloads/Prime%20Factors%20Kata.ppt

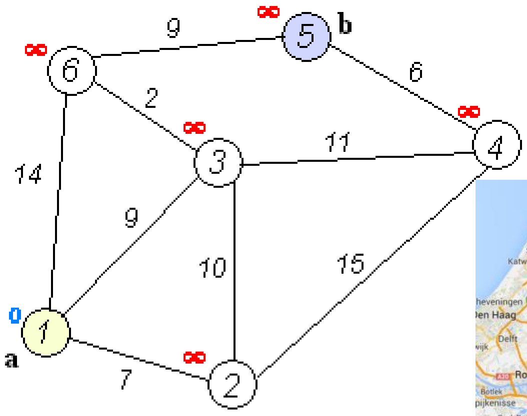
Edsger Wybe Dijkstra (1930-2002)

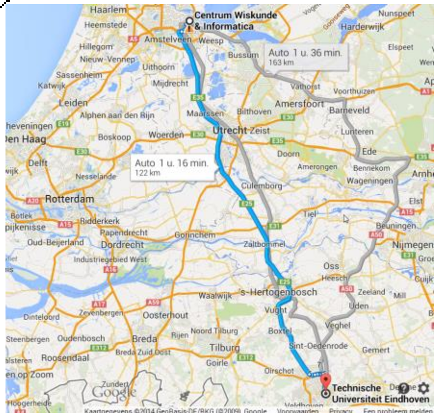


Edsger Wybe Dijkstra



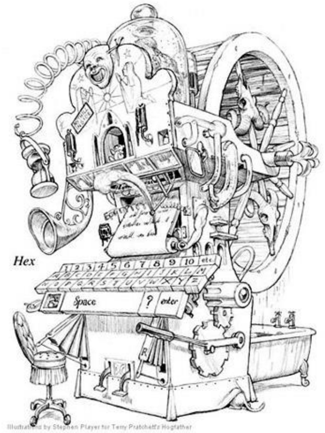






Predicates

All variables of your program



true or false

```
int a=4; int b=8;
```

After execution these predicates hold:

```
P(a) ≡ a == 4
Q(a,b) ≡ a < b
R(a,b) ≡ 2*a == b
```

For brevity we write

```
R ≡ 2*a == b
```

Hoare triple

The Hoare triple `{Pre} S {Post}` means:

When `Pre` holds and `S` is executed

then if `S` terminates `Post` holds

In the Java code samples I use this notation:

```
// Pre
S
// Post
```

Assignment

`{Pre} x = E {Post}` is equivalent to `Pre ⇒ Post(x:=E)`

Let us prove

`{x == 40} x = x+2 {x == 42}`

```
    (x == 42)(x:=x+2)
  ≡ { substitution }
    x+2 == 42
  ≡ { arithmetic }
    x == 40
```

Assignment

`{Pre} x = E {Post}` is equivalent to `Pre ⇒ Post(x:=E)`

Let us prove

```
{x == 5} x = x+3 {x > 7}

     (x > 7)(x:=x+3)
  ≡ { substitution }
     x+3 > 7
  ≡ { calculus }
     x > 4
  ⇐ { precondition }
     x == 5
```

If-statement

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

`{Pre && B} S {Post} and {Pre && !B} T {Post}`

Let us prove

`{true} if (y>=x) u=y else u=x {u == max(x,y)}`

While-loop

`{Inv} while (B) S {Post}` follows from

- `{Inv && B} S {Inv}`
- `Inv && !B ⇒ Post`
- Provided that the loop terminates

`Inv` is called an **invariant**

Termination of a while-loop

Termination is proved using a **bound function**

- A bound function decreases with at least one each iteration
- The bound function is bounded from below

While-loop: proving an algorithm to calculate 2^n

```
// pre: n >= 0
```

```
// post: p == 2^n
```

While-loop: proving an algorithm to calculate 2^n

```
// pre: n >= 0
int p=1; int i=0;


while (i != n) {
    p = 2*p; i = i+1;
}
// post: p == 2^n
```

