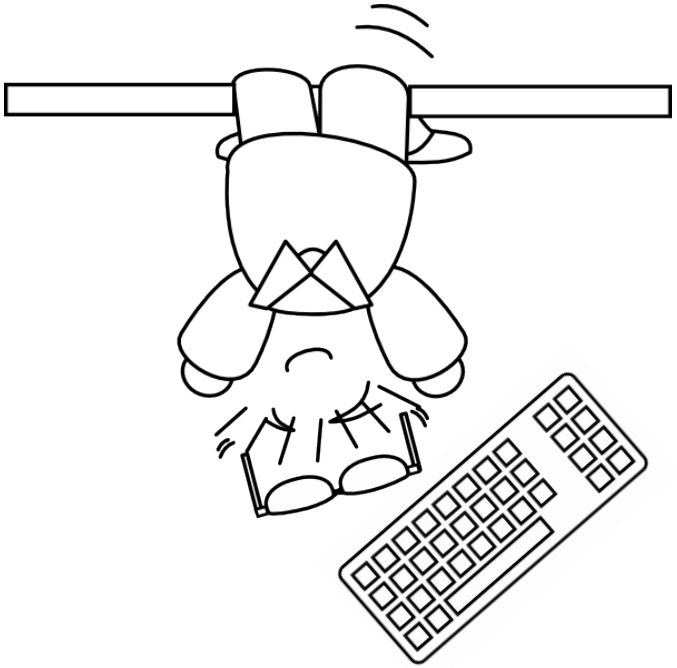
# Coding backwards
## in order to
# to think straight

2nd part



## Deepening TDD
### practice with
## Outside-In TDD

# TDD ...

# ... what's that again?

# *Two main « styles » of TDD*

# *Two main « styles » of TDD*

**Classical**

**Outside-in**

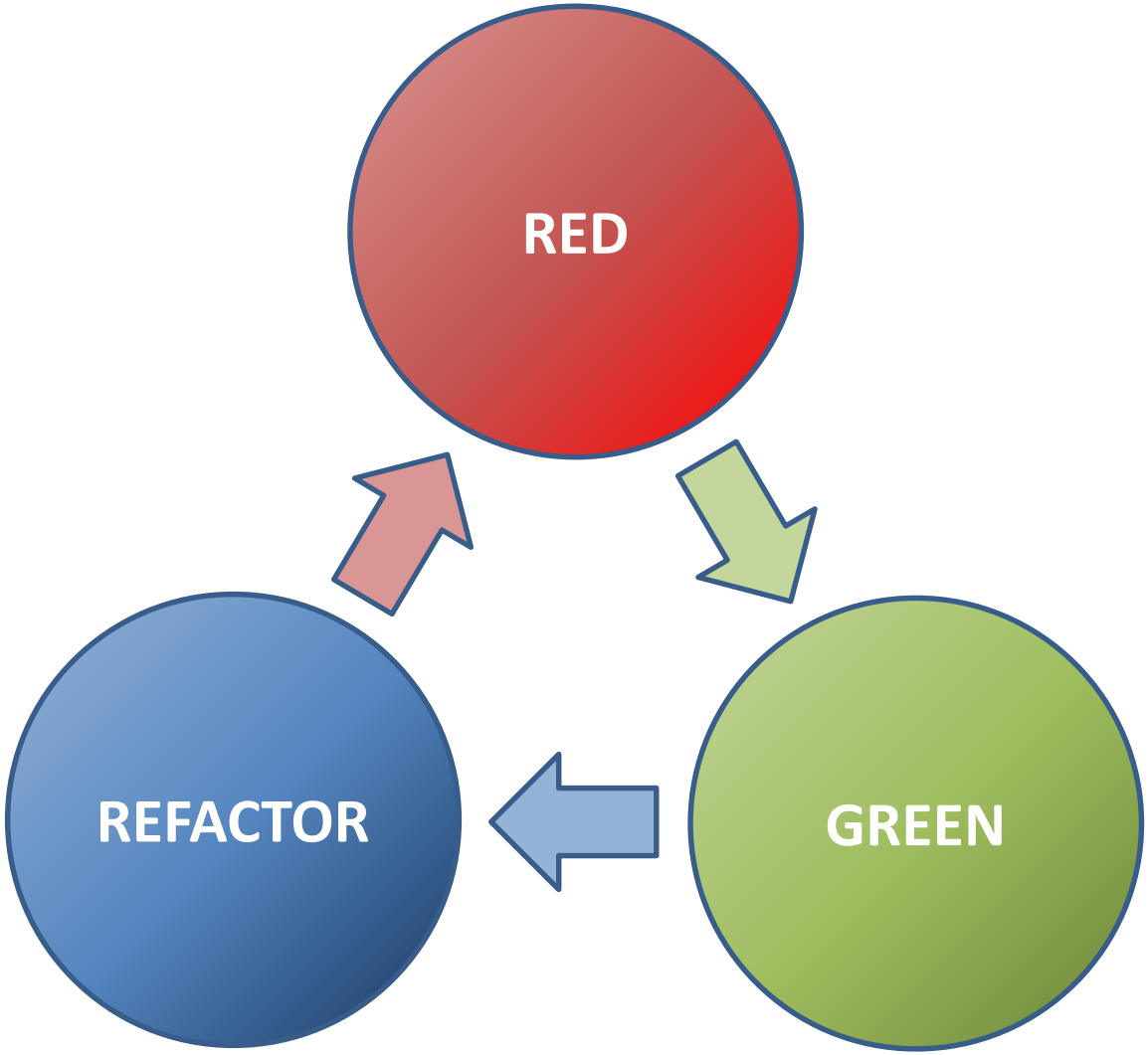
# *Two main « styles » of TDD*

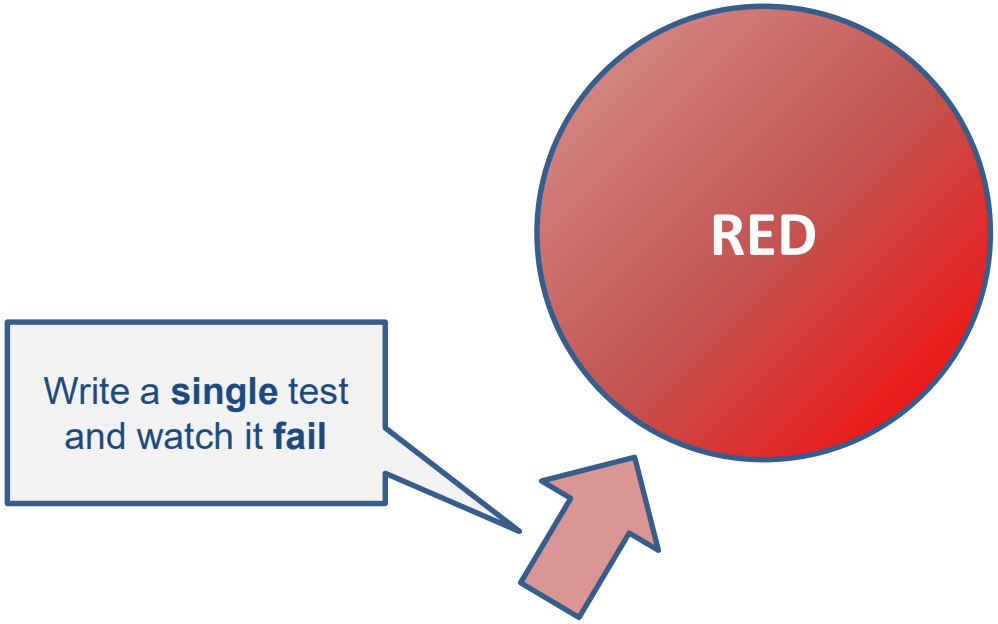
| | |
|---|---|
| « Detroit School » | **Classical** |
| « London School » | **Outside-in** |

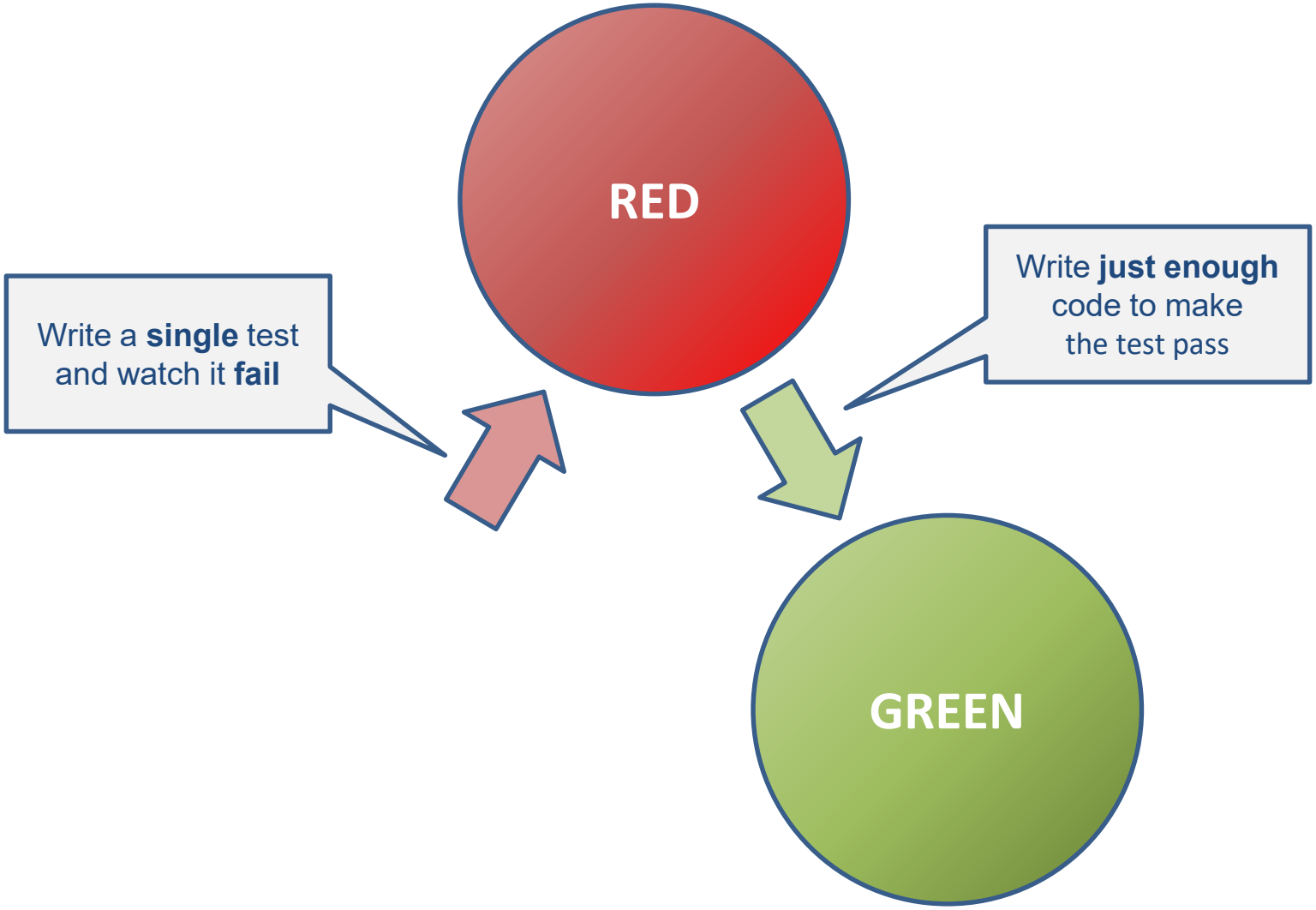
# *Two main « styles » of TDD*

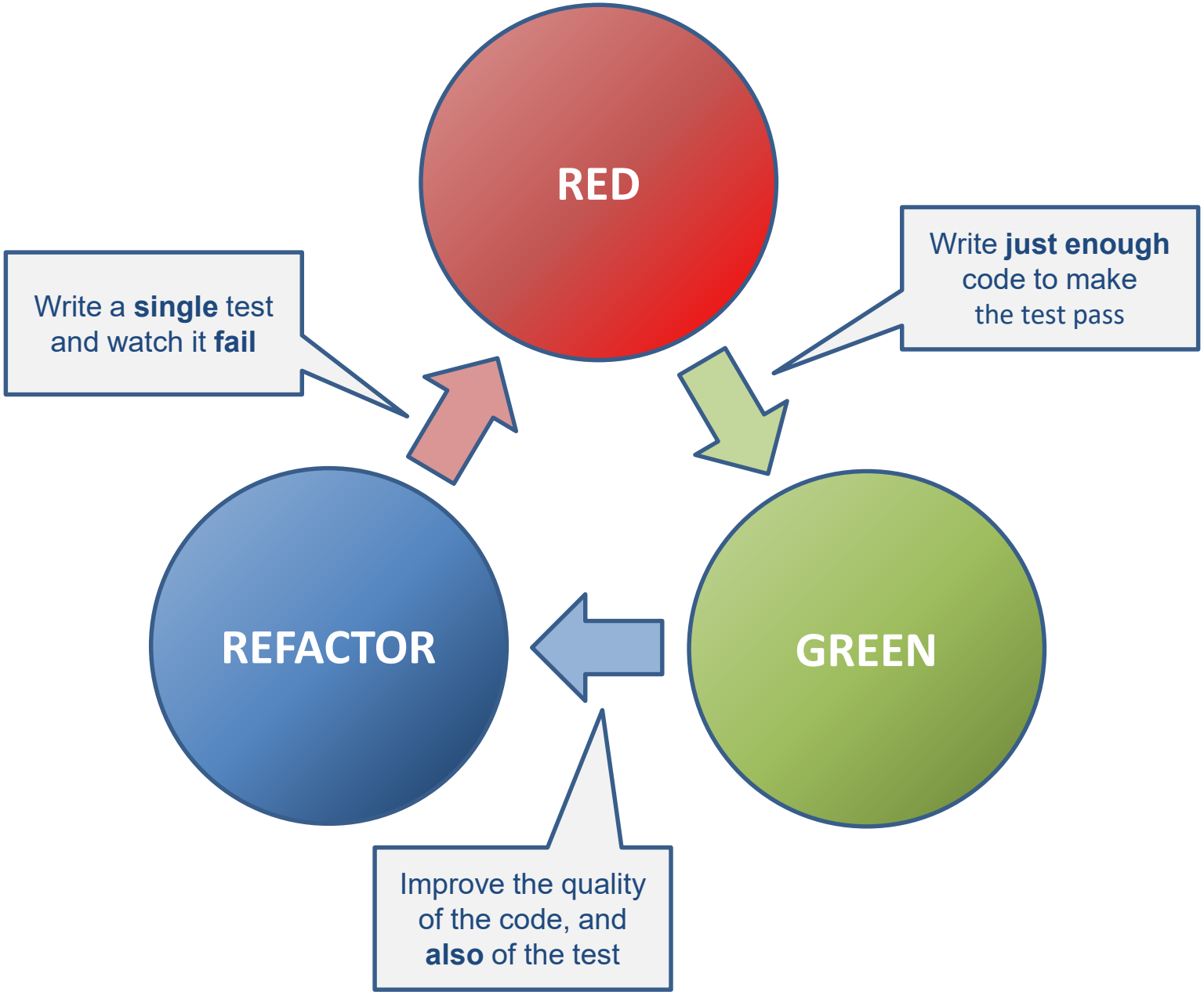
| | | |
|---|---|---|
| « Detroit School » | **Classical** | **= state verification** |
| « London School » | **Outside-in** | **= behaviour verification** |

