Business Connect 2017

Bluefruit®

# TDD driving Quality

(Software Development Process)

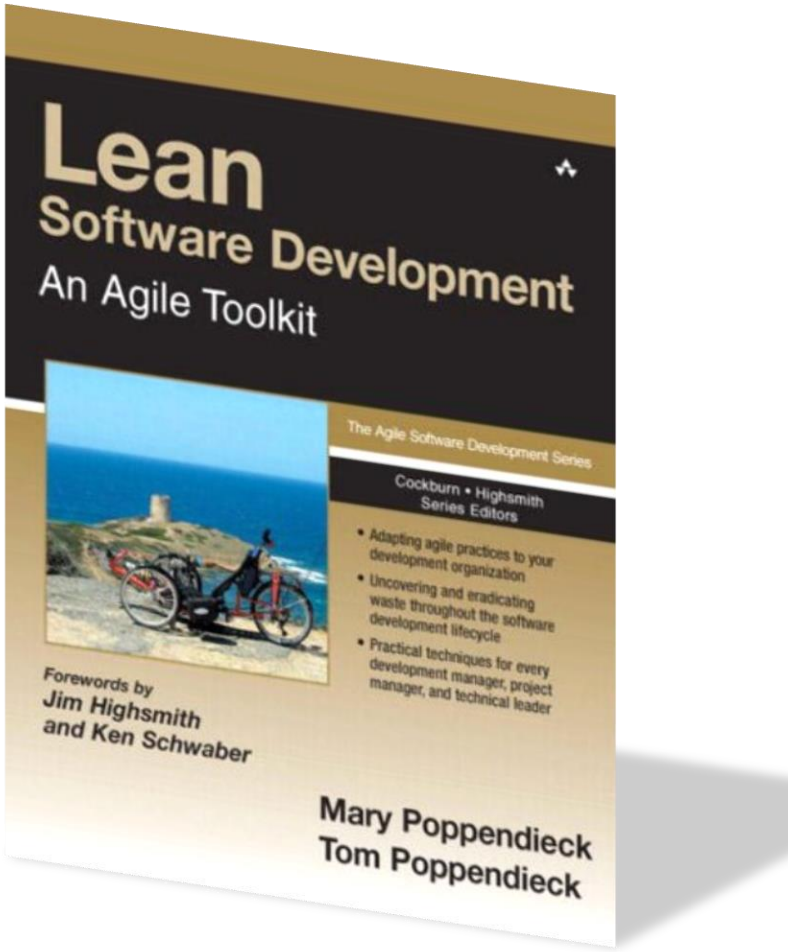
Byran Wills-Heath

Head of Development
Bluefruit Software

# Business Connect 2017

Bluefruit®

- Bluefruit established in 2000
- Embedded Software Specialists
- Clients in Automotive, Aerospace, Scientific Instruments, Consumer Goods etc.
- Strong Quality focus
- Agile since 2009
- Influenced by Lean-Agile



# Products we've worked on

Bluefruit®




# Business Connect 2017

Bluefruit®

- **What** quality means to us
- **Why** we believe it is important
- **How** we use TDD to improve quality

# What Quality means to us

Bluefruit®



# What Quality means to us

Bluefruit®

# What Quality means to us

Bluefruit®

How it "feels"

What it delivers

How it is built

# What Quality means to us

Bluefruit®

User Experience

We call what the user experiences 'Perceived Integrity'*.

How it is built

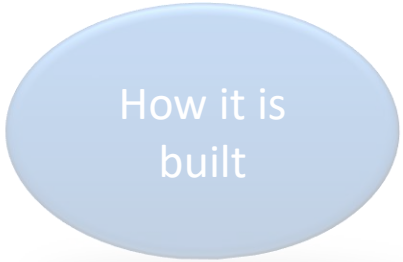
The way it is built is called 'Conceptual Integrity'*.

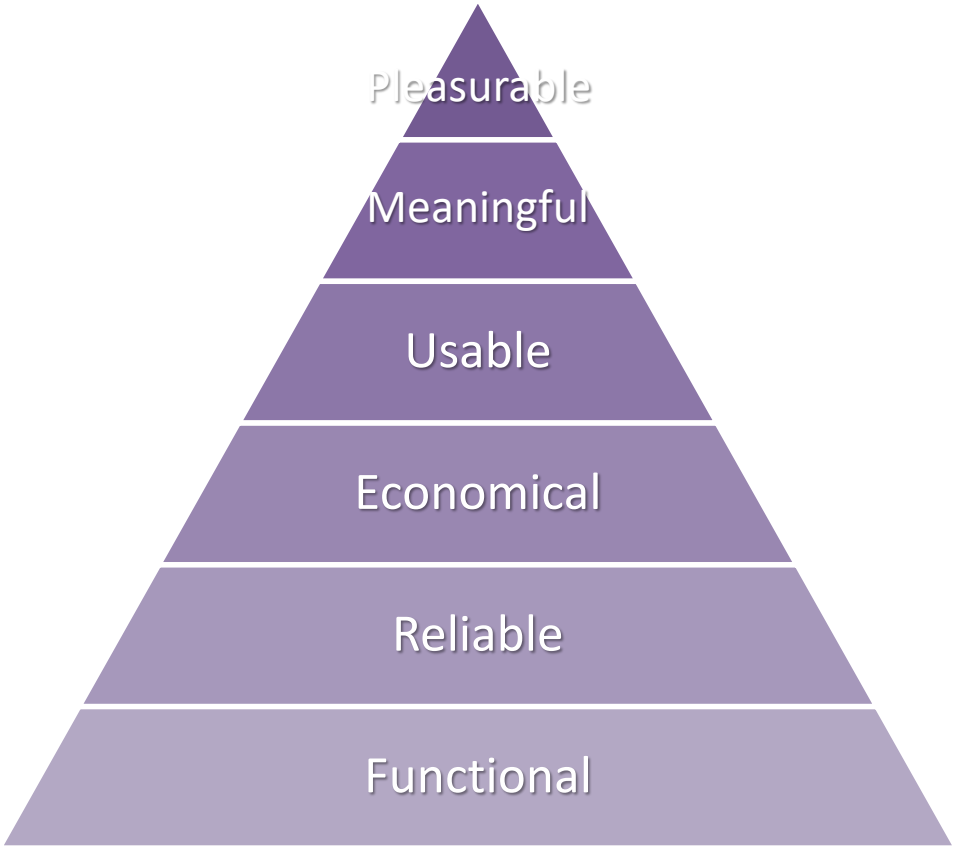
We believe these two concepts are what makes up and define the true meaning of quality in software.