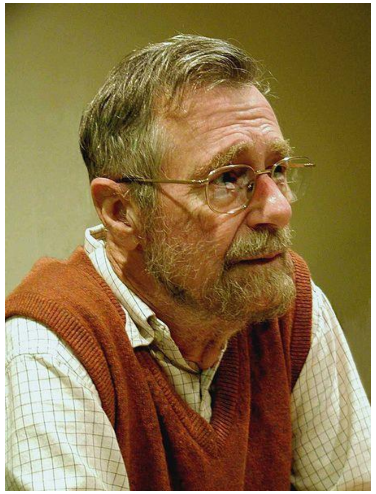
p	i	2^i
1	0	1
2	1	2
4	2	4
8	3	8

While-loop: proving an algorithm to calculate 2^n

```
// pre: n >= 0
int p=1; int i=0;
// invariant: p == 2^i
// bound function: n-i >= 0
while (i != n) {
    p = 2*p; i = i+1;
}
// post: p == 2^n
```

p	i	2^i
1	0	1
2	1	2
4	2	4
8	3	8

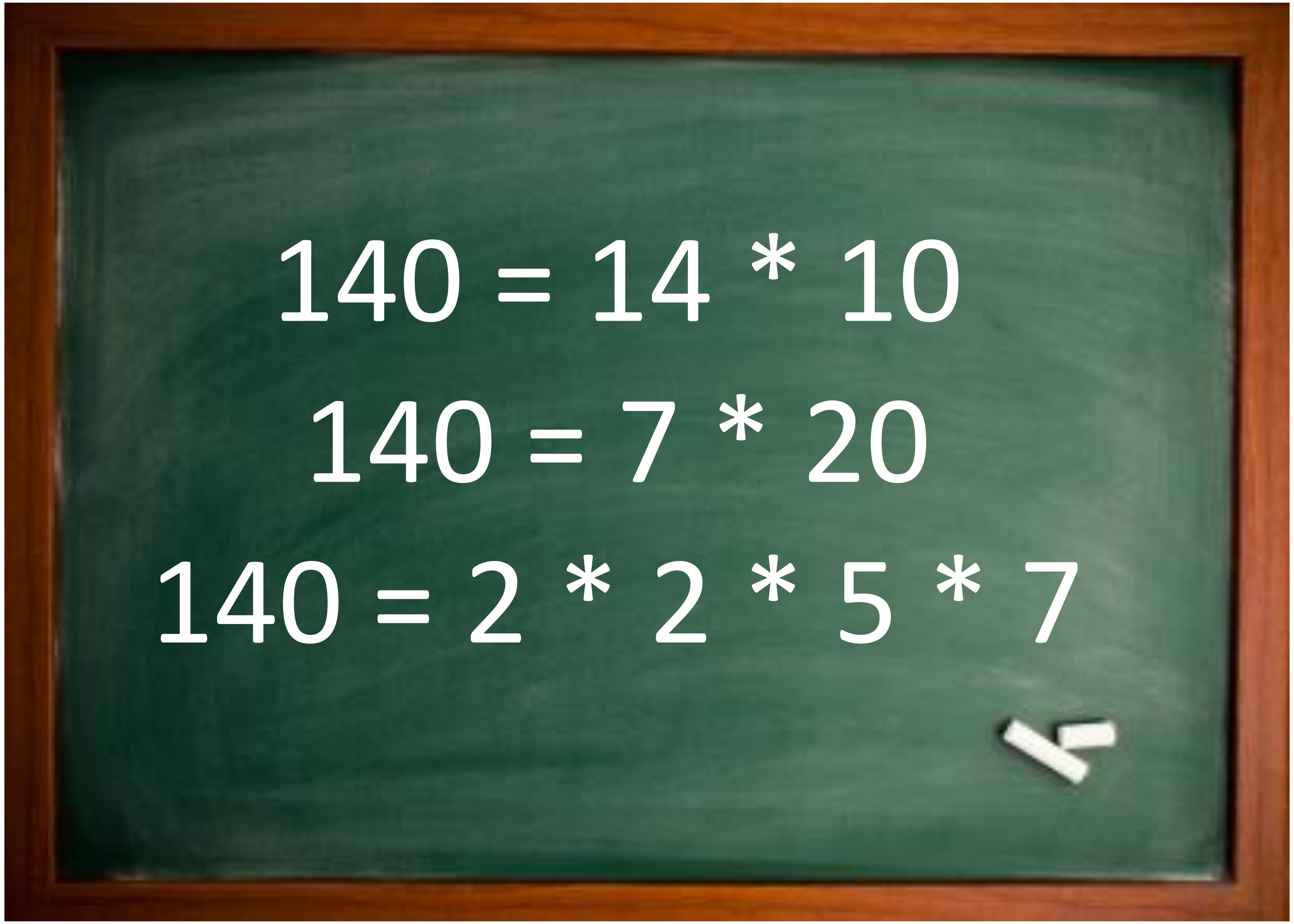
Edsger Wybe Dijkstra



program derivation:

to “develop proof and program hand in hand”