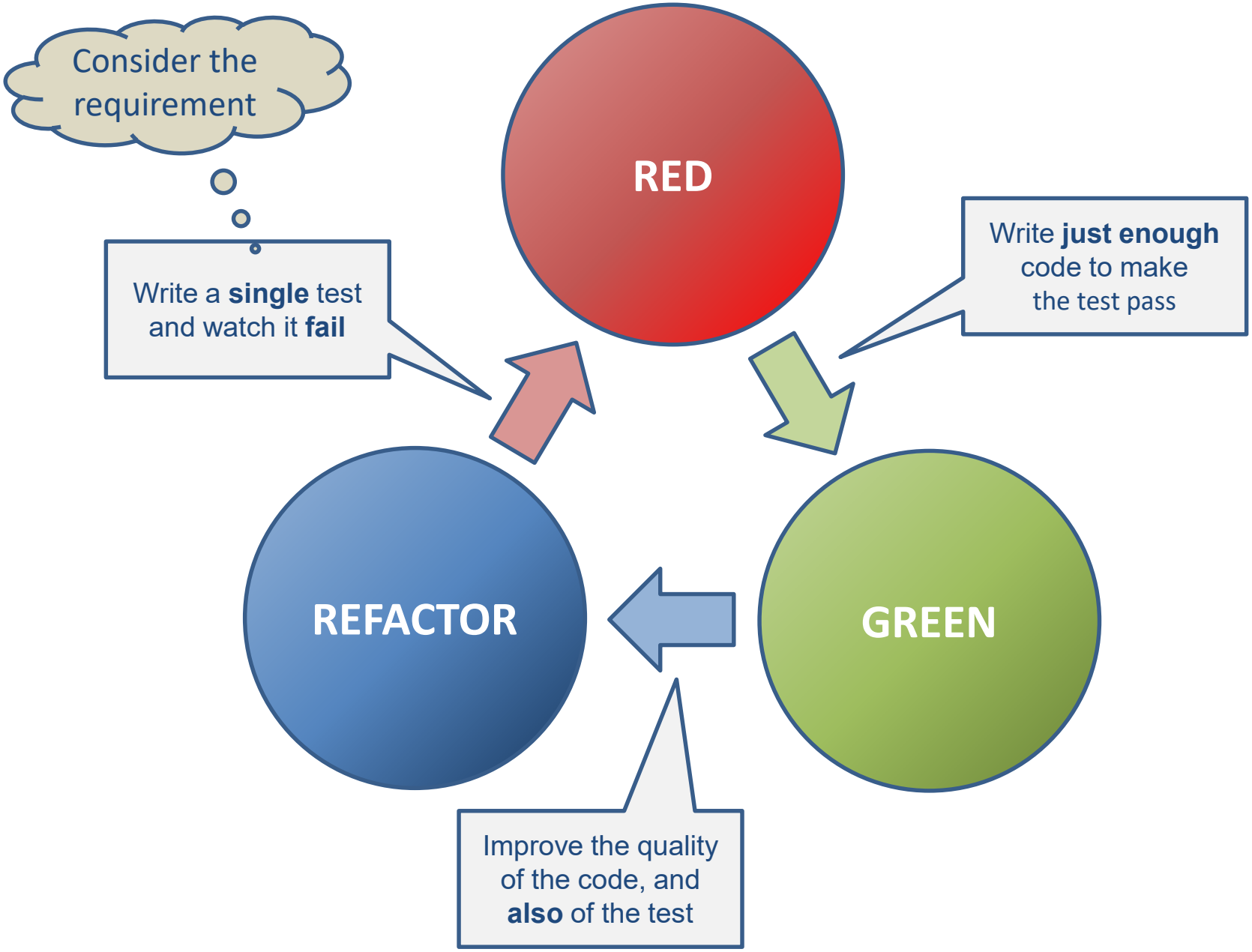
# « Classical » TDD

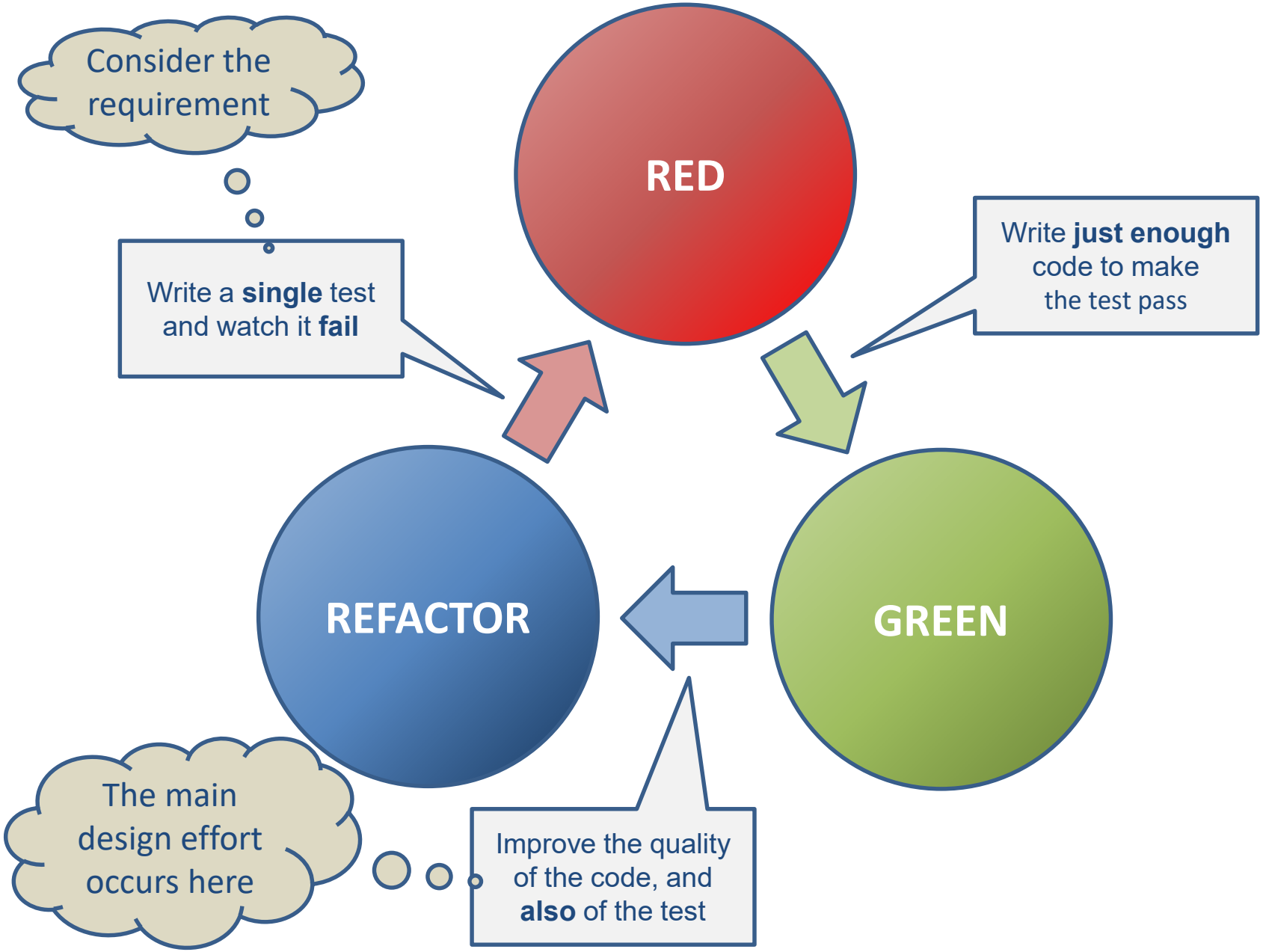

RED
GREEN
REFACTOR

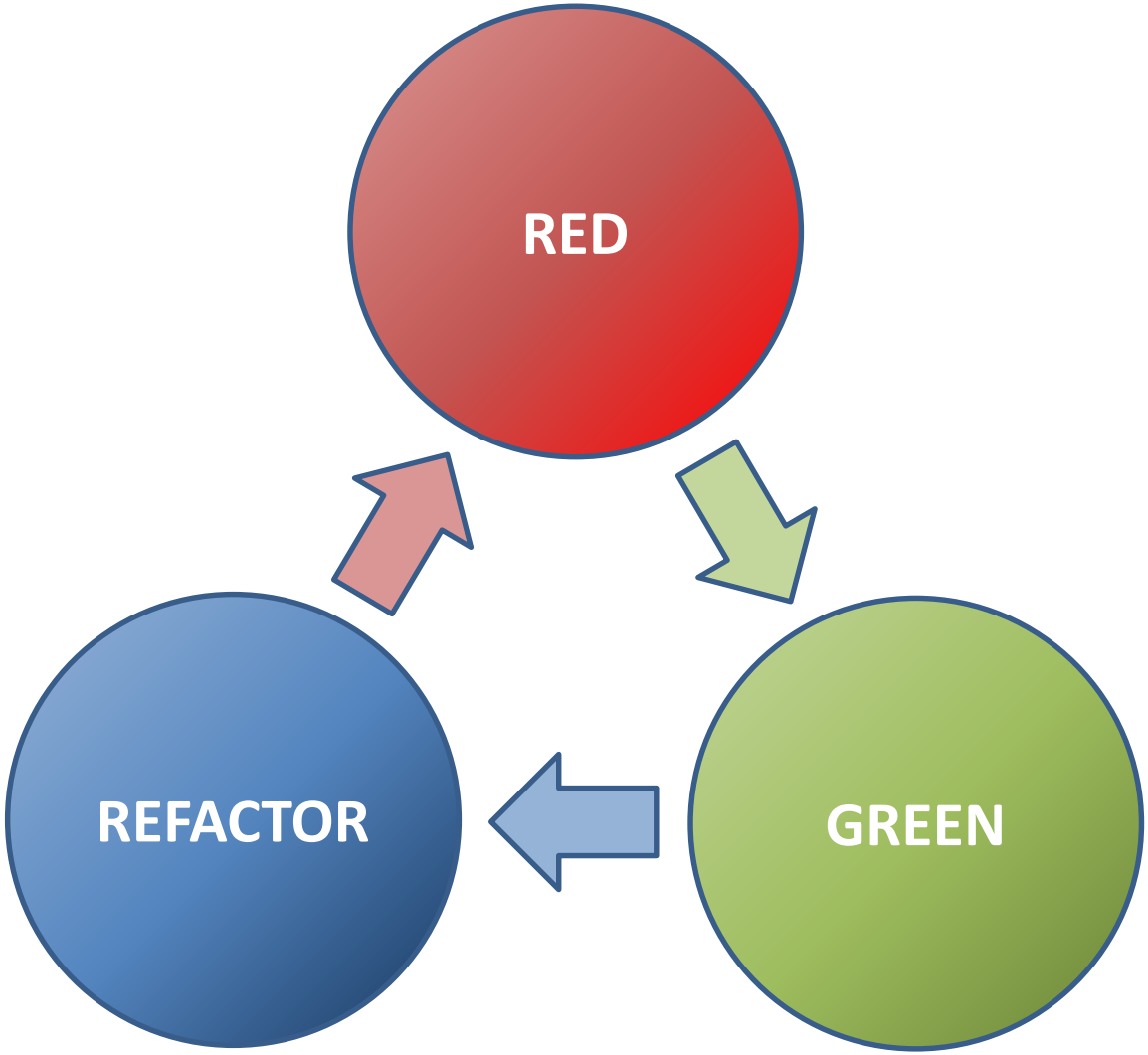

RED
Write a **single** test
and watch it **fail**


RED
Write a **single** test and watch it **fail**
Write **just enough** code to make the test pass
GREEN

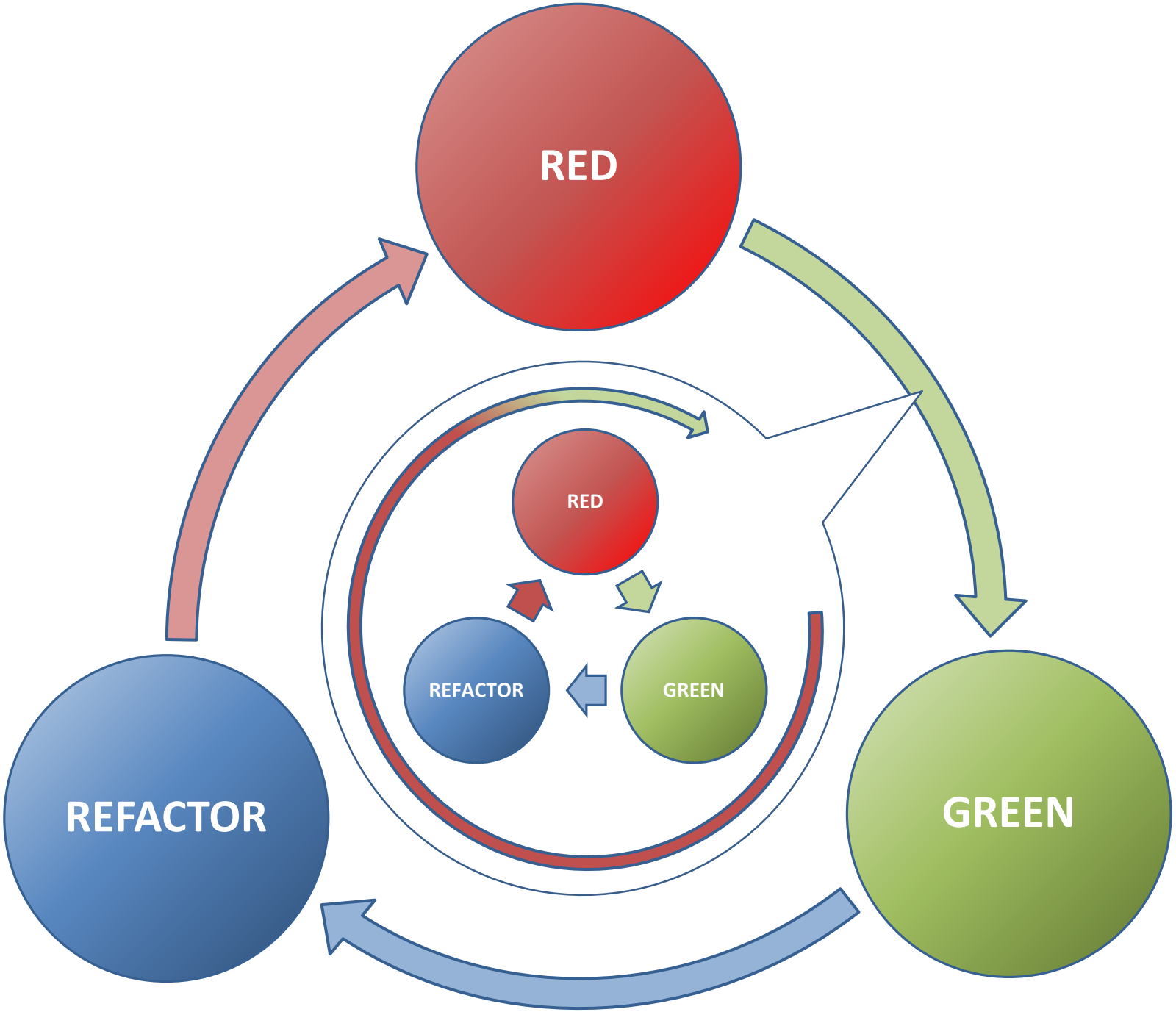

RED
Write **just enough** code to make the test pass
GREEN
Improve the quality of the code, and **also** of the test
REFACTOR
Write a **single** test and watch it **fail**