*Mary Poppendeick's 'Lean Software Development: An Agile Toolkit'

# What - Perceived Integrity

Bluefruit®

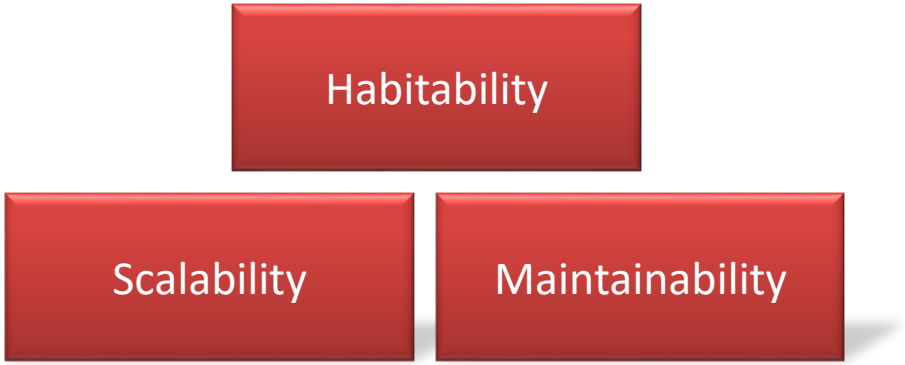



# What - Conceptual Integrity

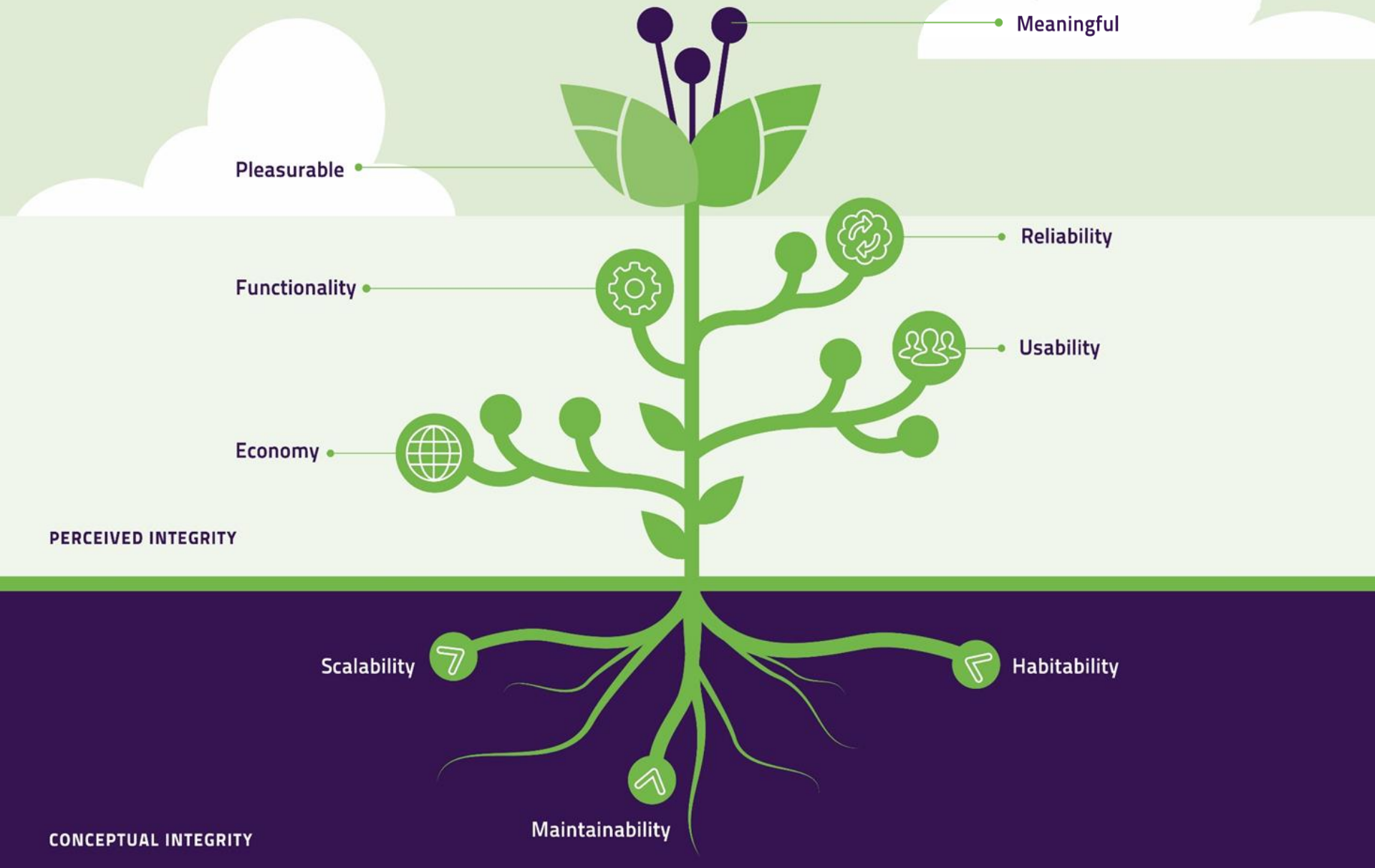
Bluefruit®


Conceptual Integrity includes the elements that are going on beneath the User Experience, but also include things that the end user will never see or engage with.

