Lightning Talks

ACCU 2012

Jonathan Wakely – Smarter Than The Average Pointer

Seb Rose – Transformation Priority Premise

Aaron Ridout – Advocating References

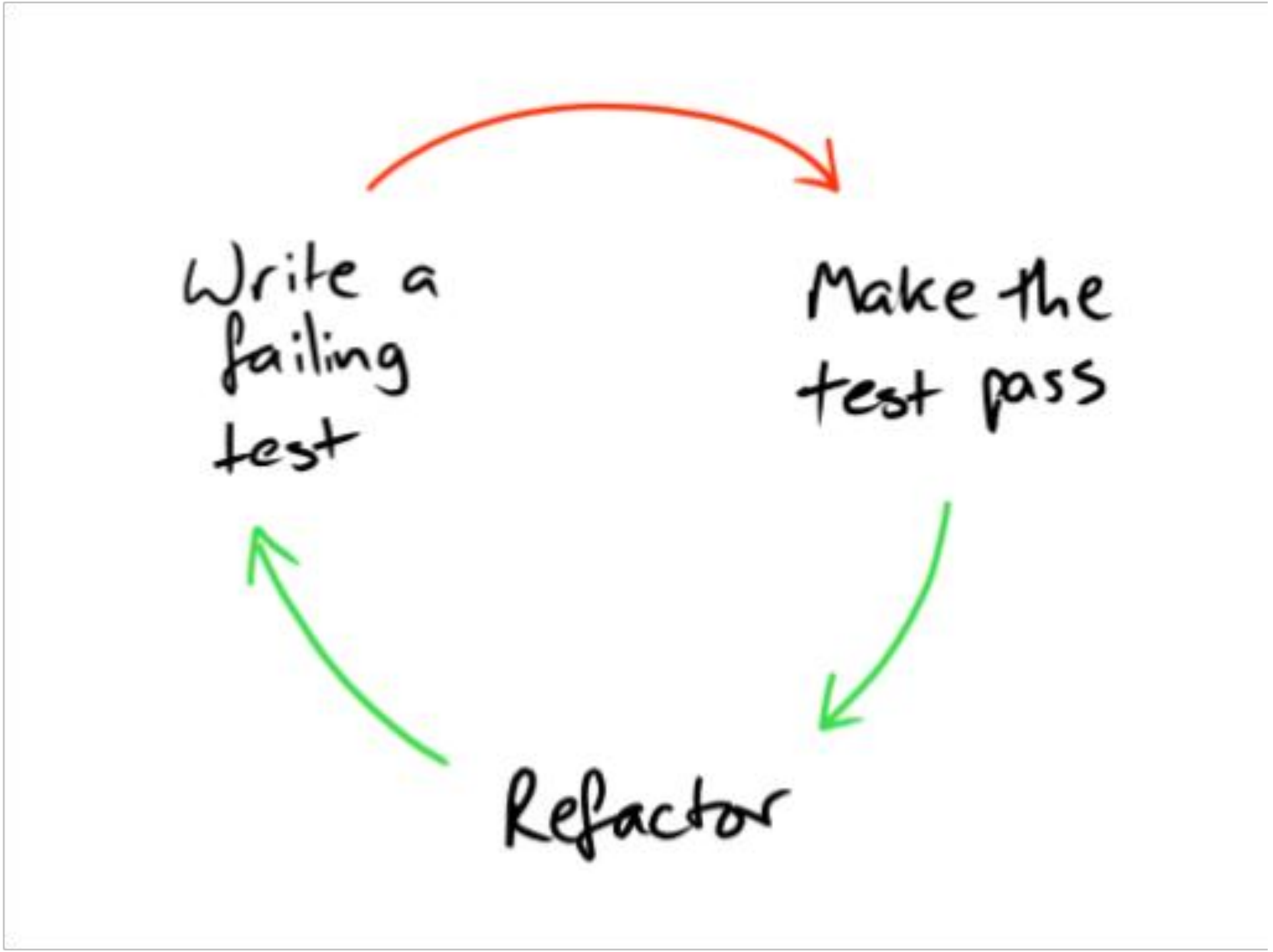
Mike Long - Metricide

Anna-Jayne Metcalfe – Don't Let The Big Ball of Mud Sneak Up on You

Roger Orr – Code Critiques

Pete Goodliffe – The C++ Cathedral & The Bizarre

Peter Sommerlad – C++'s “hello, world” Considered Harmful


Write a failing test
Make the test pass
Refactor

Uncle Bob's 3 rules

Over the years I have come to describe Test Driven Development in terms of three simple rules.

They are:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

```
@Test
public void gutterGame()
{
      for (int i=0; i<20; i++)
            game.roll(0);

    assertEquals(0, game.score());
}


public int score()
{
    return 0;
}
```

Transformations

- *Transformations* are simple operations that change the behavior of code.
- *Transformations* can be used as the sole means for passing the currently failing test in the red/green/refactor cycle.
- *Transformations* have a priority, or a preferred ordering, which if maintained, by the ordering of the tests, will prevent impasses, or long outages in the red/green/refactor cycle.

Impasse

“We pose a test only to find that we don’t know how to solve it without changing a large amount of code.”