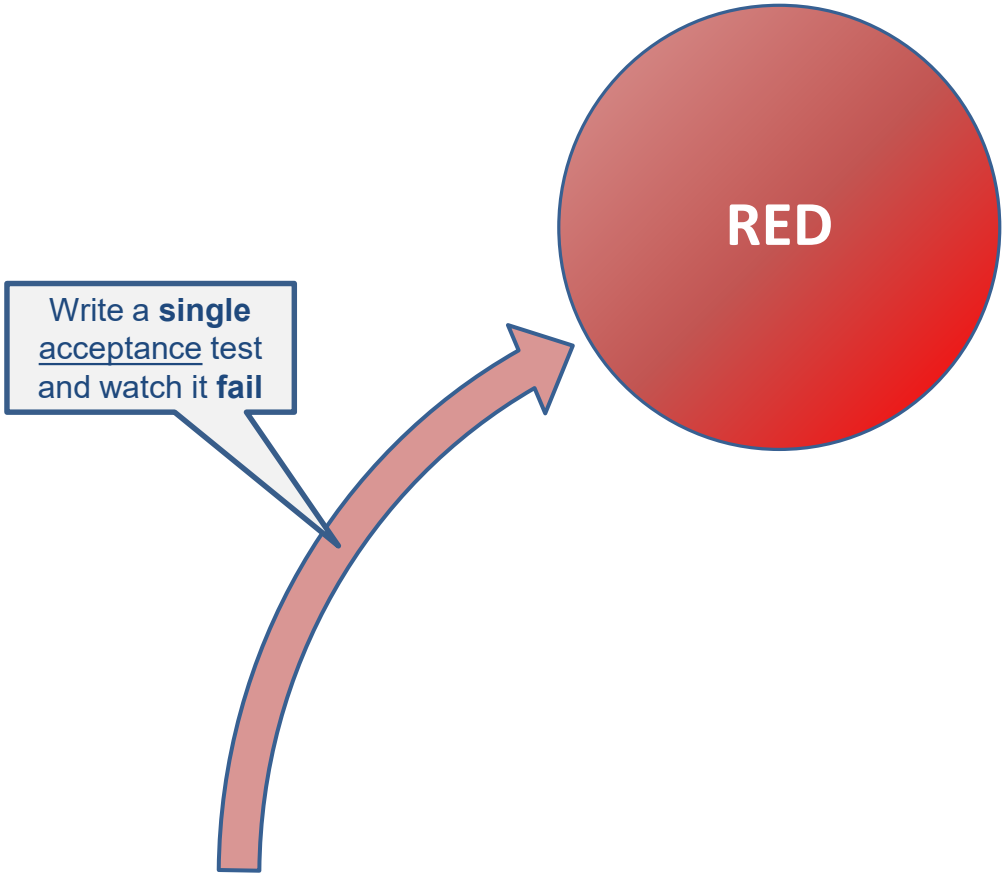

Consider the requirement
Write a **single** test and watch it **fail**
RED
Write **just enough** code to make the test pass
GREEN
Improve the quality of the code, and **also** of the test
REFACTOR

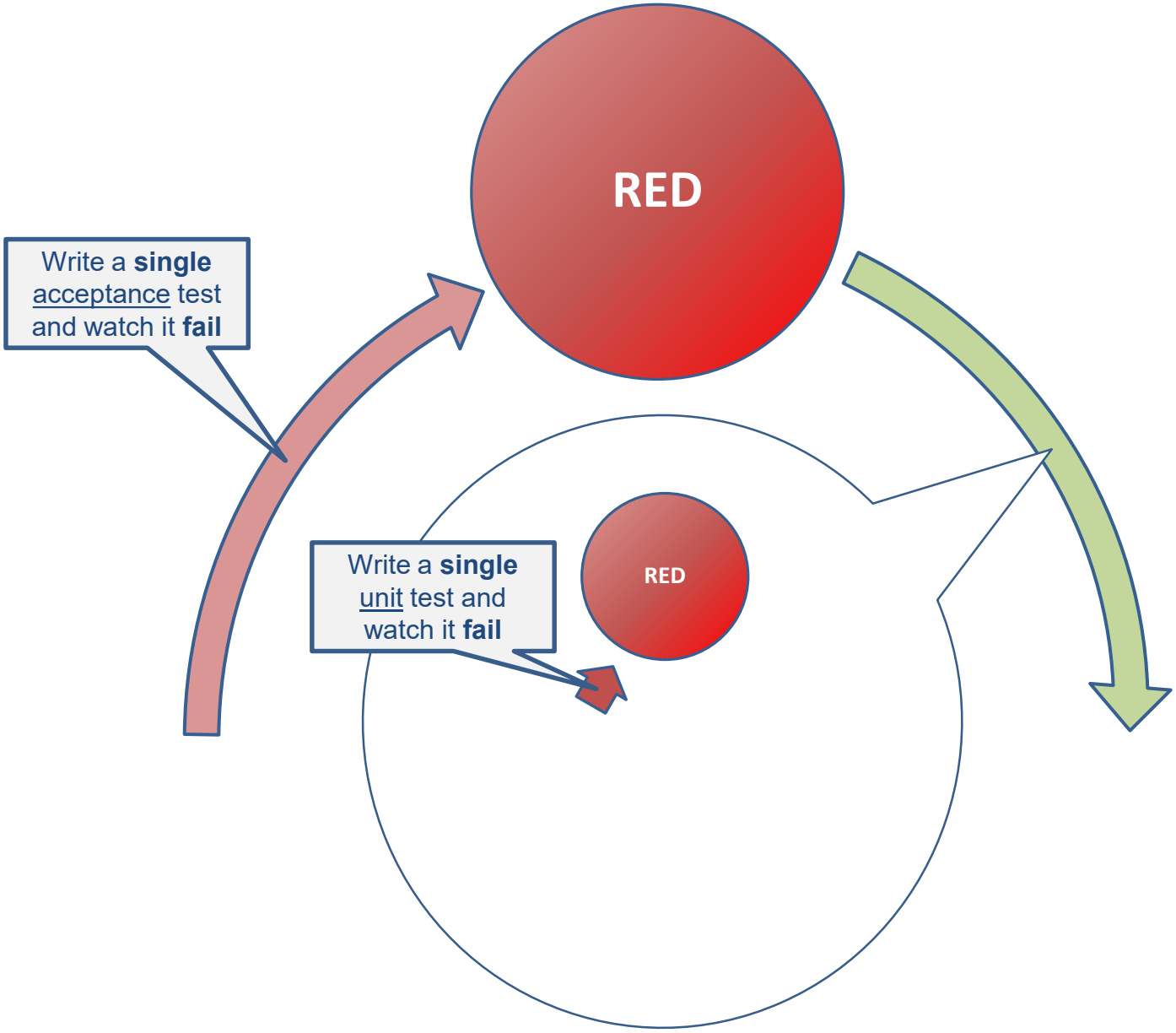

Consider the requirement
Write a **single** test and watch it **fail**
RED
Write **just enough** code to make the test pass
GREEN
REFACTOR
The main design effort occurs here
Improve the quality of the code, and **also** of the test