# What – Cultivating Quality

Bluefruit®

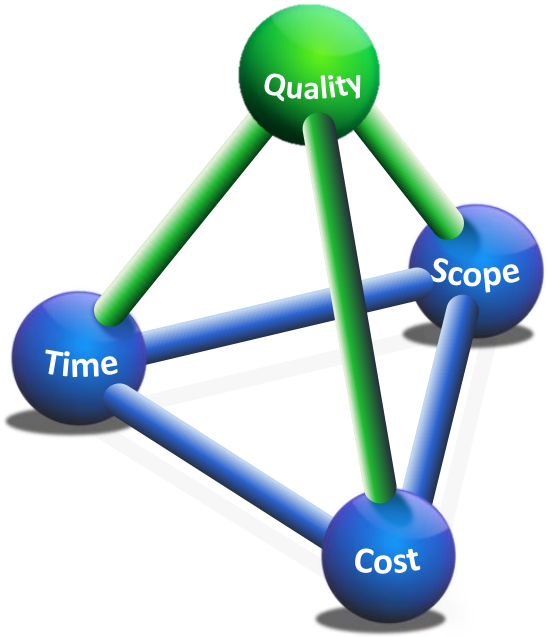


# Why is Quality Important?

- Our vision:
  - Happy Customers
  - Happy Workforce
  - Successful Projects

- Quality is a strategy, not a tactic

# Why is Quality Important?




Habitability

Scalability

Maintainability

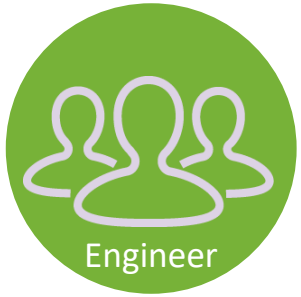
# Why is Quality Important?

Bluefruit®

"Always code as if the person who ends up maintaining your code will be a violent psychopath who knows where you live"

John F. Woods

# How is Quality achieved?

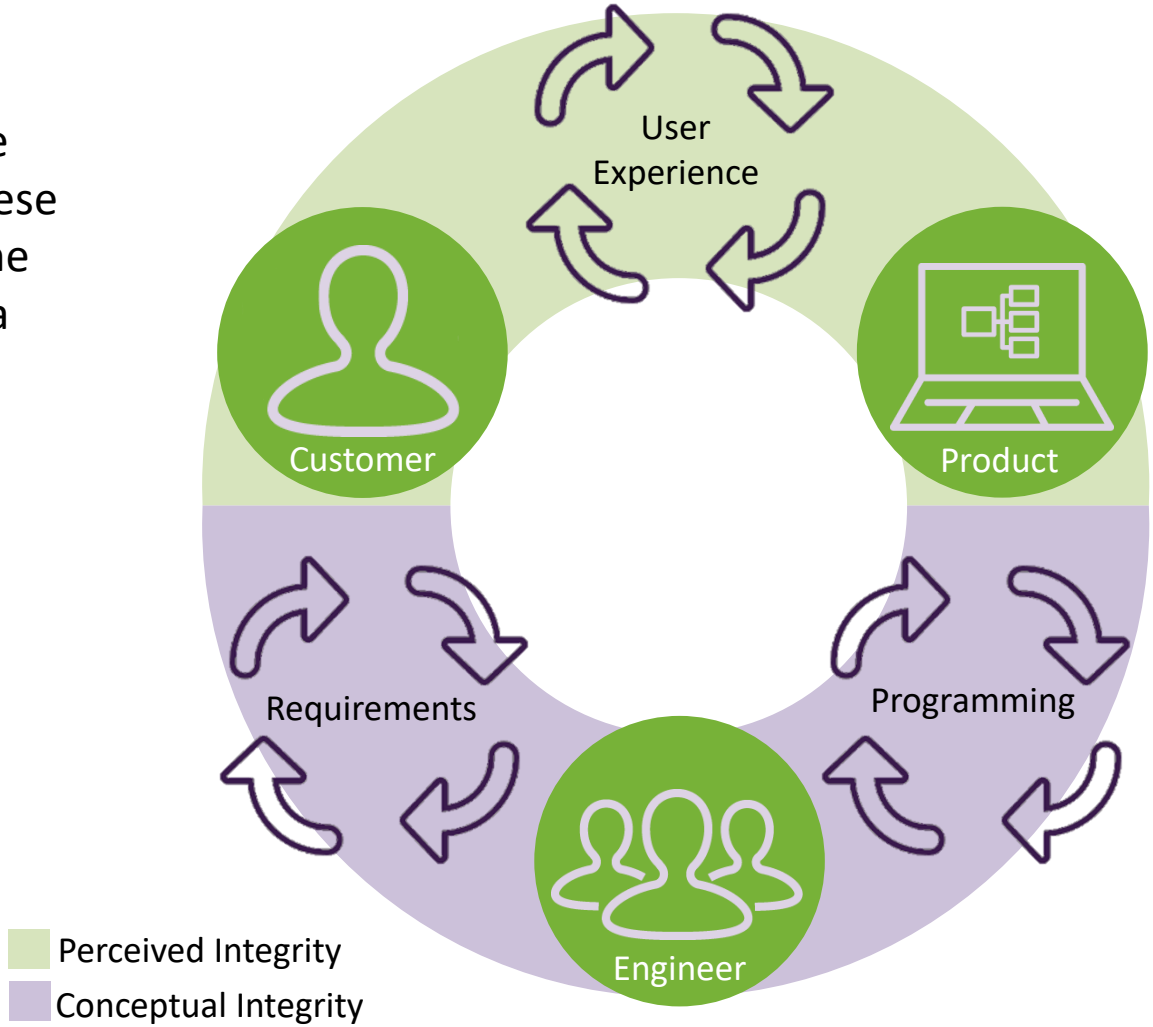





We use the Agile Toolkit to create feedback loops

# How - The Quality Wheel

- The effectiveness of the interaction between these constituents is key to the successful outcome of a Quality Solution



