# Crafting Code

sandro@codurance.com
@sandromancuso
@codurance

codurance
craft at heart
http://codurance.com

# Introductions

Who are you?
Which languages do you use?
What's your experience with TDD?

# Why this course?

# Forget what you know…

… but just for today

There is no silver bullet when it comes to software.

# Agenda

## Day 1

- Introduction to TDD
- TDD Lifecycle and Naming
- Test-Driving algorithms
- Expressing business rules
- Mocking

## Day 2

- Outside-In TDD with Acceptance Tests
- Testing and Refactoring Legacy Code

# Before we start…

Make sure you can do the following:

- Start a new project
- Write and run a failing test

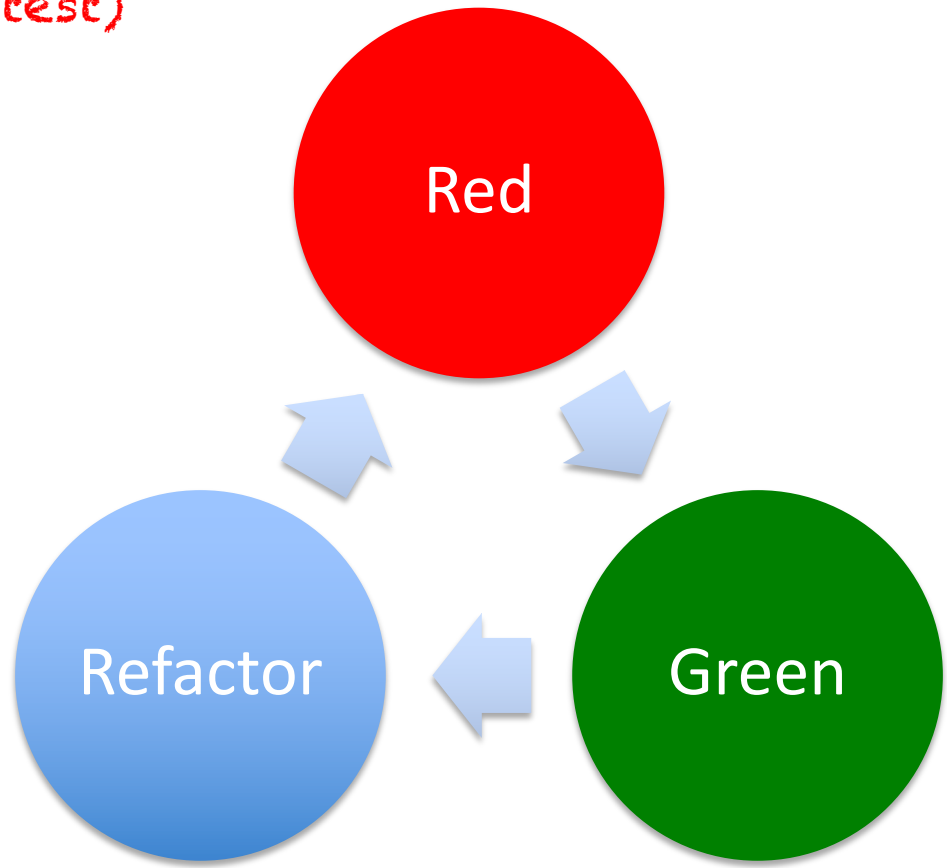
```
assertThat(true, is(false));
```

Also, make sure you have the following installed:

- Mocking framework
- Git (optional but recommended)

http://codingboard.org/boards/craftingcode

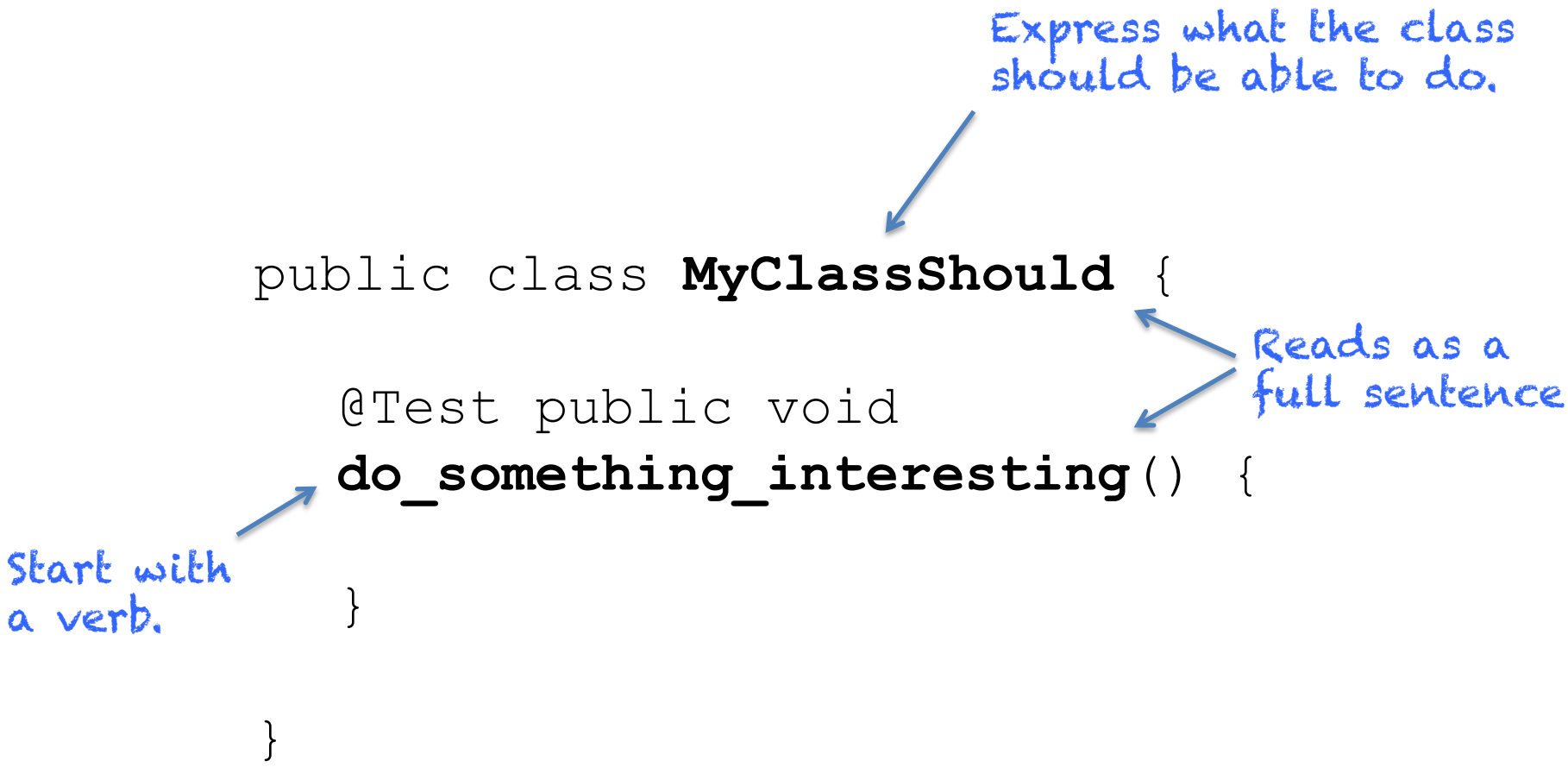
Specify what you want your code to do. (failing test)




Clean your code and test. Remove duplication. (make it better)

Create just enough code to satisfy the desired behaviour. (make it work)