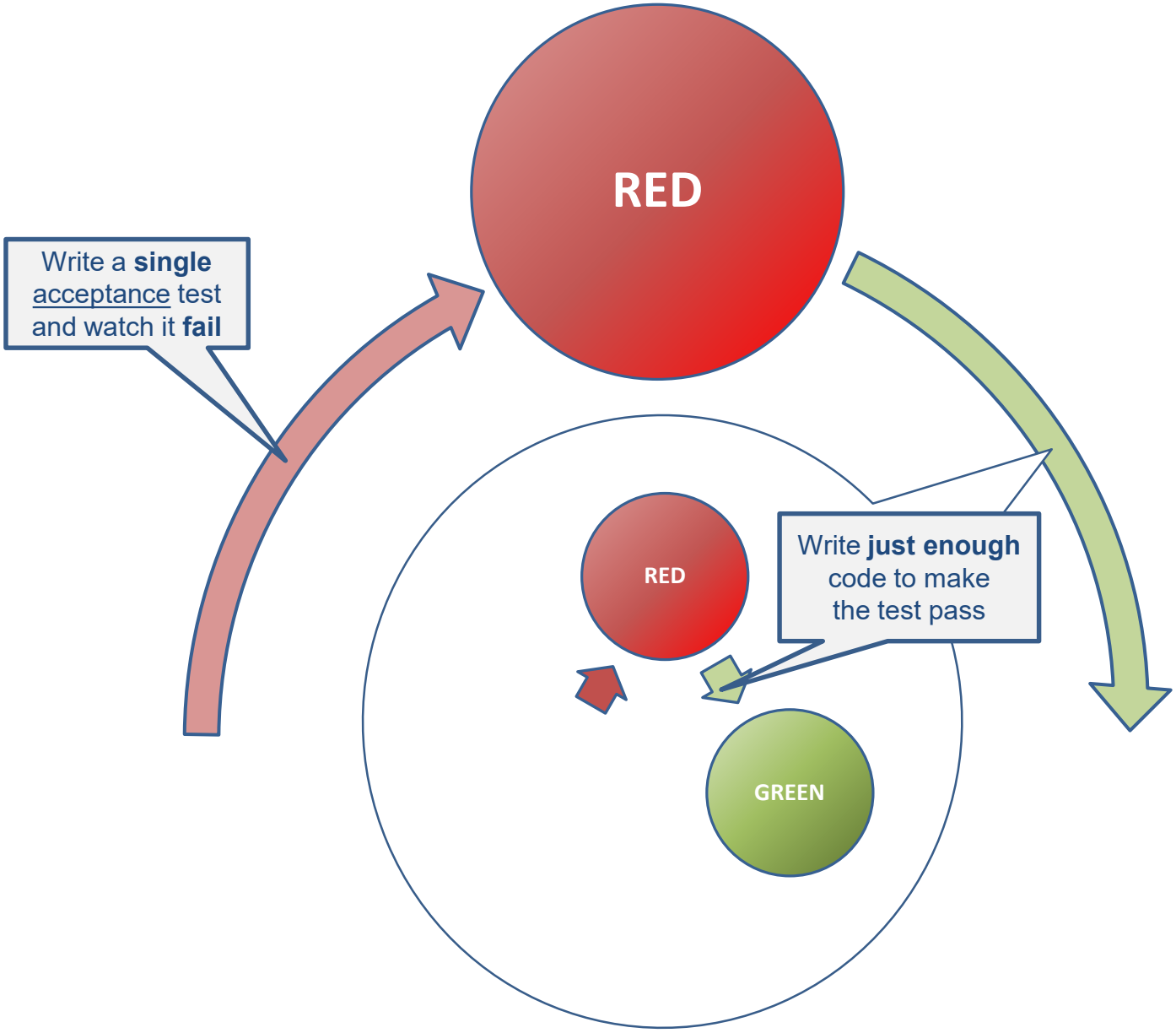

RED
GREEN
REFACTOR

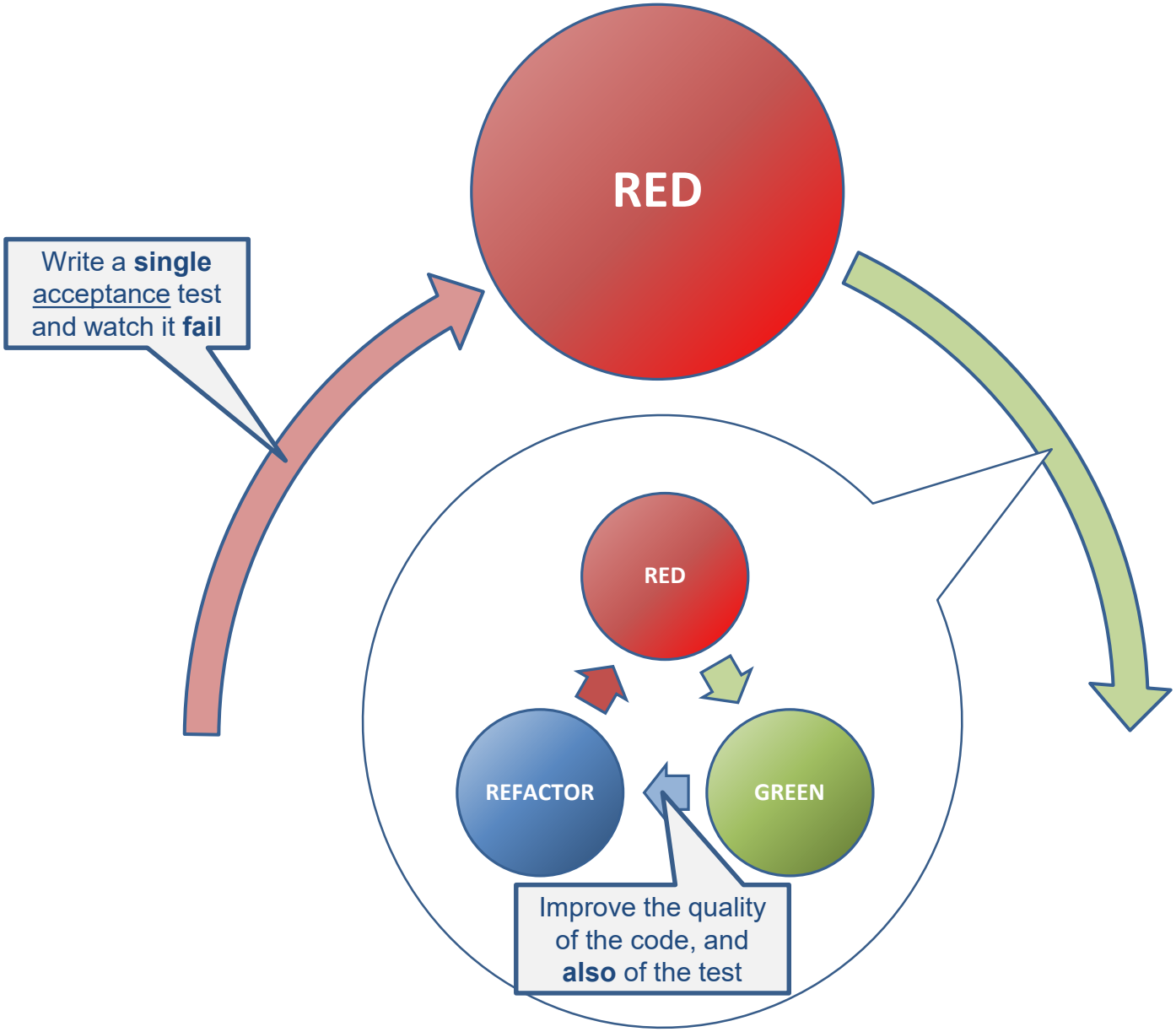
# « Outside In » TDD

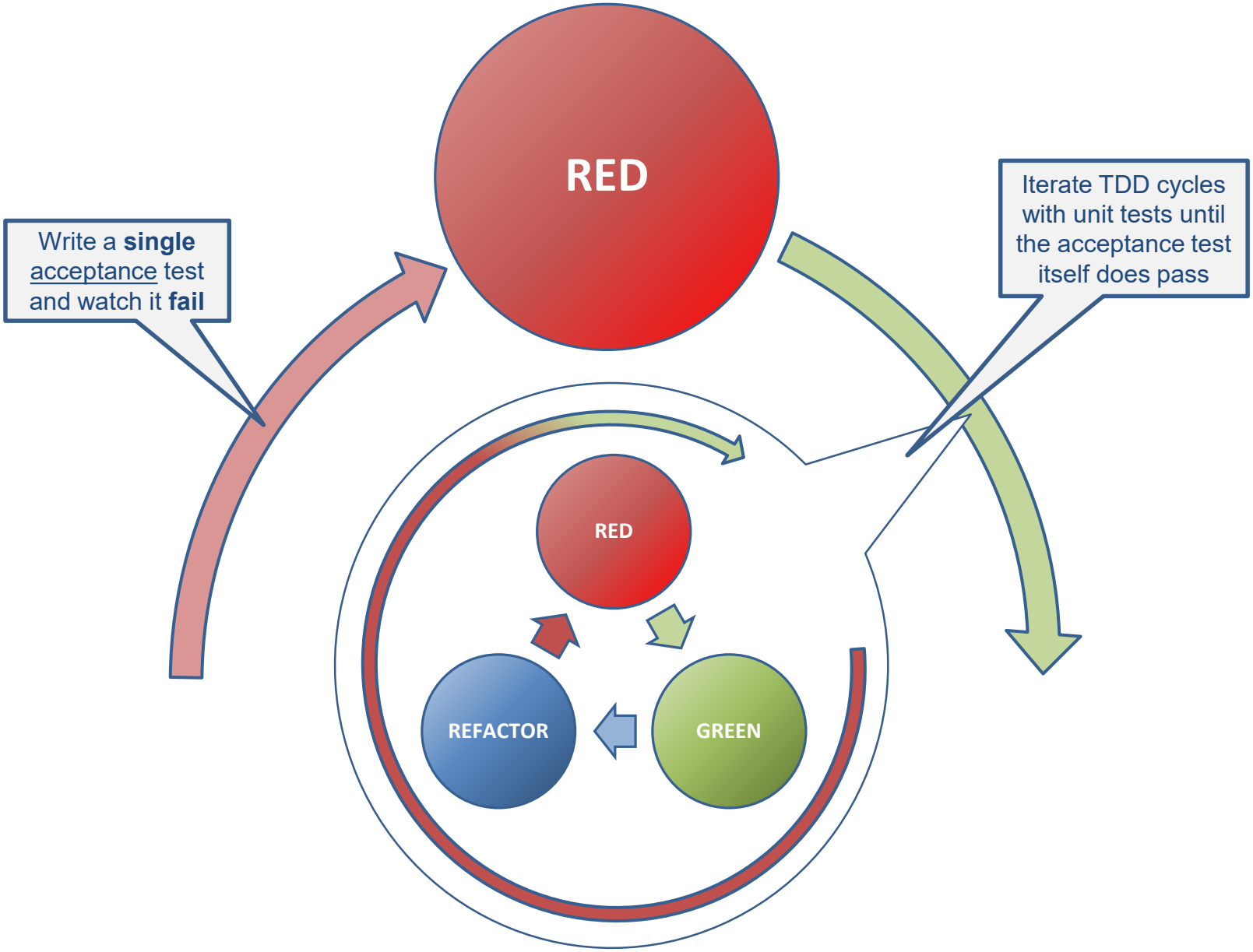

RED
GREEN
REFACTOR
RED
GREEN
REFACTOR

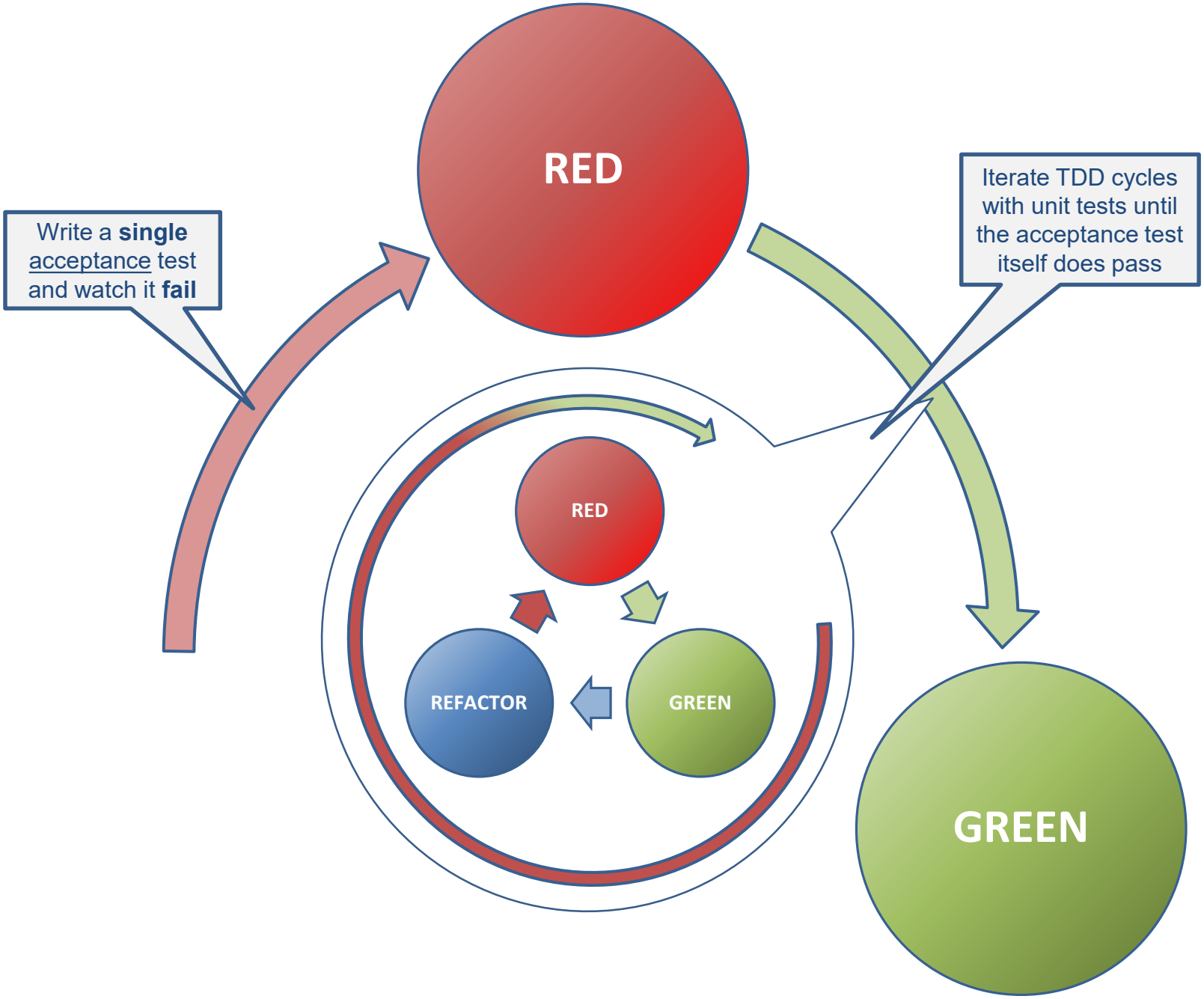

RED
Write a **single** acceptance test and watch it **fail**

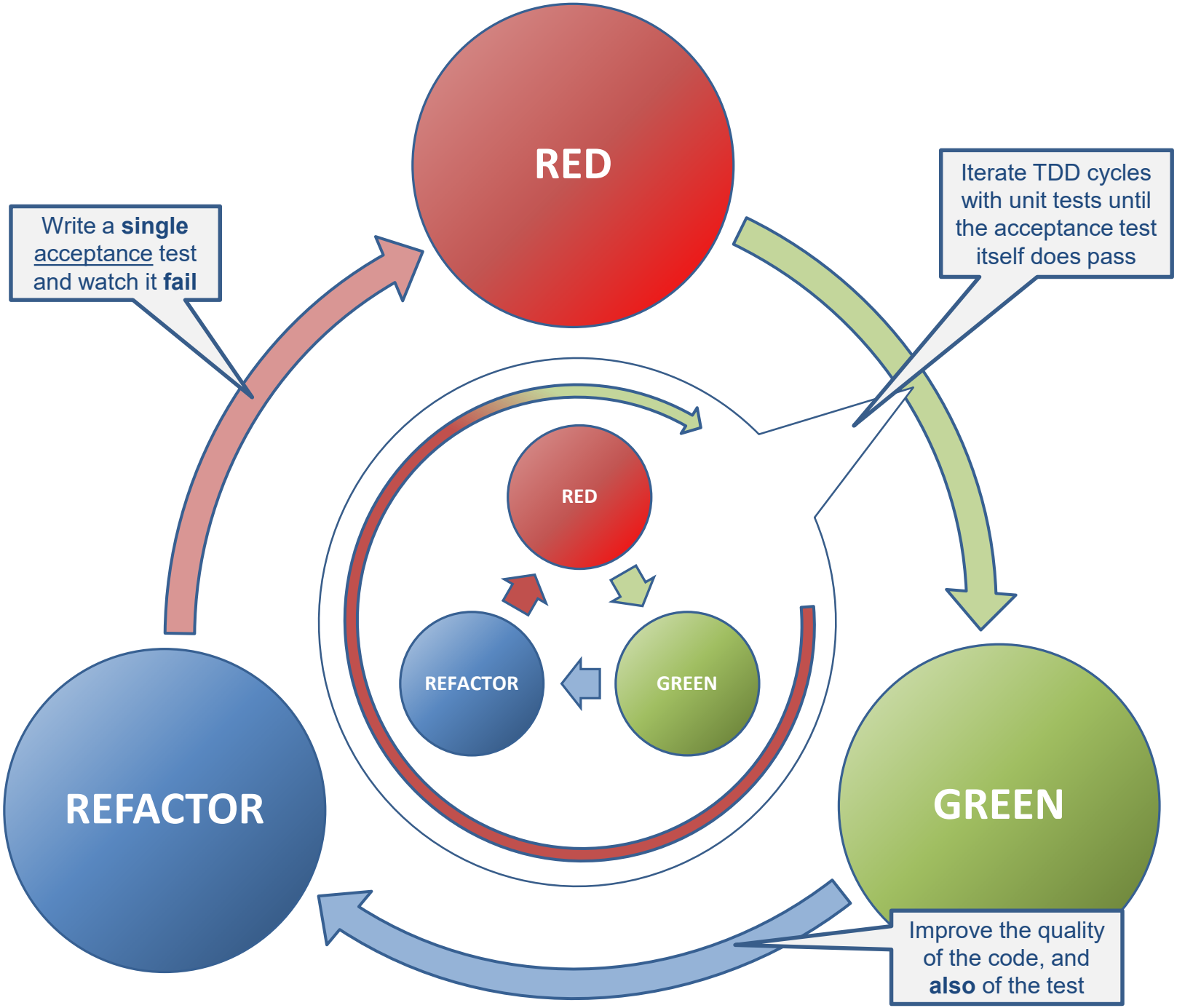

RED
Write a **single** acceptance test and watch it **fail**
Write a **single** unit test and watch it **fail**
RED

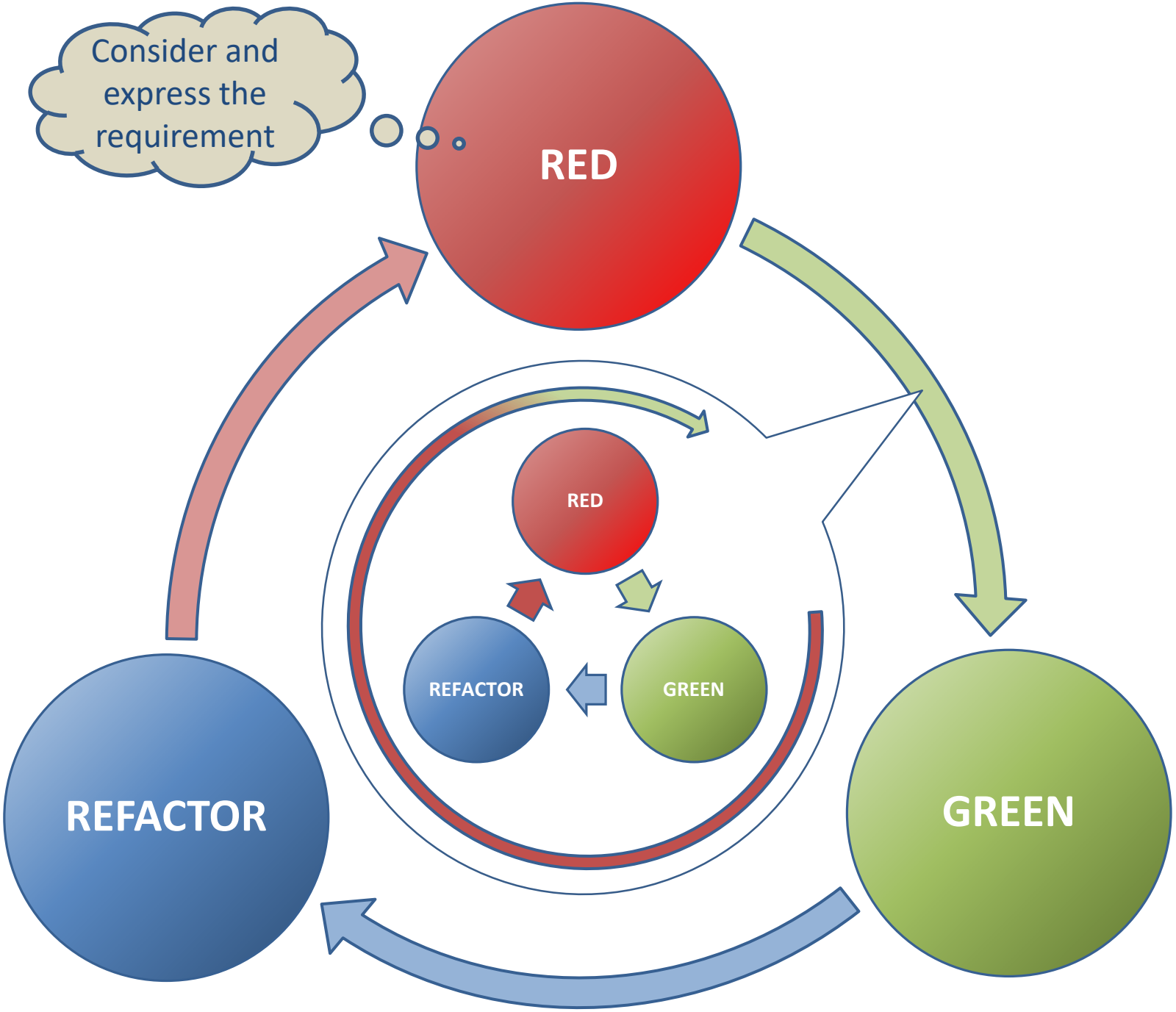

RED
Write a **single** acceptance test and watch it **fail**
RED
Write **just enough** code to make the test pass
GREEN