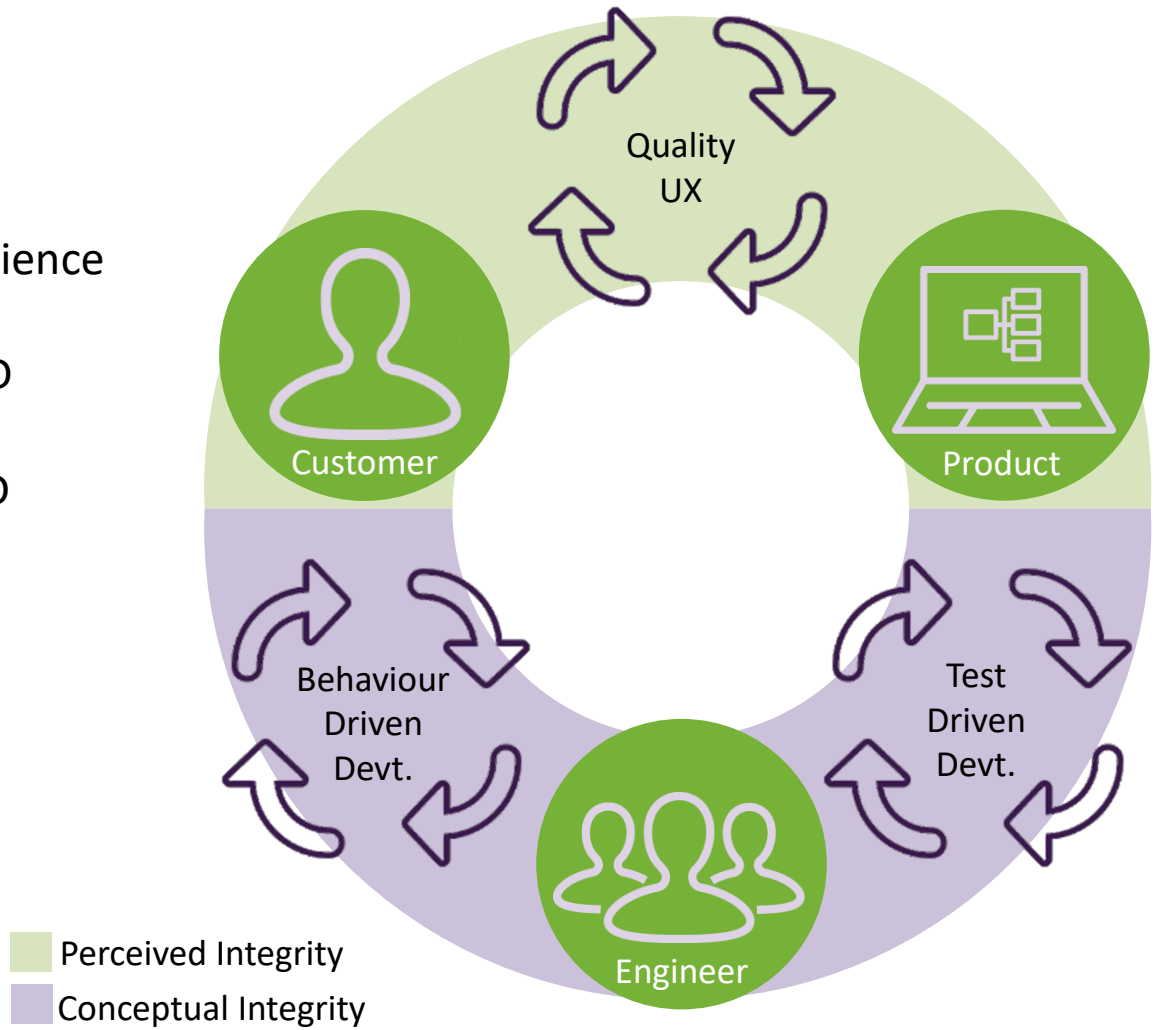
# How - The Quality Wheel

How Agile Helps

Goal: A Quality User Experience

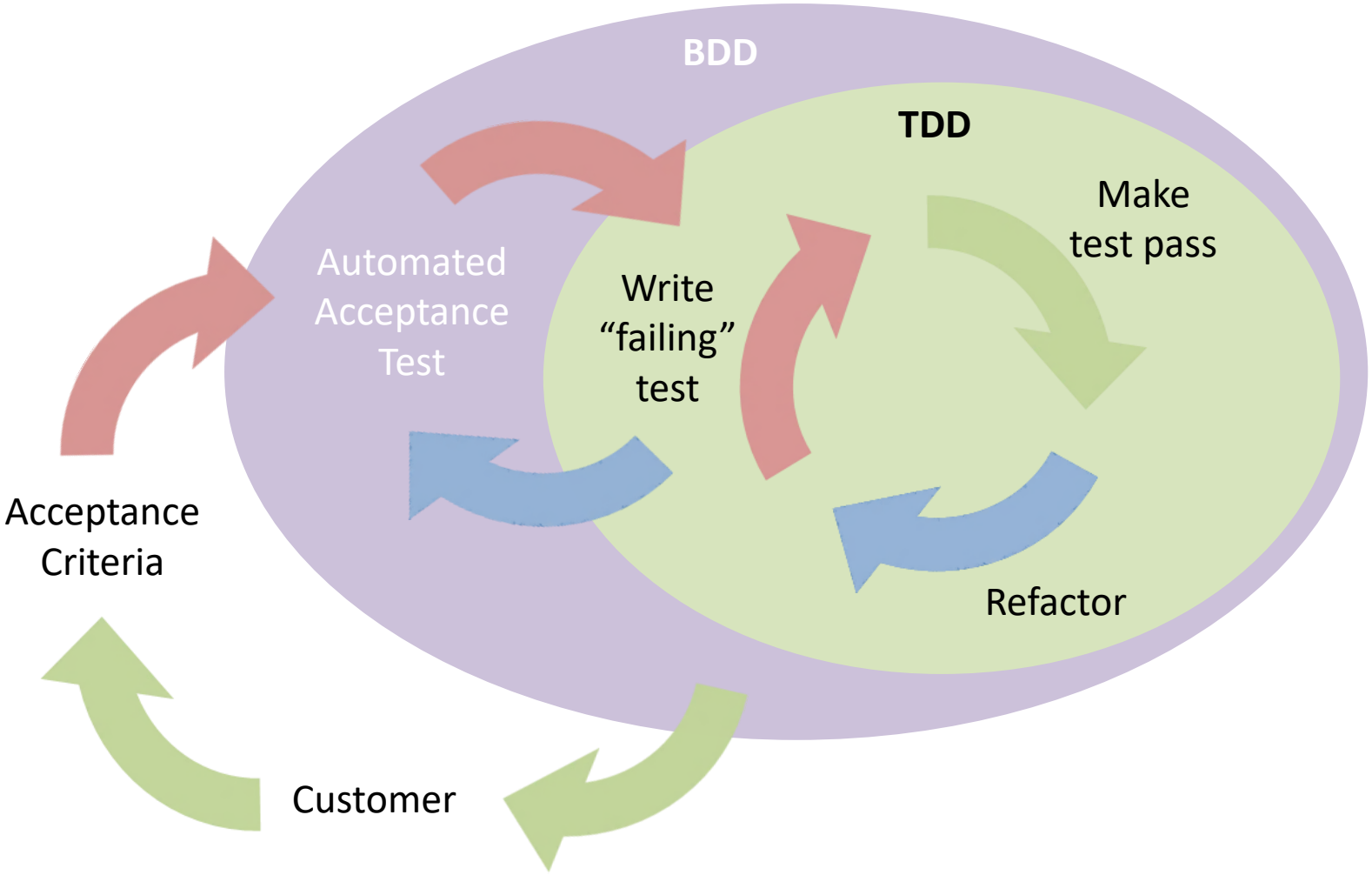
Build the Right Thing - BDD

Build the Thing Right - TDD