# Naming convention

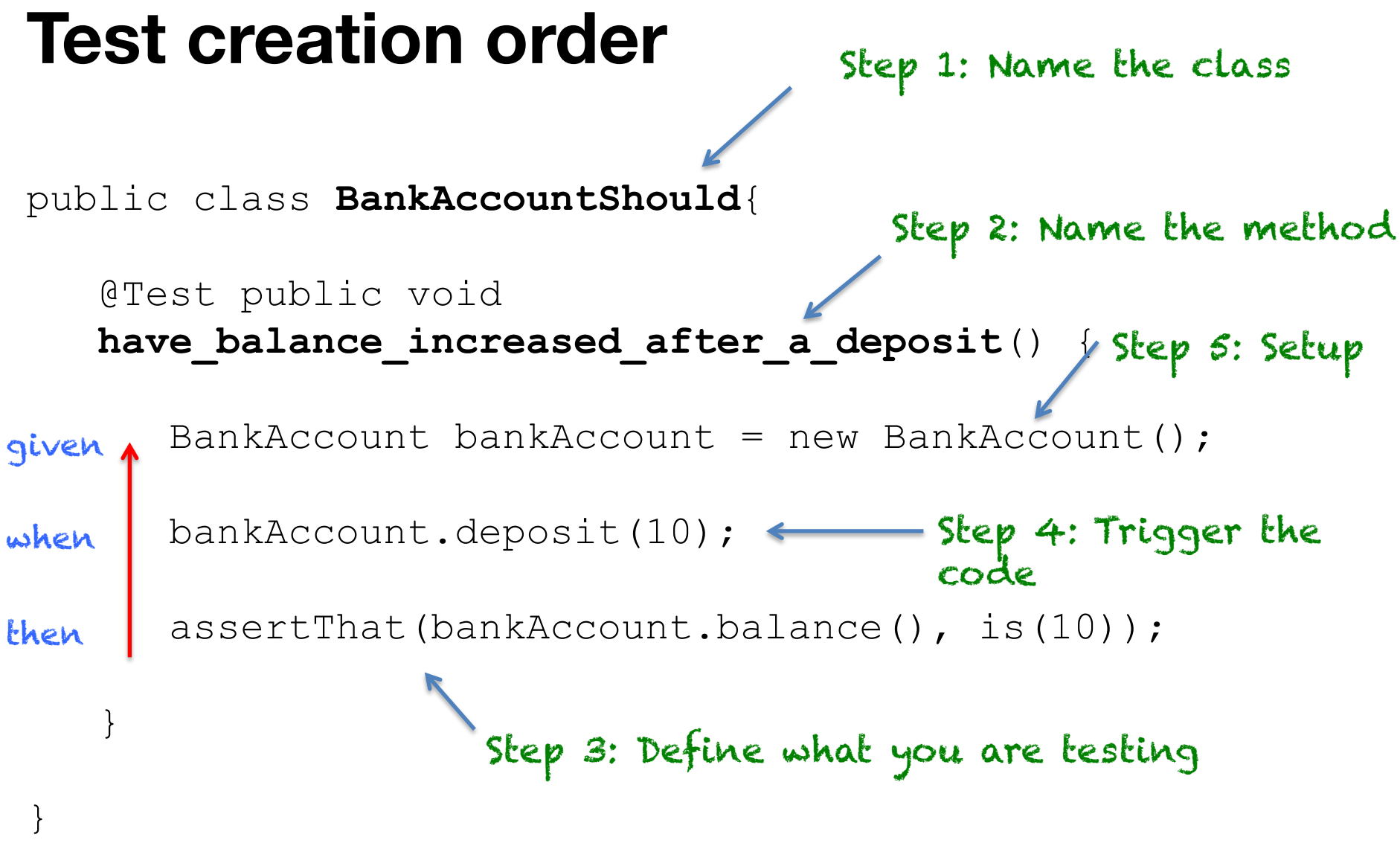



**Tests should clearly specify the behaviour of the class under test.**
**Avoid technical terms.**

# Test method structure

```
public class BankAccountShould{

    @Test public void
    have_balance_of_zero_when_created() {
        BankAccount bankAccount = new BankAccount();

        assertThat(bankAccount.balance(), is(0));
    }

    @Test public void
    have_the_balance_increased_after_a_deposit() {
given   BankAccount bankAccount = new BankAccount();

when    bankAccount.deposit(10);

then    assertThat(bankAccount.balance(), is(10));
    }

}
```

# Test creation order




Production code should be created from the tests.