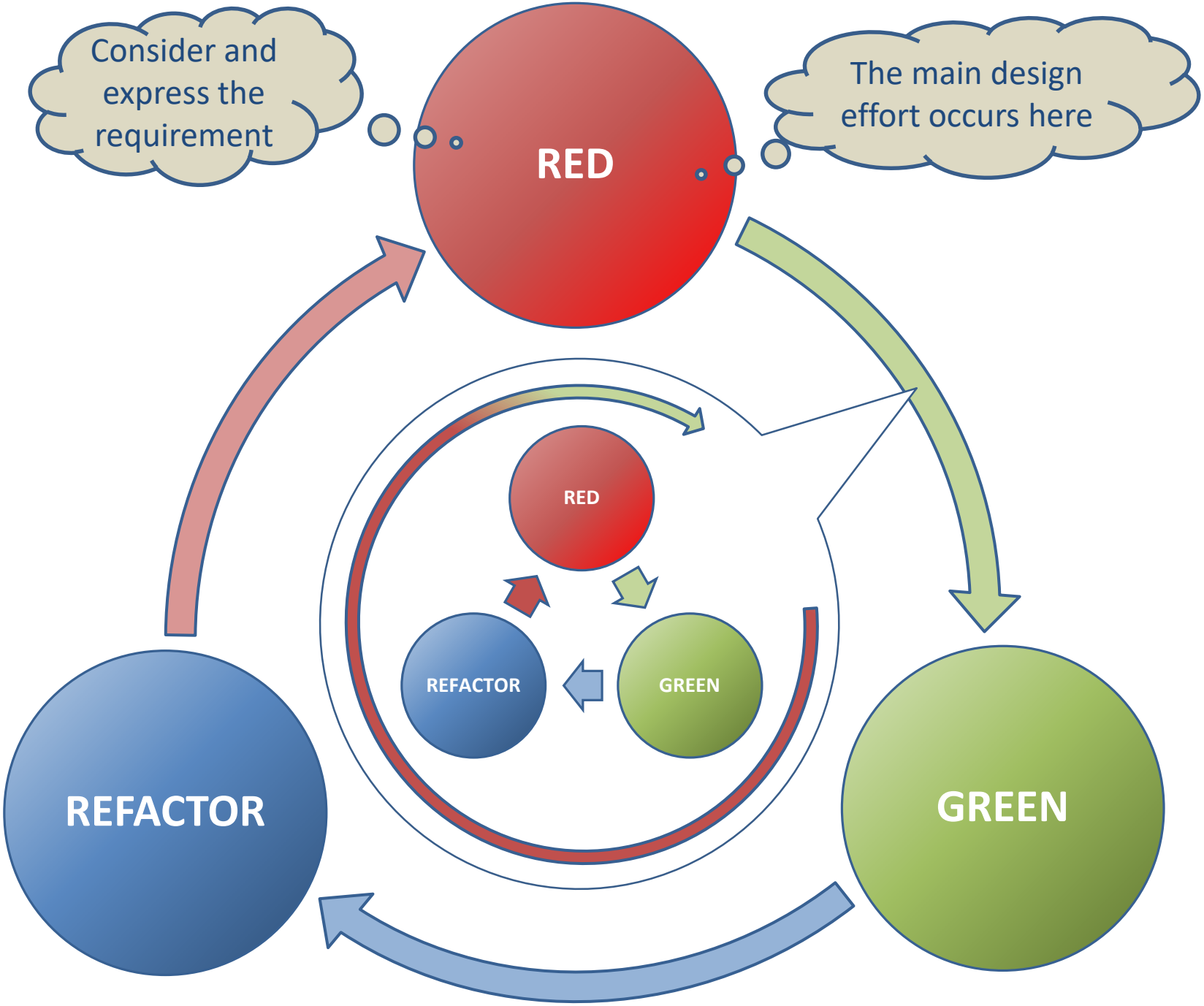

RED
Write a **single** acceptance test and watch it **fail**
RED
REFACTOR
GREEN
Improve the quality of the code, and **also** of the test

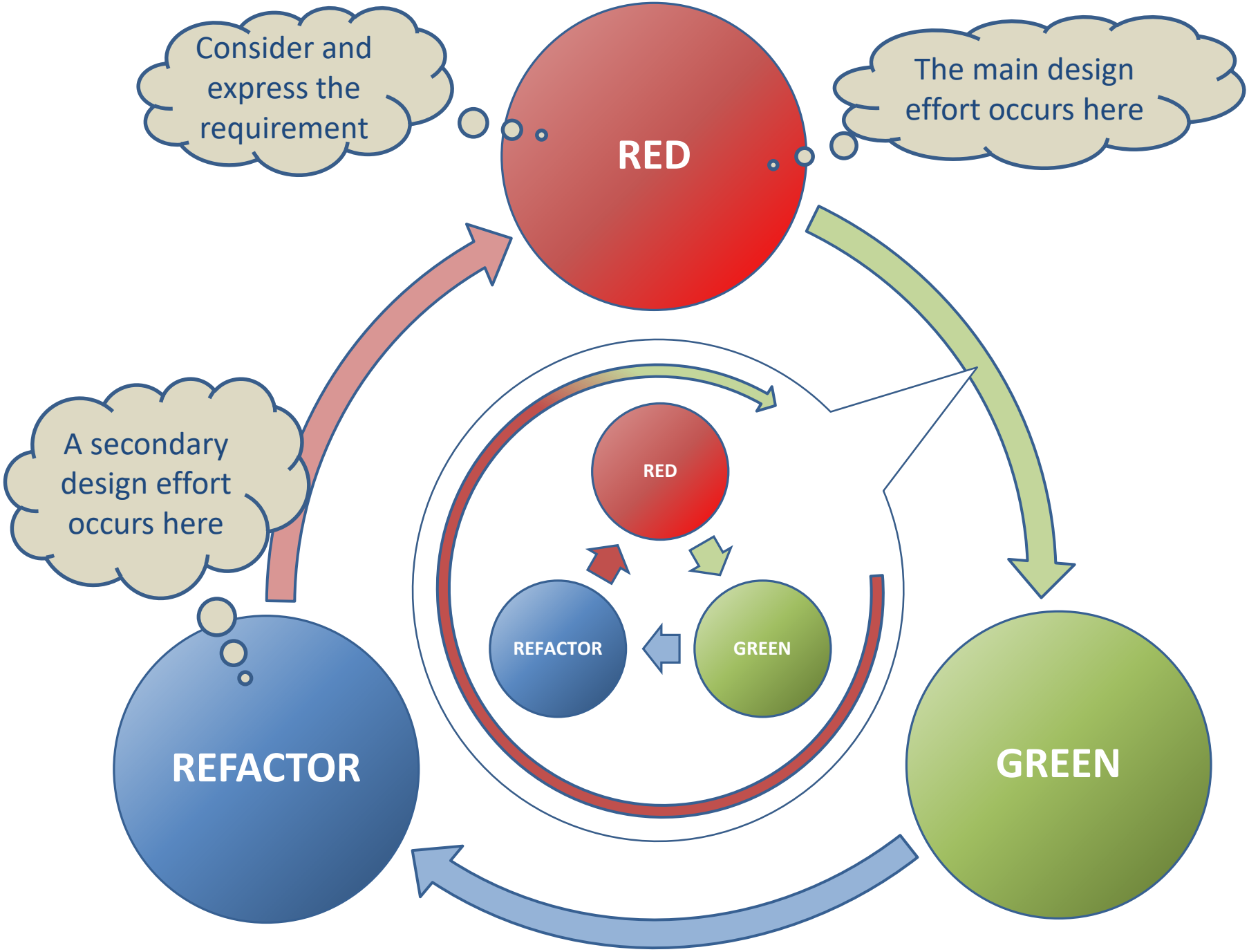

RED
Write a **single** acceptance test and watch it **fail**
Iterate TDD cycles with unit tests until the acceptance test itself does pass
RED
REFACTOR
GREEN

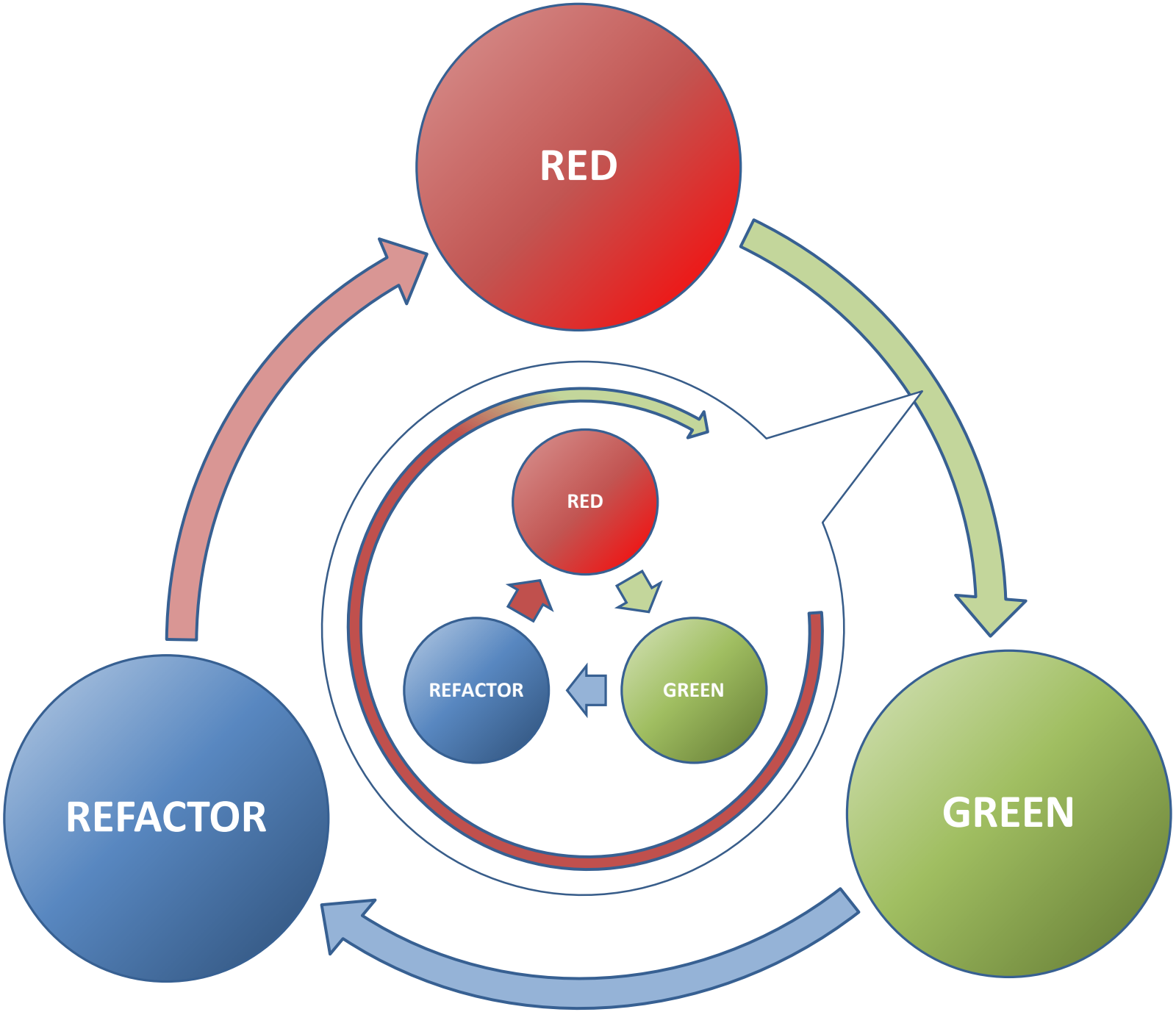

RED
Write a **single** acceptance test and watch it **fail**
Iterate TDD cycles with unit tests until the acceptance test itself does pass
RED
REFACTOR
GREEN
GREEN


RED
Write a **single** acceptance test and watch it **fail**
Iterate TDD cycles with unit tests until the acceptance test itself does pass
RED
REFACTOR
GREEN
REFACTOR
GREEN
Improve the quality of the code, and **also** of the test


Consider and express the requirement
RED
GREEN
REFACTOR
RED
GREEN
REFACTOR

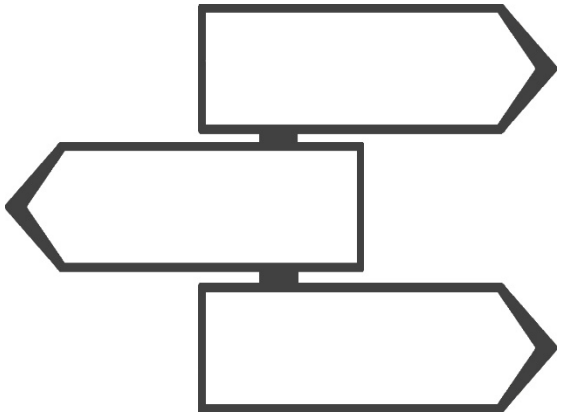

Consider and express the requirement
RED
The main design effort occurs here
RED
REFACTOR
GREEN
REFACTOR
GREEN


Consider and express the requirement
RED
The main design effort occurs here
A secondary design effort occurs here
RED
REFACTOR
GREEN
REFACTOR
GREEN

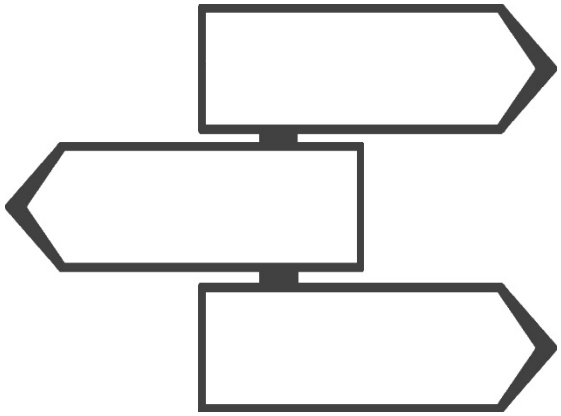

RED
GREEN
REFACTOR
RED
GREEN
REFACTOR

Okay, but ...

... for what purpose?



# *The Use Case as central focus*



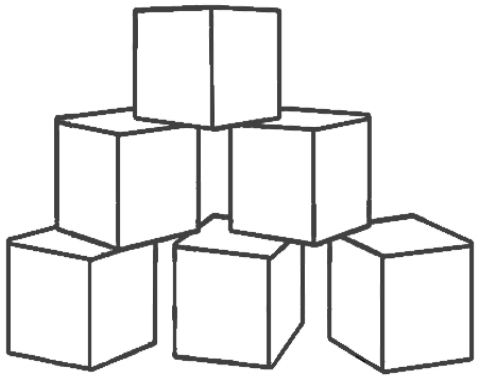
# *The Use Case as central focus*

**« Framing » use cases closer to business needs**

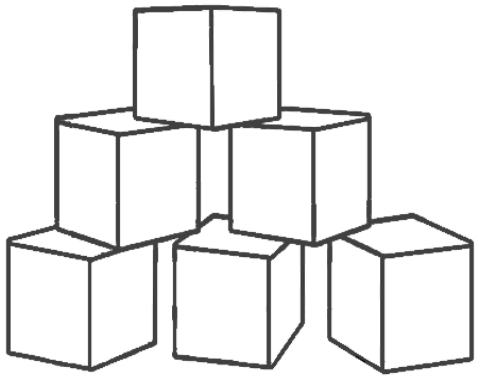
**An indicator of progress in the product, over time**