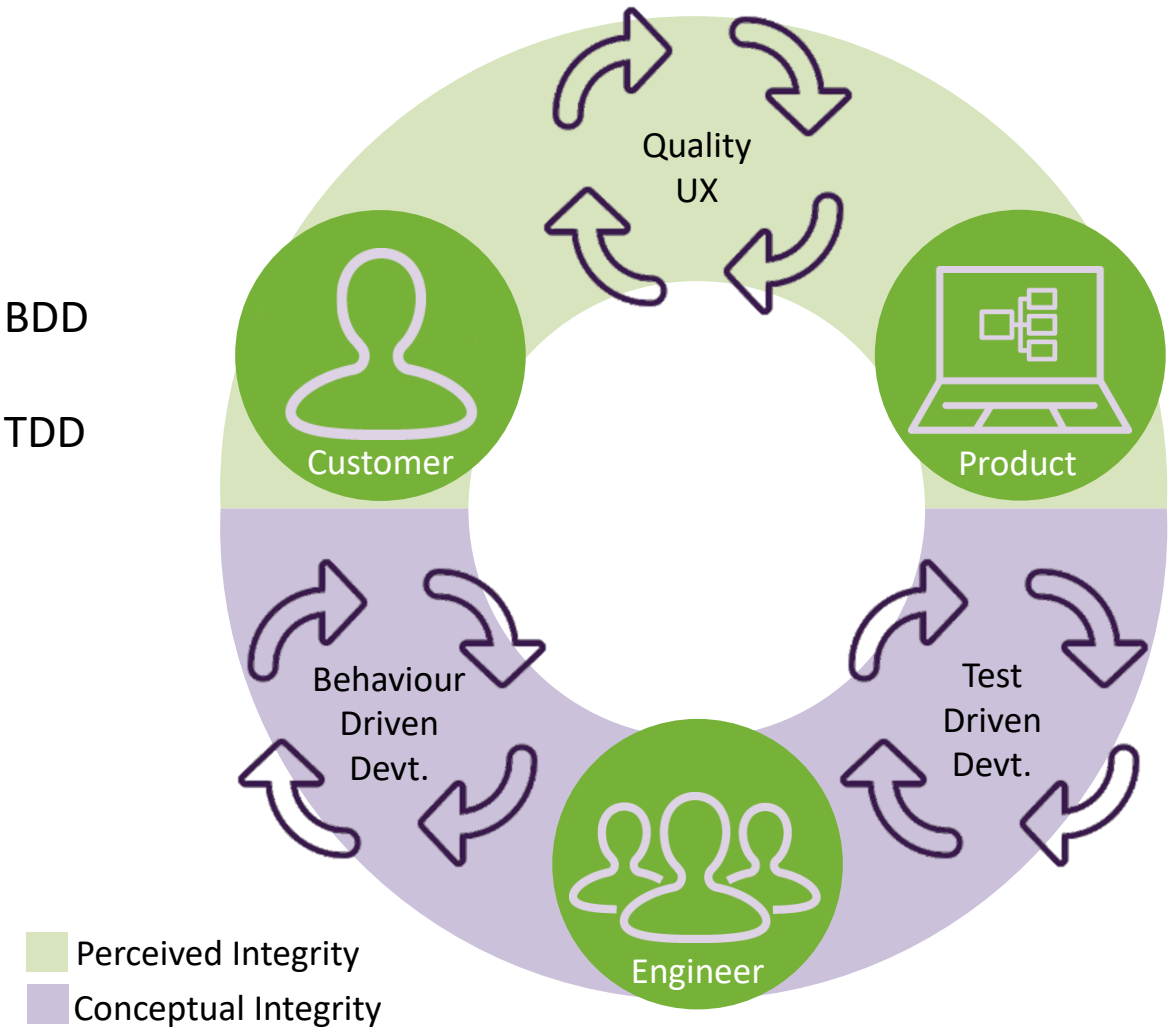
# How - TDD and BDD

Bluefruit®



# How - The Quality Wheel

- Goal: A Quality User Experience
- Build the Right Thing - BDD
- Build the Thing Right - TDD