# Bad naming used on tests

```
BankAccount testee = new BankAccount();
BankAccount sut = new BankAccount();
BankAccount ba = new BankAccount();
```

```
test_deposit_works() {…}
test_deposit_works_correctly() {…}
test_deposit() {…}
check_balance_after_deposit() {…}
```

# Bad naming used on tests

```
BankAccount testee = new BankAccount();
BankAccount sut = new BankAccount();
BankAccount ba = new BankAccount();
```

If it is a bank account, call it "bankAccount"

```
test_deposit_works() {…}
test_deposit_works_correctly() {…}
test_deposit() {…}
check_balance_after_deposit() {…}
```

Test methods should indicate the expected behaviour

```
increase_the_balance_after_a_deposit()
```

# Tip for professional TDDers: Split your screen

```
public class MyClassShould {

    @Test public void
    do_something_useful() {
    }

}
```

```
public class MyClass {

    public void
    some_behaviour() {
    }

}
```

Don’t rush.
Write code you are proud of.

# E.1 – TDD lifecycle & naming

**Objective**
- Introduce naming convention
- Create production code from test
- Start from assertion
- Tip for deciding the first test to write: The simplest possible.

**Problem description: Stack**

Implement a Stack class with the following public methods:

```
+ void push(Object object)
+ Object pop()
```

Stack should throw an exception if popped when empty.

# Stack demo

# E.2 – Test-Driving Algorithms

**Objective**

- Grow an algorithm bit by bit
- Delay treating exceptions (in this case, because they are more complex)
- Intentionally cause duplication
- Focus on simple structures first

**Problem description: Roman Numerals Converter**

Implement a Roman numeral converter. The code must be able to take decimals up to 3999 and convert to their roman equivalent.