**Design effort occurs earlier in the development cycle**

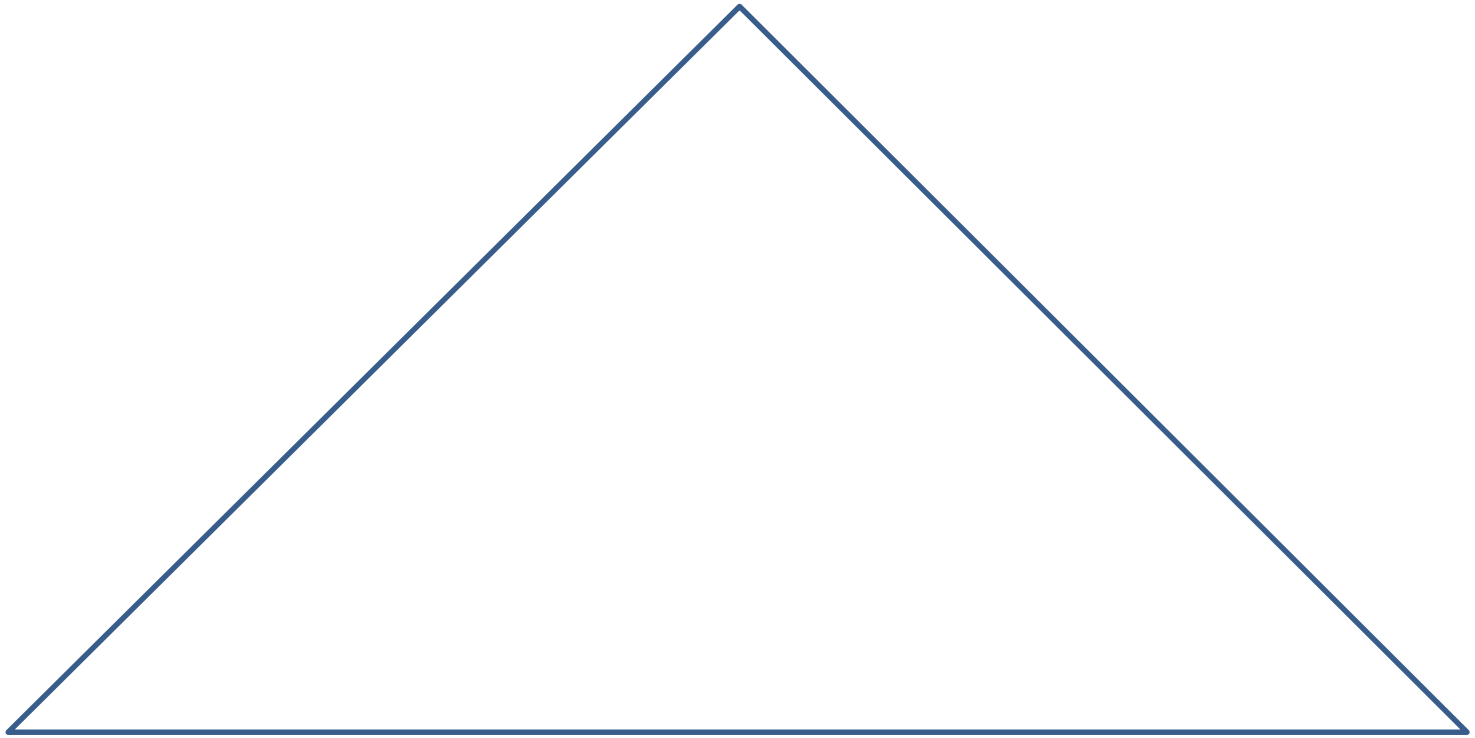
**The preponderance of truly unit tests is facilitated**