# Test Driven Development

Bluefruit®

- **What** Test Driven Development is
- **Why** we believe TDD is important
- **How** we develop code using TDD

# What is TDD?

Bluefruit®

- Test Driven Development/Design is a design process.
  - It's a robust way of designing software components (units) interactively so that their behaviour is specified through unit tests
- TDD is not about finding existing bugs.
  - It's a process that assists you in not introducing bugs in the first place

# Why we value TDD

Bluefruit®

- Check the code does what we expect it to (Functional & Reliable)

- Safety net (Scalability & Maintainability)
  - Allows the code to be extended and/or modified without breaking existing functionality
  - Allows team ownership

# Why we value TDD

Bluefruit®

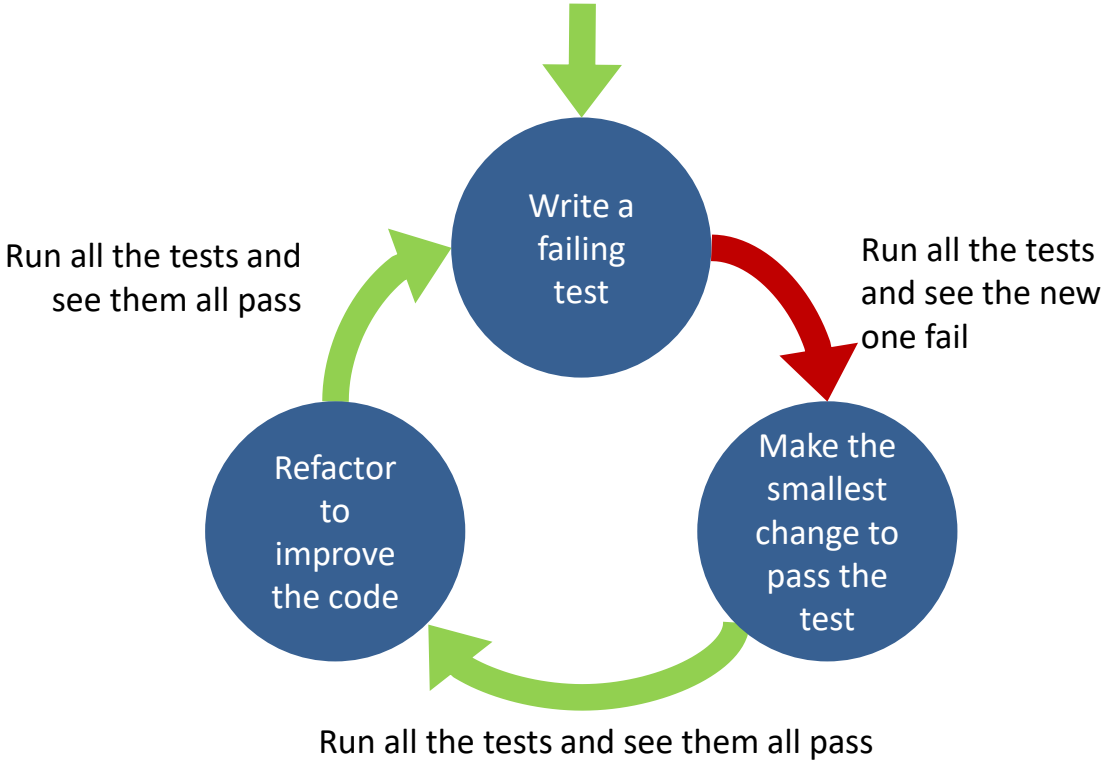
- Tests document the code (Habitability)
  - Inform other developers how the code works and how it can be used
- Better architecture (Scalability & Maintainability)
  - Separates interface thinking from implementation thinking

# Why we value TDD

Bluefruit®

- Encourages refactoring (Habitability)

# How – TDD Cycle



# How - TDD a quick example

Bluefruit®

Write a function that returns true or false depending on whether its input integer is a leap year or not.

A leap year is defined as one that is divisible by 4, but is not otherwise divisible by 100 unless it is also divisible by 400.

For example, 2001 is a typical common year and 1996 is a typical leap year, whereas 1900 is an atypical common year and 2000 is an atypical leap year.

# How - Rules & Examples

Bluefruit®

Unit tests specify concrete examples of the rules for the expected behaviour

We create examples for each rule starting with the simplest possible example

# How - Rules & Examples

Bluefruit®

Unit tests specify concrete examples of the rules for the expected behaviour

We create examples for each rule starting with the simplest possible example

A leap year is defined as:

- A year that is divisible by 4, but is not divisible by 100

or

- A year that is divisible by 400

# Our first test

Red
Green
Refactor

## Tests

```
def test_A_typical_common_year_returns_false
  assert_equal(false, is_a_leap_year(2001), '2001 is not a leap year')
end
```

## Code

```
def is_a_leap_year(year)
  true
end
```

## Result

```
1) Failure: TestHiker#test_A_typical_common_year_returns_false [test_leap_years.rb:7]:
2001 is not a leap year.
Expected: false
  Actual: true
1 runs, 1 assertions, 1 failures, 0 errors, 0 skips
```

# Making the smallest/simplest change to pass the first test

Red
Green
Refactor

## Tests

```
def test_A_typical_common_year_returns_false
  assert_equal(false, is_a_leap_year(2001), '2001 is not a leap year')
end
```

## Code

```
def is_a_leap_year(year)
  false
end
```

## Result

```
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

# Add second test for the first rule

Red

Green

Refactor

## Tests