1. **({}–>nil)** no code at all->code that employs nil
2. **(nil->constant)**
3. **(constant->constant+)** a simple constant to a more complex constant
4. **(constant->scalar)** replacing a constant with a variable or an argument
5. **(statement->statements)** adding more unconditional statements.
6. **(unconditional->if)** splitting the execution path
7. **(scalar->array)**
8. **(array->container)**
9. **(statement->recursion)**
10. **(if->while)**
11. **(expression->function)** replacing an expression with a
12. **(variable->assignment)** replacing the value of a variable.

Priority Premise

“… if you choose the tests and implementations that employ transformations that are higher on the list, you will avoid the impasse.”

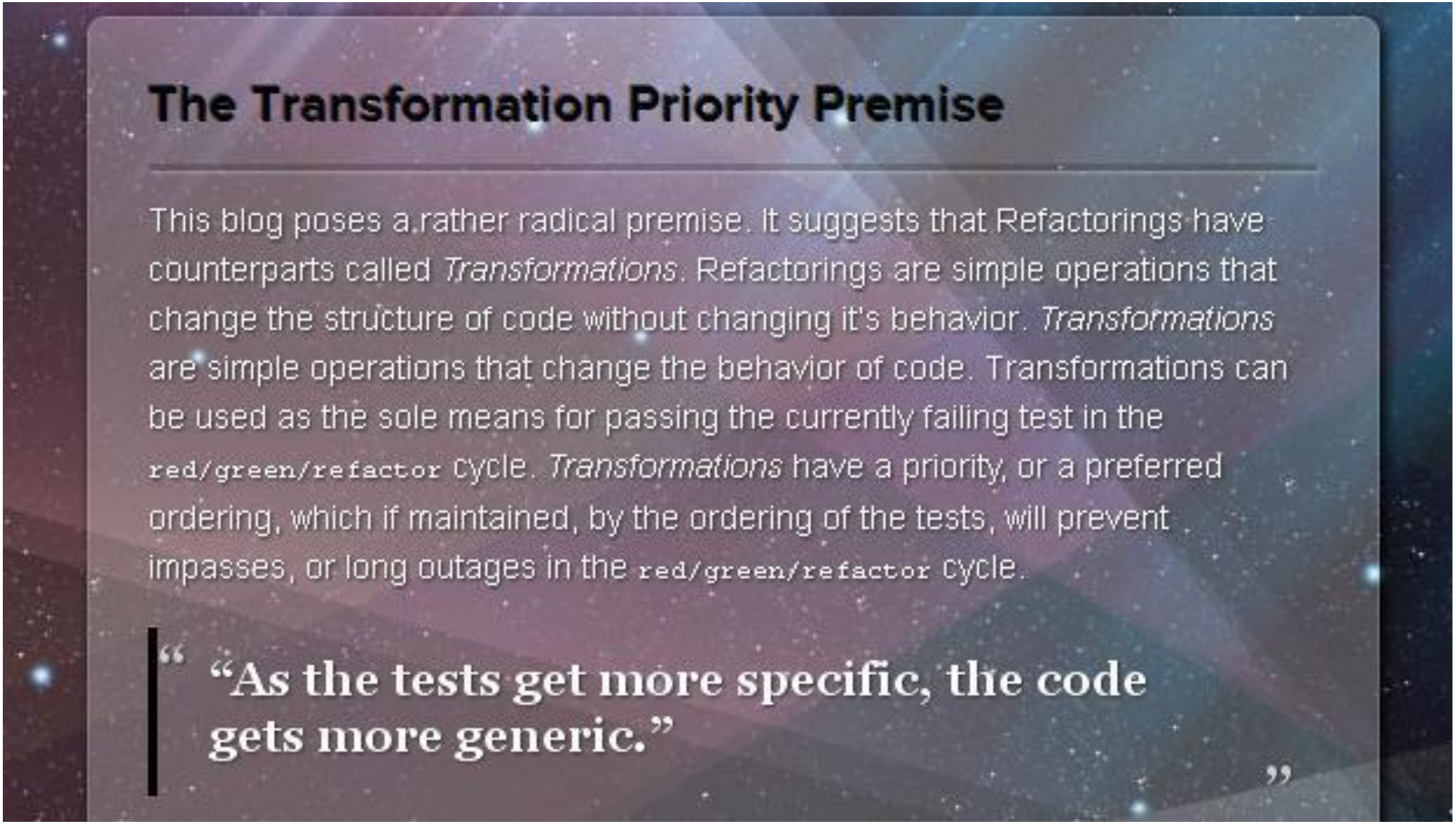
Uncle Bob Says…

“It is better (or simpler) to change a constant into a variable than it is to add an if statement. So when making a test pass, you try to do so with transformations that are simpler (higher on the list) than those that are more complex.”

“What’s more, when you pose a test, you try to pose one that allows simpler transformations rather than complex transformations; since the more complexity required by the test the larger the risk you take to get that test to pass.”

A guideline in the process of stepwise refinement should be the principle to decompose decisions as much as possible, to untangle aspects which are only seemingly interdependent, and to defer those decisions which concern details of representation as long as possible.

Wirth, N.: Program development by stepwise refinement.
Comm. ACM 14(4) (1968) 221–227.



http://cleancoder.posterous.com/the-transformation-priority-premise