Deriving a solution to the Prime Factors Kata



Deriving a solution to the Prime Factors Kata

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

Deriving a solution to the Prime Factors Kata

q == 140

n	factors
2 * 2 * 5 * 7	[]
2 * 2 * 5	[7]
2 * 5	[2, 7]
5	[2, 2, 7]
1	[2, 2, 5, 7]

```
Invariant 1: factors contains only primes
Invariant 2: n * "product of factors" == q

Bound function: n >= 1
```

Deriving a solution to the Prime Factors Kata

```
// q >= 1
int n = q;
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 q.
```

```
Invariant 1: factors contains only primes
Invariant 2: n * "product of factors" == q

Bound function: n >= 1
```

Deriving a solution to the Prime Factors Kata

n	factors	c
2 * 2 * 3 * 3 * 5	[]	2
3 * 3 * 5	[2, 2]	3
5	[2, 2, 3, 3]	4, 5
1	[2, 2, 3, 3, 5]	6, 7, 8, …

q == 180

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

Deriving a solution to the Prime Factors Kata

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

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

Bound function: q + 1 - c >= 0
```

Deriving a solution to the Prime Factors Kata

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

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

Bound function: q + n - c >= 0
```

Deriving a solution to the Prime Factors Kata

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

If c is not prime,
then c == p_1 * … * p_m
where p_1 … p_m are primes.
Thus p_1 < c and p_1 is a prime factor of q.
This contradicts invariant 3.
Therefore c is prime.

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

Bound function: q + n - c >= 0
```

Deriving a solution to the Prime Factors Kata

```
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;
  }
}
```

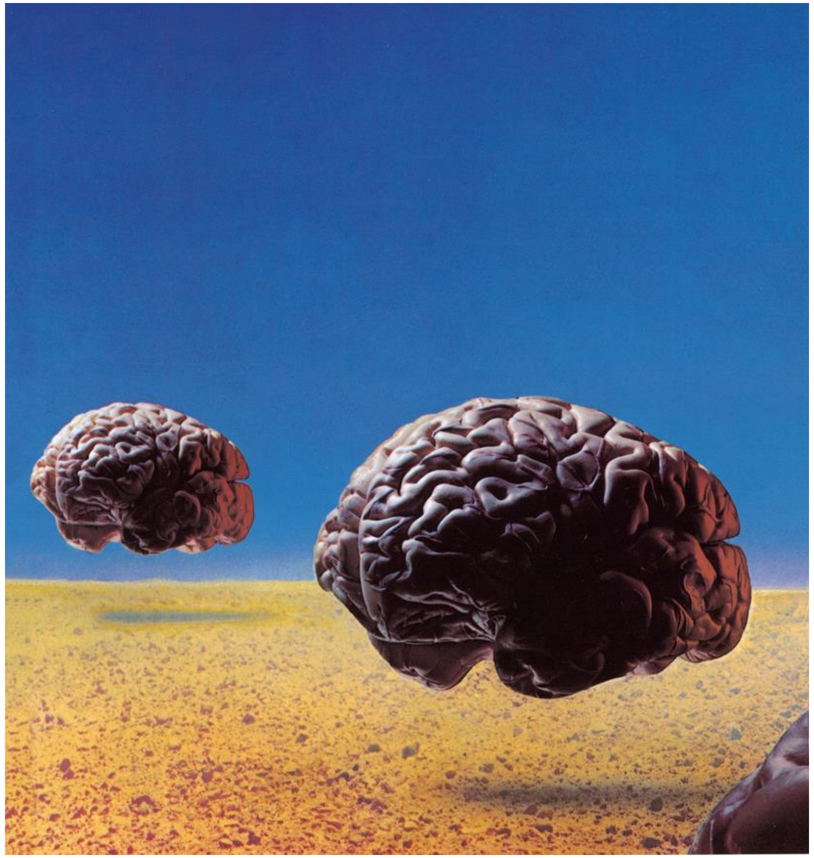
Now we **proved** it is correct

Conclusions

- Test-Driven Design
  - No guarantee that TDD leads to correct code
  - Empirical proof
  - Tests are repeatable
  - Tests run fast
- Program derivation (Dijkstra style)
  - Mathematical proof of correctness
  - Deriving and proving can go hand in hand
  - Time consuming
  - What if proof is wrong?

Conclusions

- TDD and formal proofs are tools, not goals
- The goal is correct code covered by tests



Questions?

www.hightechict.nl
sander.kooijmans@hightechict.nl

www.gogognome.nl