```
def test_A_typical_common_year_returns_false
  assert_equal(false, is_a_leap_year(2001), '2001 is not a leap year')
end

def test_A_typical_leap_year_returns_true
  assert_equal(true, is_a_leap_year(1996), '1996 is a leap year')
end
```

## Code

```
def is_a_leap_year(year)
  false
end
```

## Result

```
1) Failure: TestHiker#test_A_typical_leap_year_returns_true [test_leap_years.rb:11]:
1996 is a leap year.
Expected: true
  Actual: false
2 runs, 2 assertions, 1 failures, 0 errors, 0 skips
```

# Making the smallest/simplest change to pass both tests

Red
Green
Refactor

## Tests

```
def test_A_typical_common_year_returns_false
  assert_equal(false, is_a_leap_year(2001), '2001 is not a leap year')
end

def test_A_typical_leap_year_returns_true
  assert_equal(true, is_a_leap_year(1996), '1996 is a leap year')
end
```

## Code

```
def is_a_leap_year(year)
  year % 4 == 0
end
```

## Result

```
2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

# Leap Year Rules

A leap year is defined as:

- A year that is divisible by 4, but is not divisible by 100

or

- A year that is divisible by 400

# Add third test for the first rule

Red

Green

Refactor

## Tests

```
...

def test_A_typical_leap_year_returns_true
  assert_equal(true, is_a_leap_year(1996), '1996 is a leap year')
end

def test_An_atypical_common_year_returns_false
  assert_equal(false, is_a_leap_year(1900), '1900 is not a leap year')
end
```

## Code

```
def is_a_leap_year(year)
  year % 4 == 0
end
```

## Result

```
1) Failure:TestHiker#test_An_atypical_common_year_returns_false [test_leap_years.rb:15]:
1900 is not a leap year.
Expected: false
  Actual: true
3 runs, 3 assertions, 1 failures, 0 errors, 0 skips
```

# Making the smallest/simplest change to pass all three tests

Red
Green
Refactor

## Tests

```
...

def test_A_typical_leap_year_returns_true
  assert_equal(true, is_a_leap_year(1996), '1996 is a leap year')
end

def test_An_atypical_common_year_returns_false
  assert_equal(false, is_a_leap_year(1900), '1900 is not a leap year')
end
```

## Code

```
def is_a_leap_year(year)
  year % 4 == 0 and not year % 100 == 0
end
```

## Result

```
3 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

# Refactor to make the code more readable

Red
Green
Refactor

## Tests

```
...

def test_A_typical_leap_year_returns_true
  assert_equal(true, is_a_leap_year(1996), '1996 is a leap year')
end

def test_An_atypical_common_year_returns_false
  assert_equal(false, is_a_leap_year(1900), '1900 is not a leap year')
end
```

## Code

```
def is_a_leap_year(year)
  (year % 4 == 0) and not(year % 100 == 0)
end
```

## Result

```
3 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

# Leap Year Rules

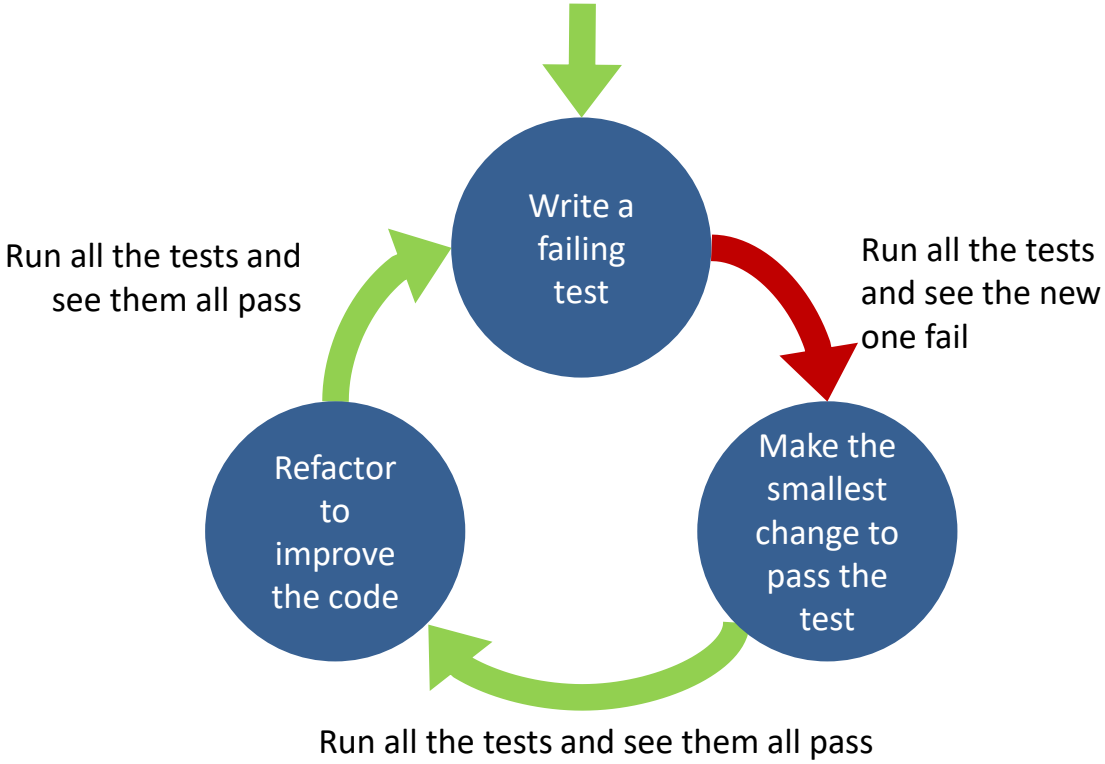
A leap year is defined as:

- A year that is divisible by 4, but is not divisible by 100

or

- A year that is divisible by 400

# How – TDD Cycle



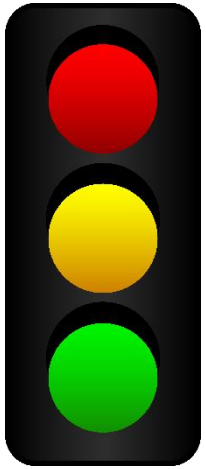
# Common problems with TDD

Bluefruit®

- Easy to pick up, difficult to master
  - Dependencies make testing more complicated
  - Creating tests that document code is difficult to begin with
- Initial development can take longer, but the overall development time will be reduced

# Test Driven Development

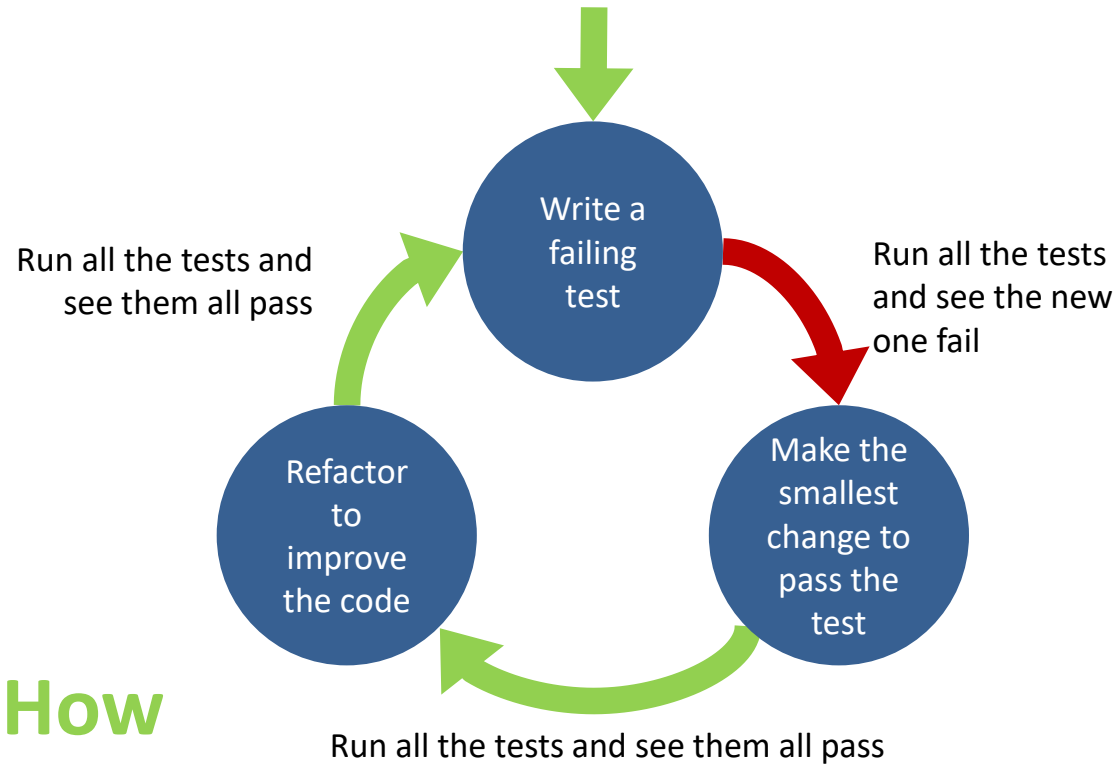
Bluefruit®



**What**

**Why**

- Habitability
- Scalability
- Maintainability

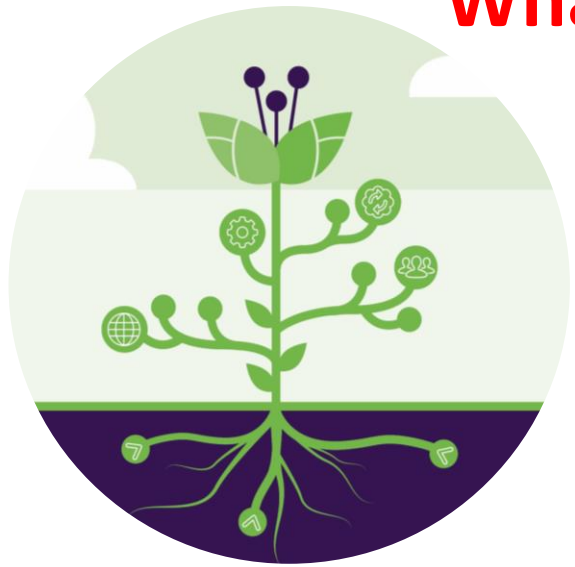



**How**

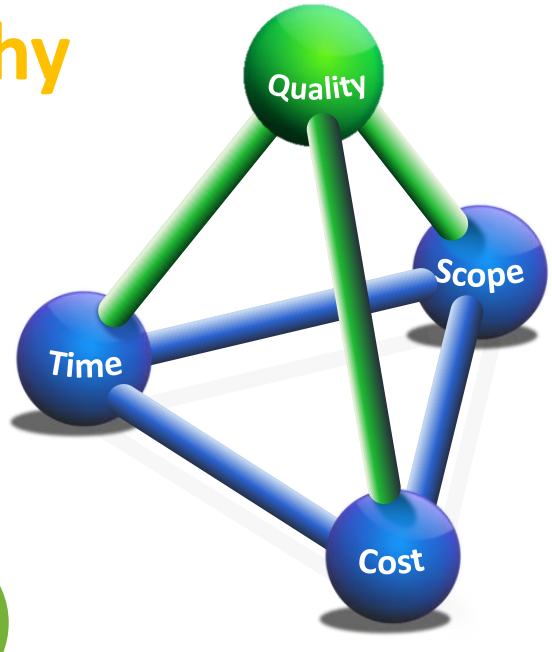
# Bringing it all together

Bluefruit®

**What**



**Why**




**How**