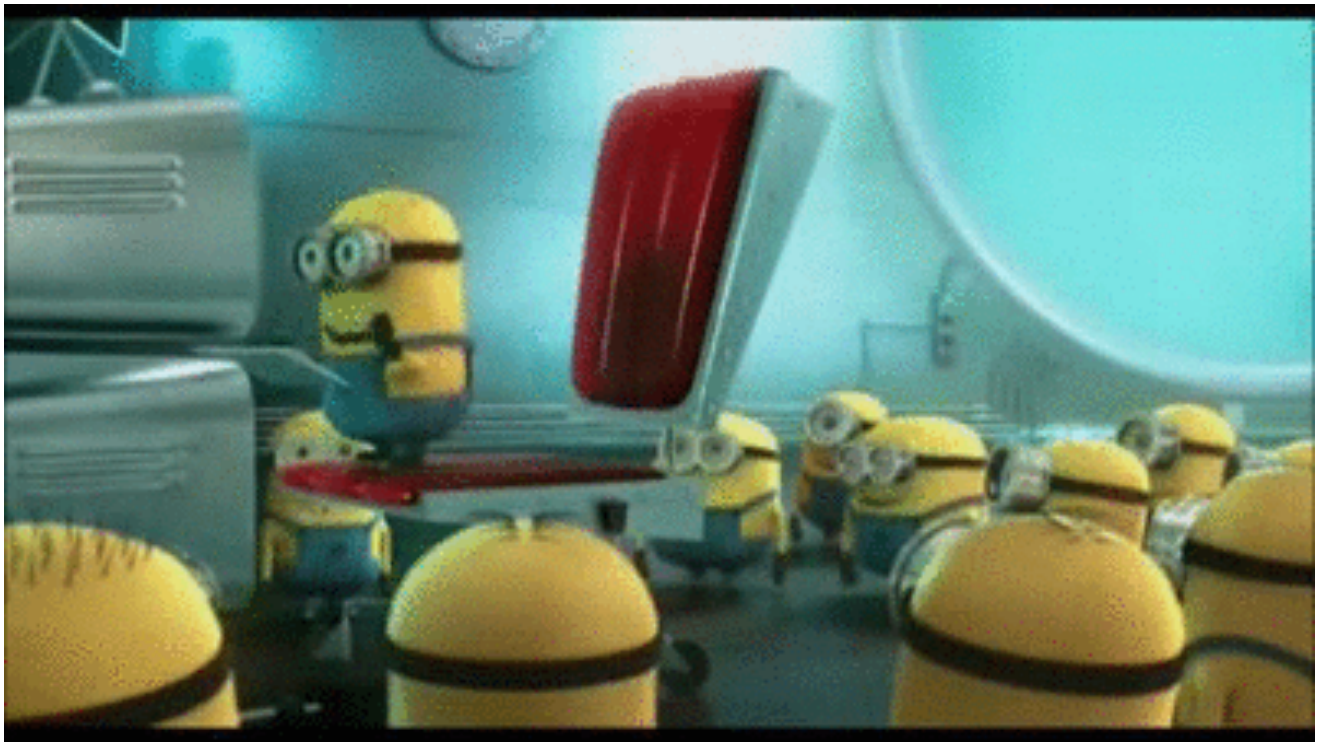
**Examples**

**1 - I**
**5 - V**
**10 - X**
**50 - L**
**100 - C**
**500 - D**
**1000 - M**
**2499 - MMCDXCIX**
**3949 - MMMCMXLIX**

# Transformation Priority Premise - TPP

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 function or algorithm
12. (variable->assignment) replacing the value of a variable.