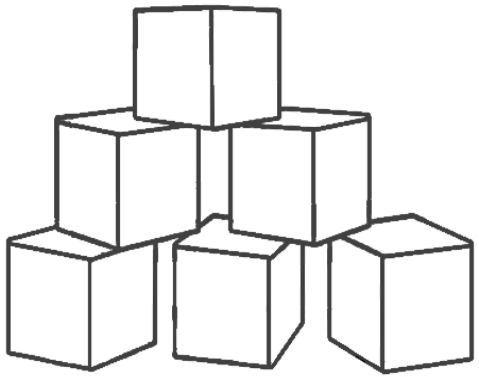


# *The Test Pyramid*

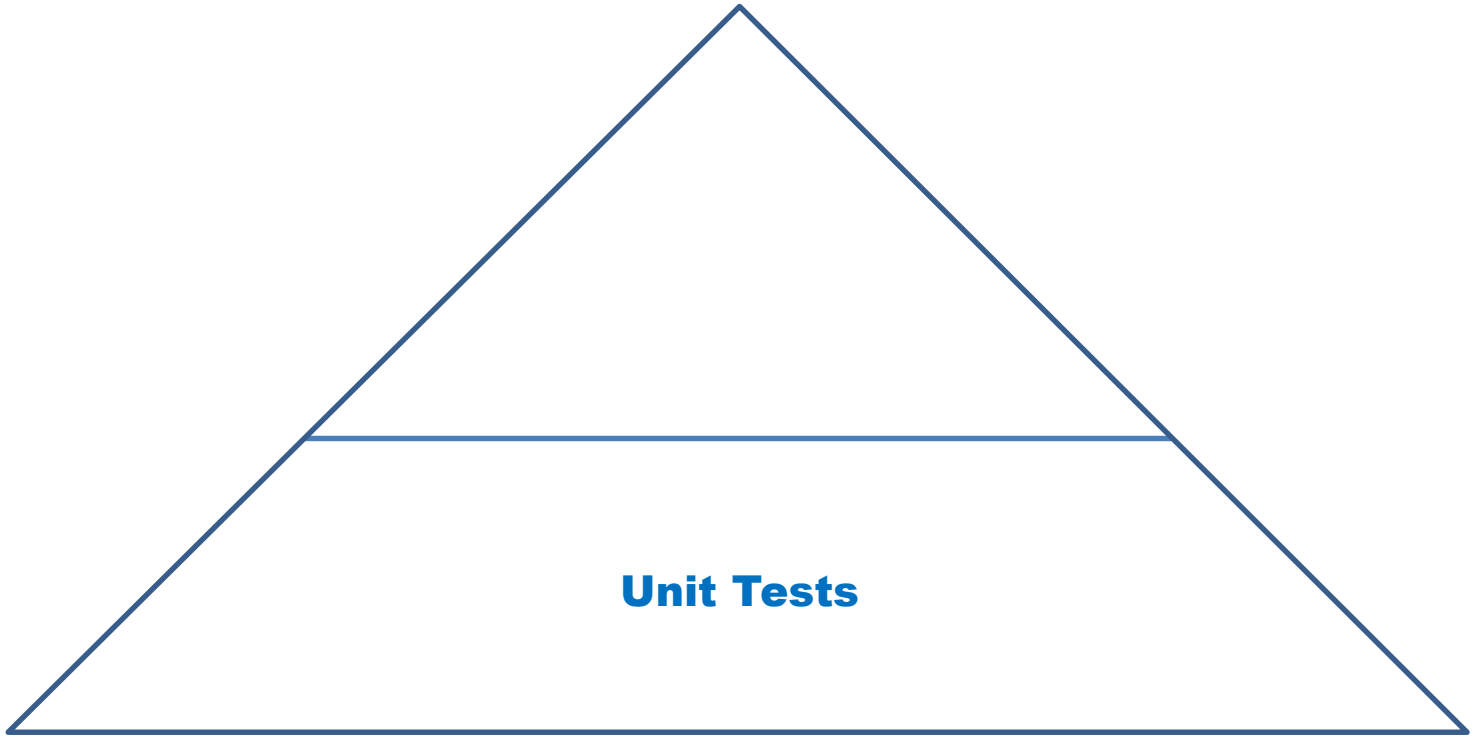


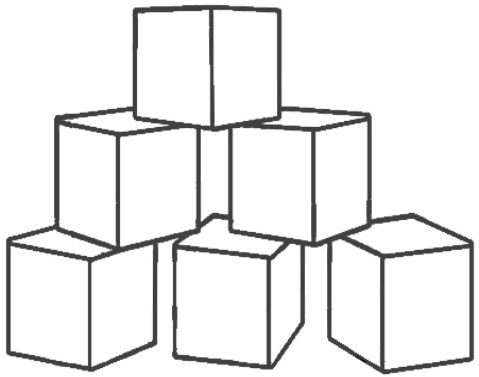
# *The Test Pyramid*



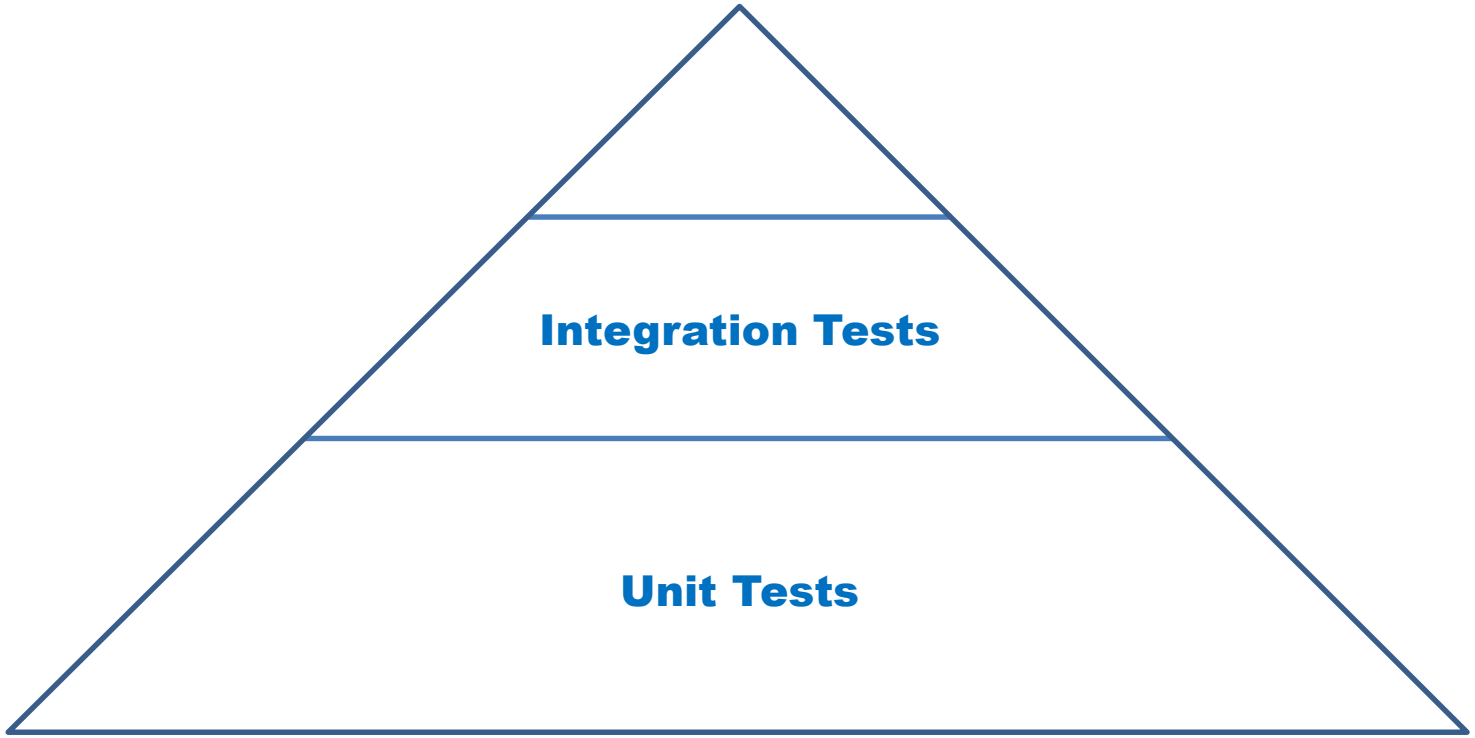


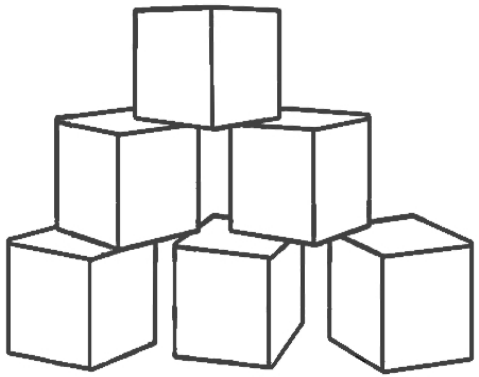
# The Test Pyramid



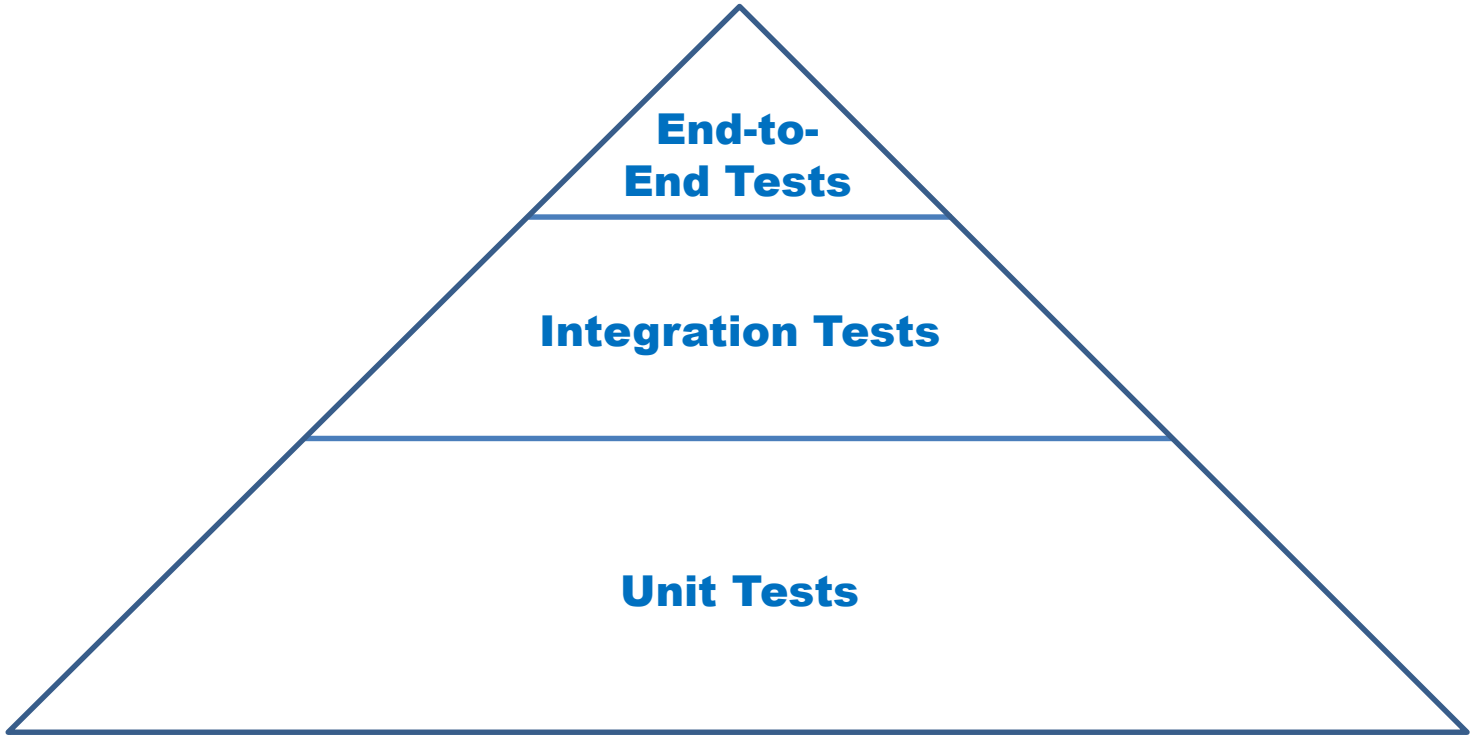


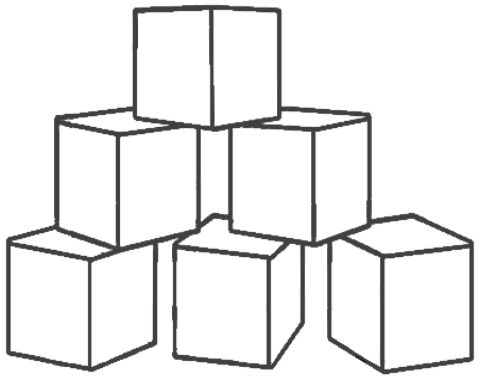
# The Test Pyramid



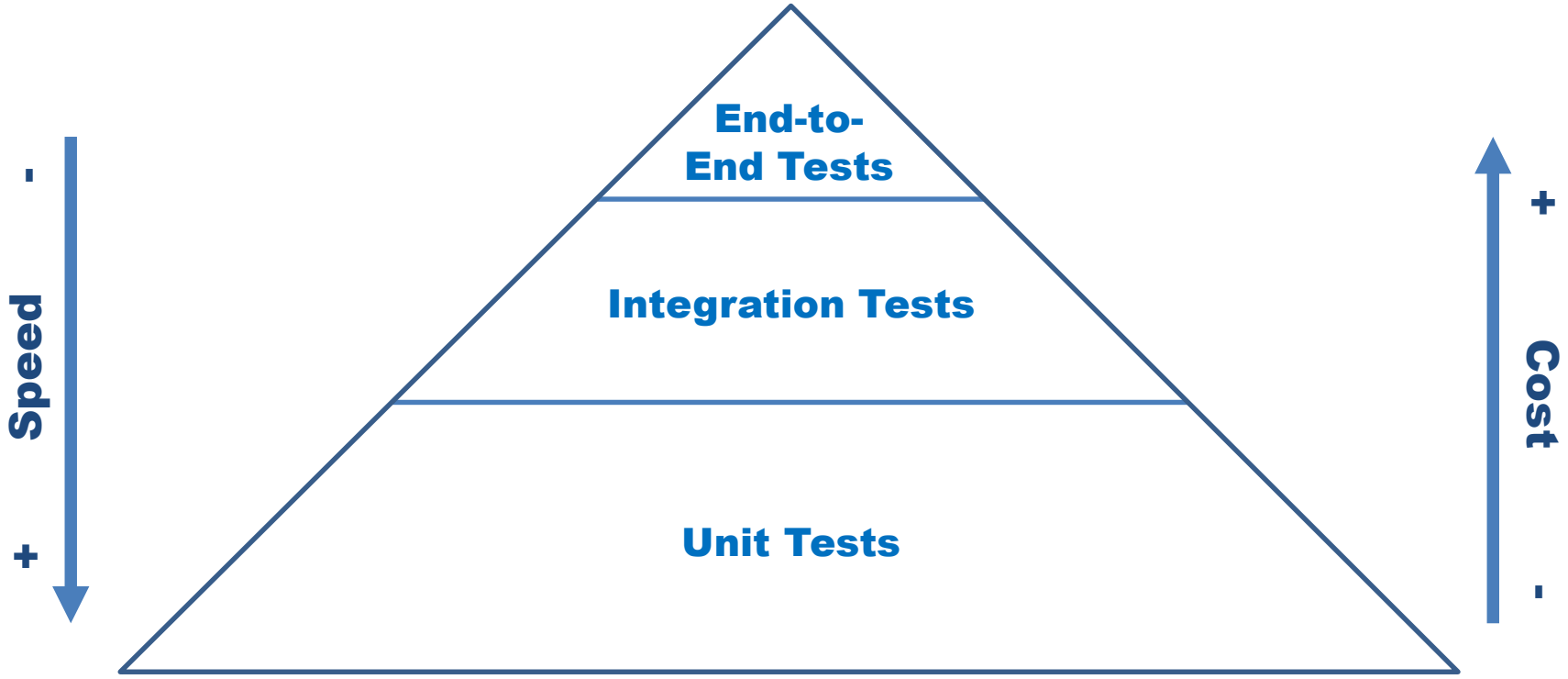


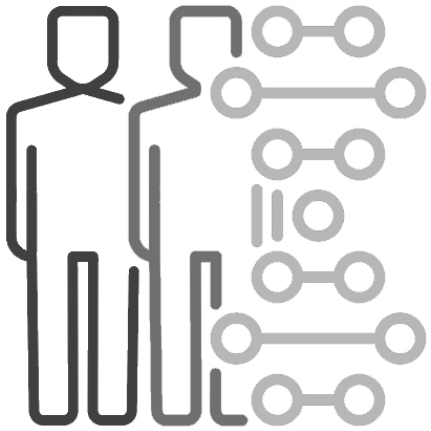
# The Test Pyramid



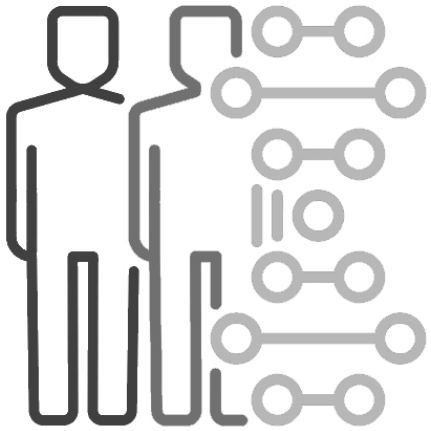


# The Test Pyramid





# *Powerful allies : Test Doubles*



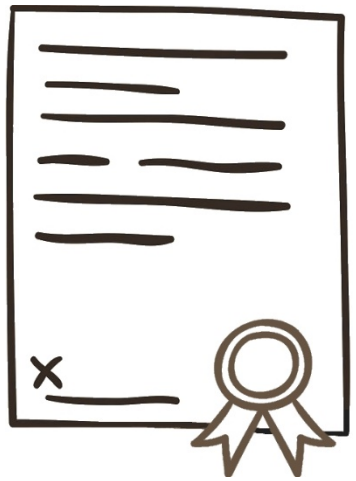
# *Powerful allies : Test Doubles*

**Dummy, Fake, Stub, Spy, Mock ... often simplified as: Mock**

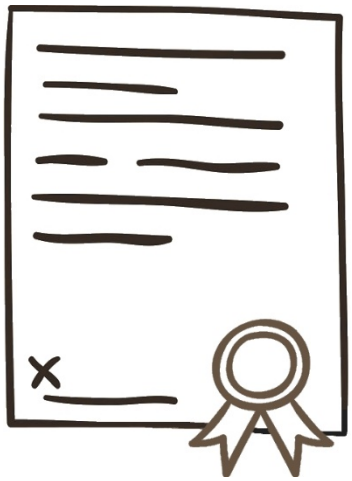
**Collaborators are only revealing their interfaces**

**Their behaviours and interactions are checked, not their states**

**One same test truly has only one single reason to fail**



***Public interface standing as a contract***



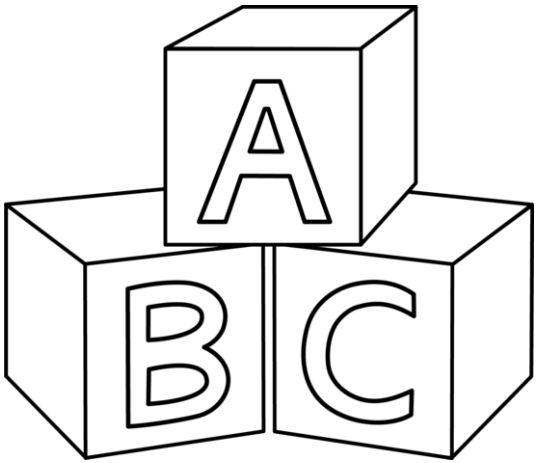
# *Public interface standing as a contract*

**Business object interfaces define their behavior**

**The separation of responsibilities and dependencies is facilitated**

**Tests can be truly effective unit ones, as they are in isolation**

# Some reminders




# *Four Rules of Simple Design*

- ✓ **Passes the tests**
- ✓ **Reveals intention**
- ✓ **No duplication**
- ✓ **Fewest elements**

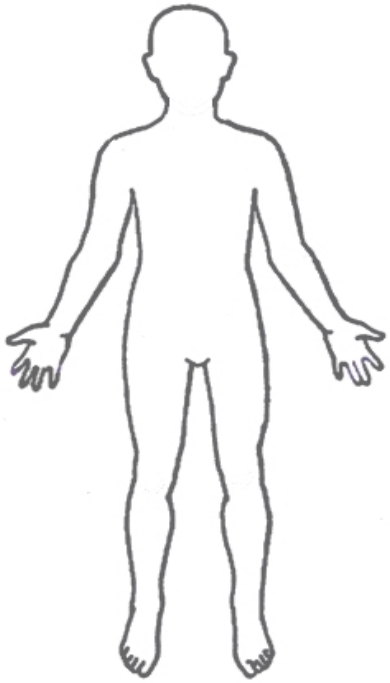
# *Some guidelines for the development cycle*

**Add one (and only one) new test while in « Green »**
**(at a same given test level: unit / acceptance)**

**Watch the test fail before coding corresponding solution**

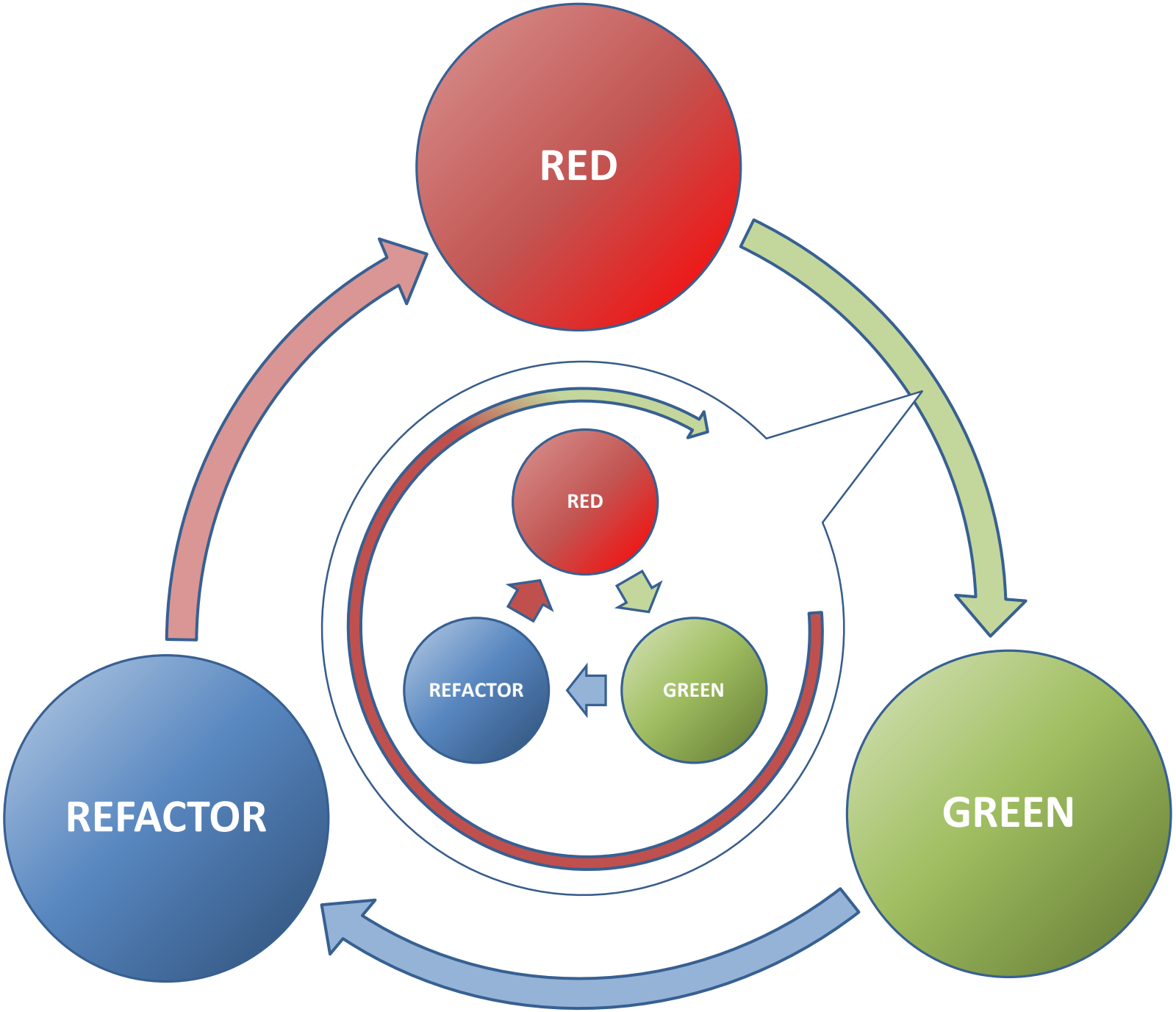
**Code in order to return as soon as possible to « Green »**

**Refactor code or test at any one time, not both**

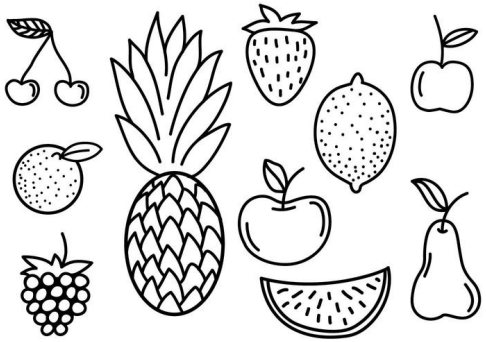


# *Anatomy of a Test*

| | | | |
|---|---|---|---|
| 3 | GIVEN | Context | = states / data |
| 2 | WHEN | Event | = what is being tested |
| 1 | THEN | Expectation | = solution to the requirement |


RED
GREEN
REFACTOR
RED
GREEN
REFACTOR

# Now let’s do some coding!



# *Fruit Shop*

**Charge the right amount when the customer goes to the checkout.**

**Business rules (prices in cents):**

- **1 apple costs 100**
- **1 banana costs 150**
- **1 cherry costs 75**

# FruitShopTest

# FruitShopTest

```
@Test
public void noCheckoutForEmptyCart() {

}
```

# FruitShopTest

```
@Test
public void noCheckoutForEmptyCart() {

    // THEN
    assertThat(totalAmount).isEqualTo(0);
}
```

# FruitShopTest

```
@Test
public void noCheckoutForEmptyCart() {

    // WHEN
    int totalAmount = checkout.computeTotalAmount();

    // THEN
    assertThat(totalAmount).isEqualTo(0);
}
```

# FruitShopTest

```
@Test
public void noCheckoutForEmptyCart() {

    // GIVEN
    // Empty cart

    // WHEN
    int totalAmount = checkout.computeTotalAmount();

    // THEN
    assertThat(totalAmount).isEqualTo(0);
}
```

# FruitShopTest

```
private Checkout checkout = new Checkout();

@Test
public void noCheckoutForEmptyCart() {

    // GIVEN
    // Empty cart

    // WHEN
    int totalAmount = checkout.computeTotalAmount();

    // THEN
    assertThat(totalAmount).isEqualTo(0);
}
```

# FruitShopTest

```
private Cart cart = new Cart();
private Checkout checkout = new Checkout(cart);

@Test
public void noCheckoutForEmptyCart() {

    // GIVEN
    // Empty cart

    // WHEN
    int totalAmount = checkout.computeTotalAmount();

    // THEN
    assertThat(totalAmount).isEqualTo(0);
}
```

# CheckoutTest

# CheckoutTest

```
@Test
public void noCheckoutForEmptyCart() {
}
```

# CheckoutTest

```
@Test
public void noCheckoutForEmptyCart() {

    // THEN
    assertThat(totalAmount).isEqualTo(0);
}
```

# CheckoutTest

```
@Test
public void noCheckoutForEmptyCart() {

    // WHEN
    int totalAmount = checkout.computeTotalAmount();

    // THEN
    assertThat(totalAmount).isEqualTo(0);
}
```

# CheckoutTest

```
private Checkout checkout = new Checkout();

@Test
public void noCheckoutForEmptyCart() {

    // WHEN
    int totalAmount = checkout.computeTotalAmount();

    // THEN
    assertThat(totalAmount).isEqualTo(0);
}
```

# CheckoutTest

```
private Checkout checkout = new Checkout();

@Test
public void noCheckoutForEmptyCart() {

    // GIVEN
    given(cart.listFruits())
        .willReturn(Collections.emptyList());

    // WHEN
    int totalAmount = checkout.computeTotalAmount();

    // THEN
    assertThat(totalAmount).isEqualTo(0);
}
```

# CheckoutTest

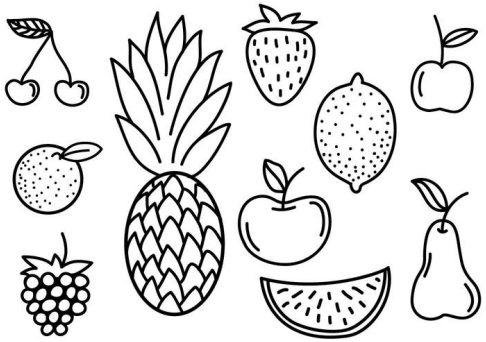
```
private Cart cart = new Mock(Cart.class);
private Checkout checkout = new Checkout(cart);

@Test
public void noCheckoutForEmptyCart() {

    // GIVEN
    given(cart.listFruits())
        .willReturn(Collections.emptyList());

    // WHEN
    int totalAmount = checkout.computeTotalAmount();

    // THEN
    assertThat(totalAmount).isEqualTo(0);
}
```

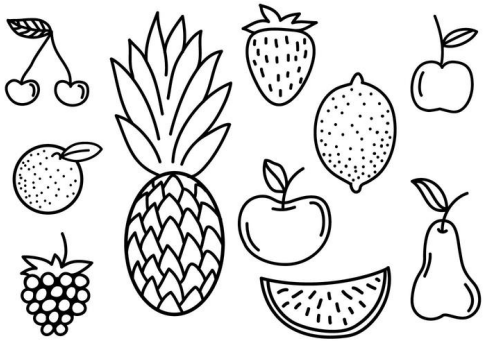


# *Fruit Shop*

**Charge the right amount when the customer goes to the checkout.**

**Business rules (prices in cents):**

- **1 apple costs 100**
- **1 banana costs 150**
- **1 cherry costs 75**

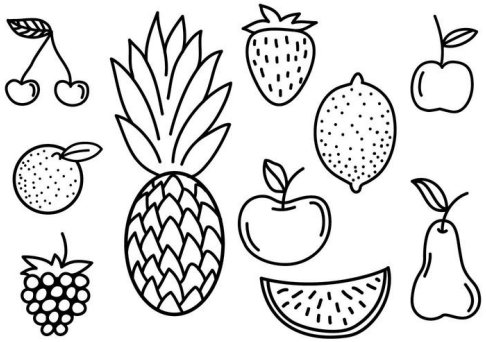


# Fruit Shop

**Charge the right amount when the customer goes to the checkout.**

**Business rules (prices in cents):**

- **1 apple costs 100**
- **1 free apple when two bought apples**
- **1 banana costs 150**
- **1 cherry costs 75**

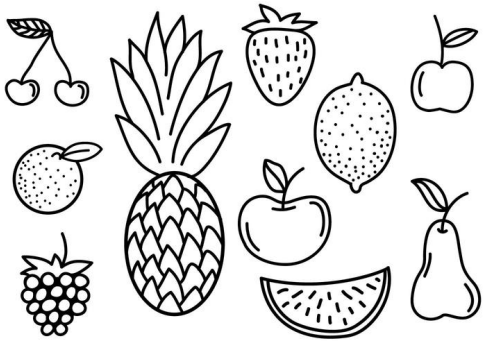


# Fruit Shop

**Charge the right amount when the customer goes to the checkout.**

**Business rules (prices in cents):**

- **1 apple costs 100**
- **1 free apple when two bought apples**
- **1 banana costs 150**
- **The second banana is half price**
- **1 cherry costs 75**



# Fruit Shop

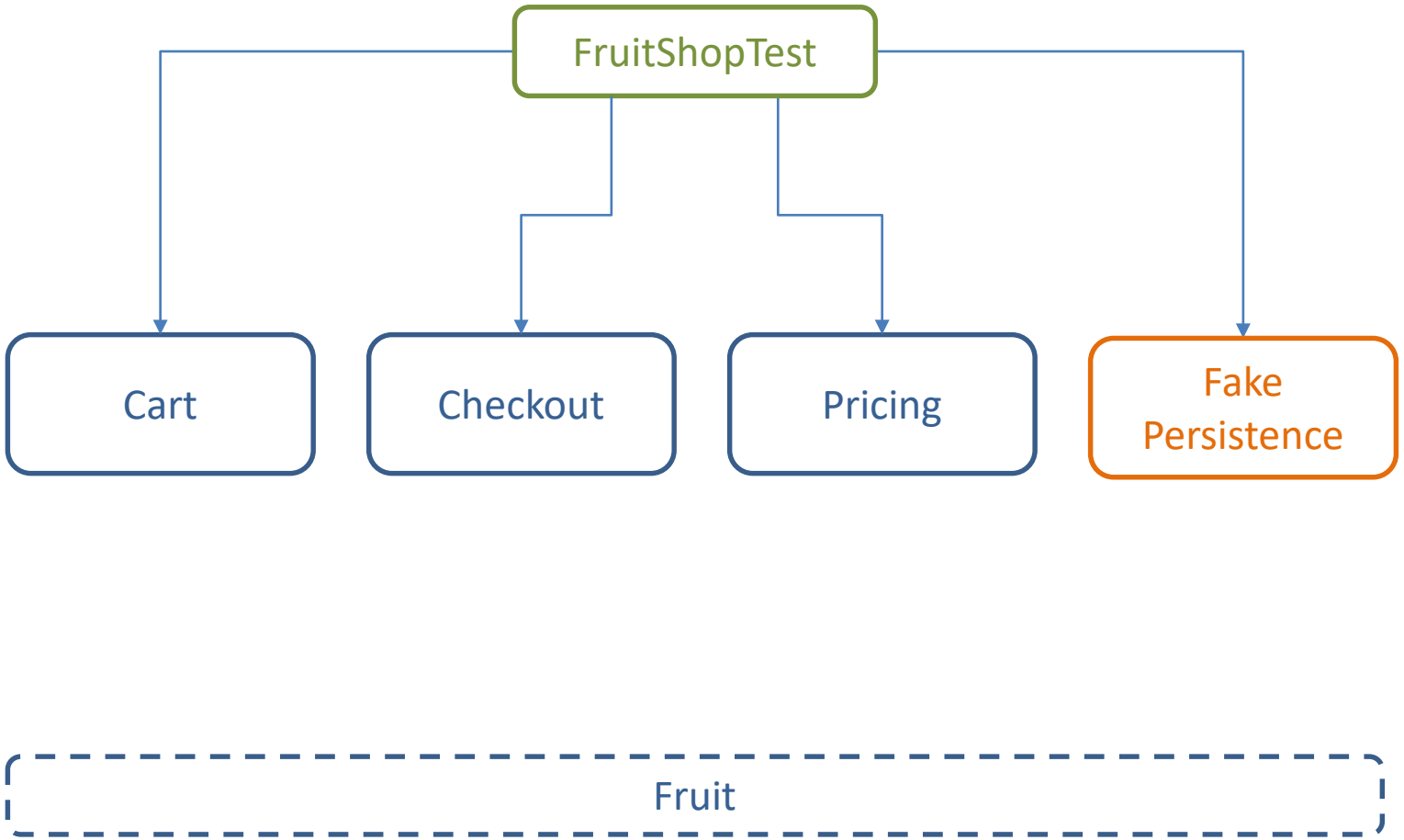
**Charge the right amount when the customer goes to the checkout.**

**Business rules (prices in cents):**

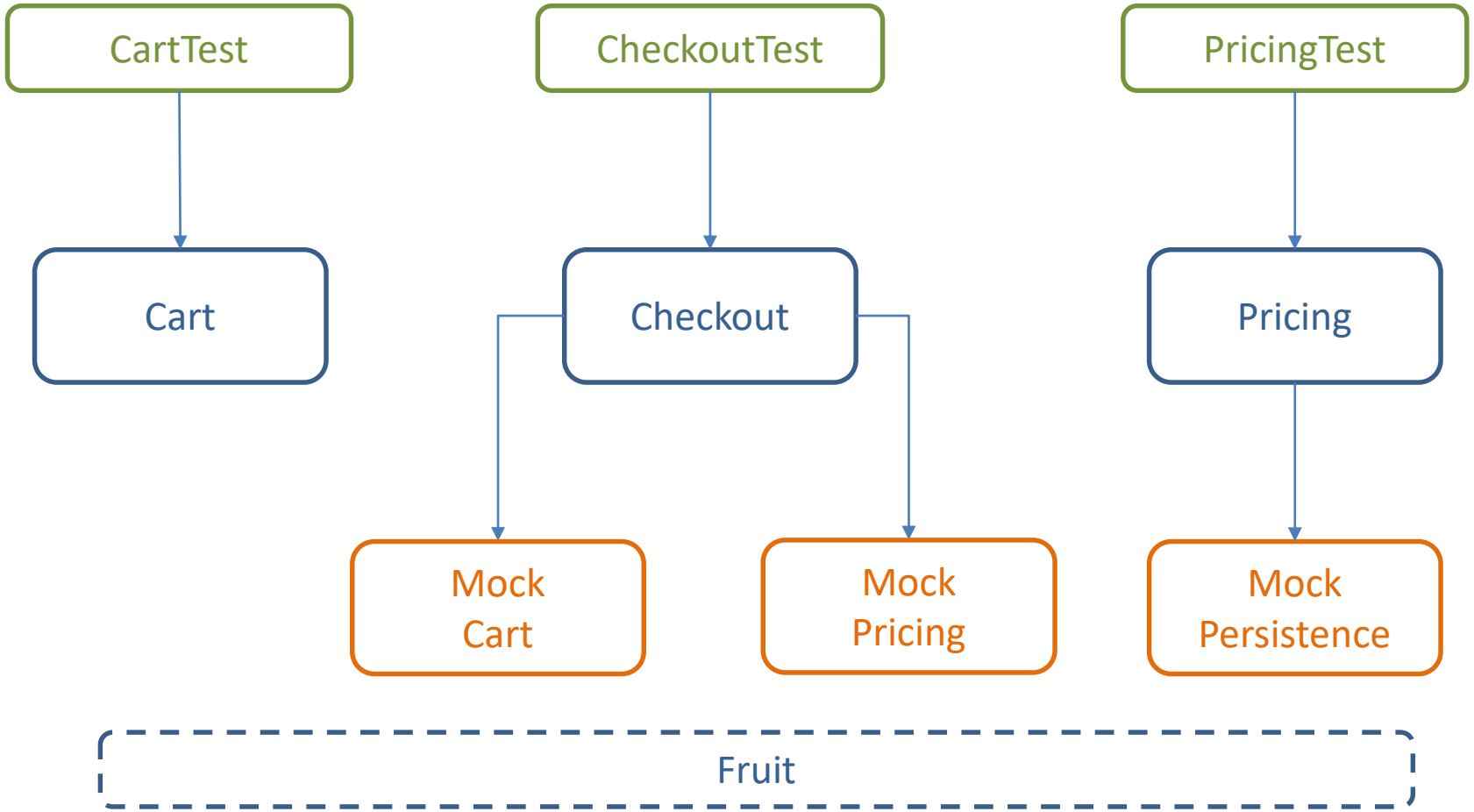
- **1 apple costs 100**
- **1 free apple when two bought apples**
- **1 banana costs 150**
- **The second banana is half price**
- **1 cherry costs 75**
- **A loyalty program customer is entitled to a 10% discount**

# One possible solution

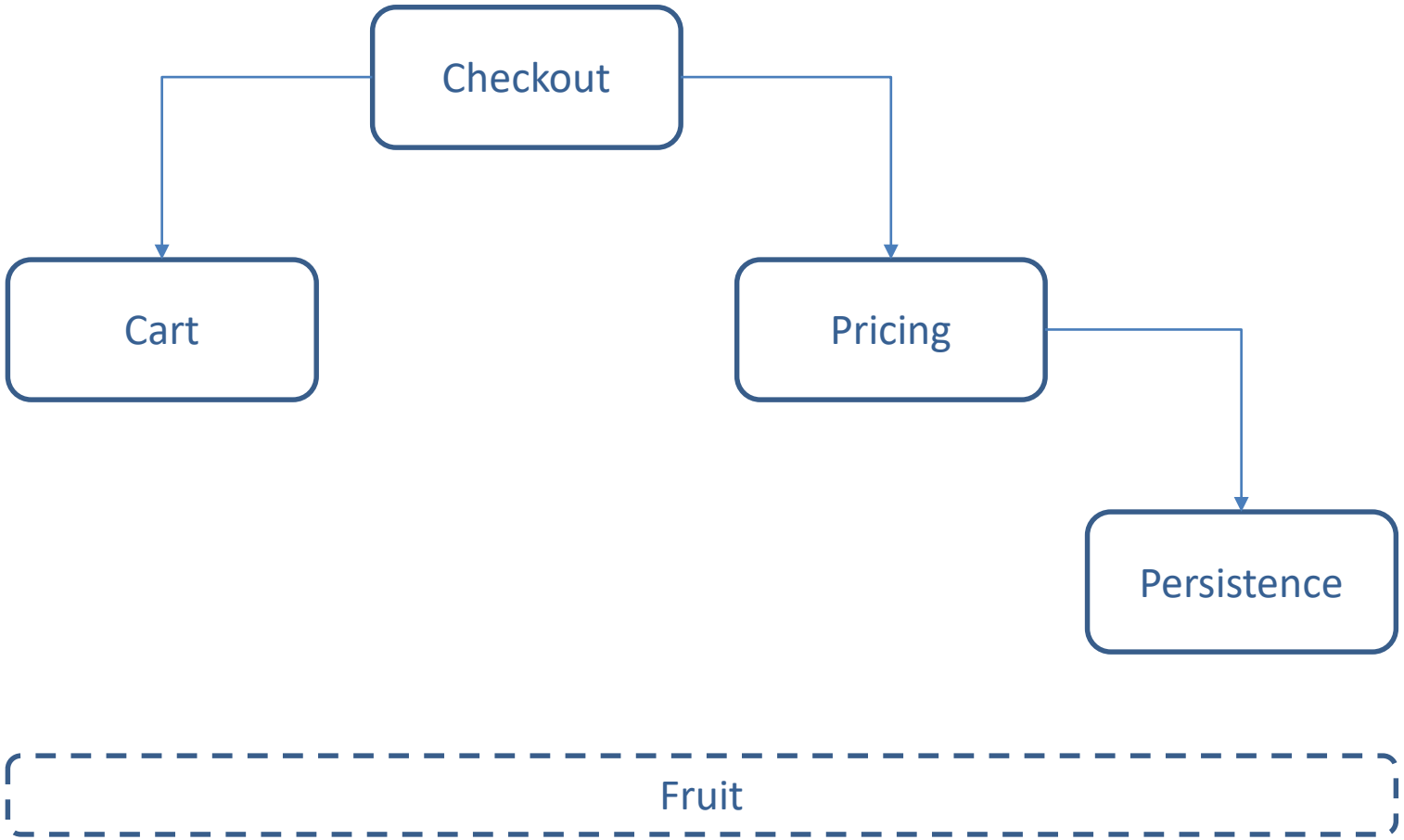
# Acceptance Test



# Unit Tests


CartTest
CheckoutTest
PricingTest
Cart
Checkout
Pricing
Mock Cart
Mock Pricing
Mock Persistence
Fruit

# *Implementation*




THANK
YOU