# Roman Numerals demo



# E.3 – Expressing Business Rules

## Objective

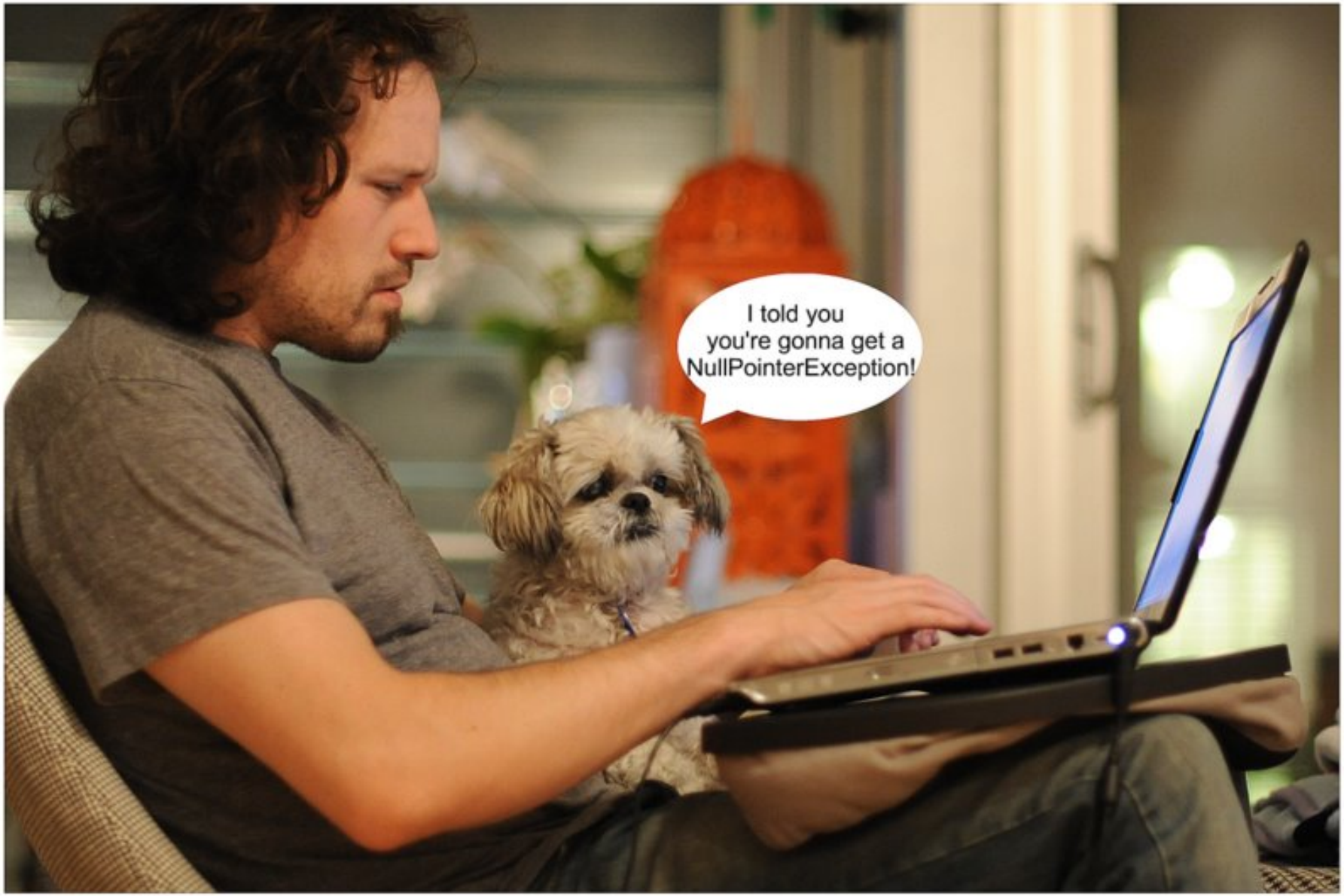
- Clearly express business rules and domain.
- It's OK to write a passing test if it express a valid business rule.

## Problem description: Leap Year

Here are the rules for identifying a Leap Year:

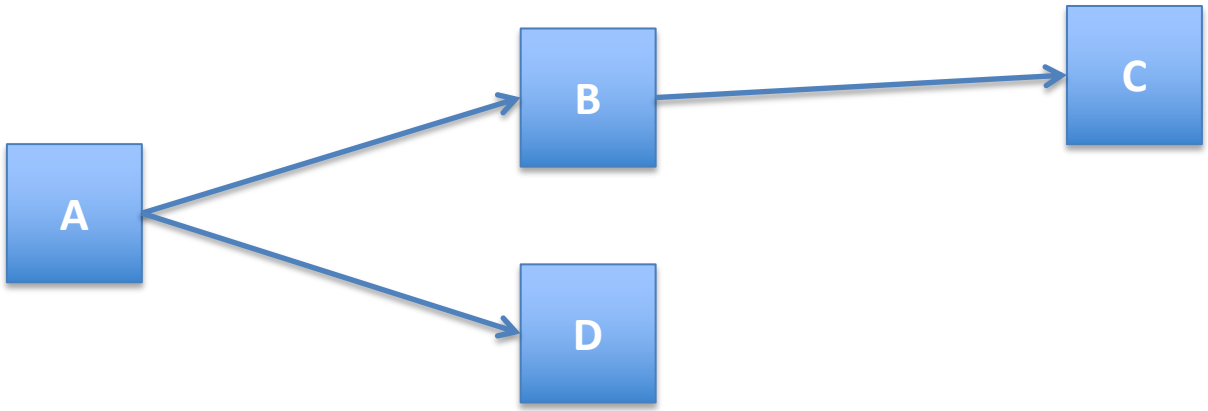
- Is leap year if divisible by 400
- Is NOT leap year if divisible by 100 but not by 400
- Is leap year if divisible by 4

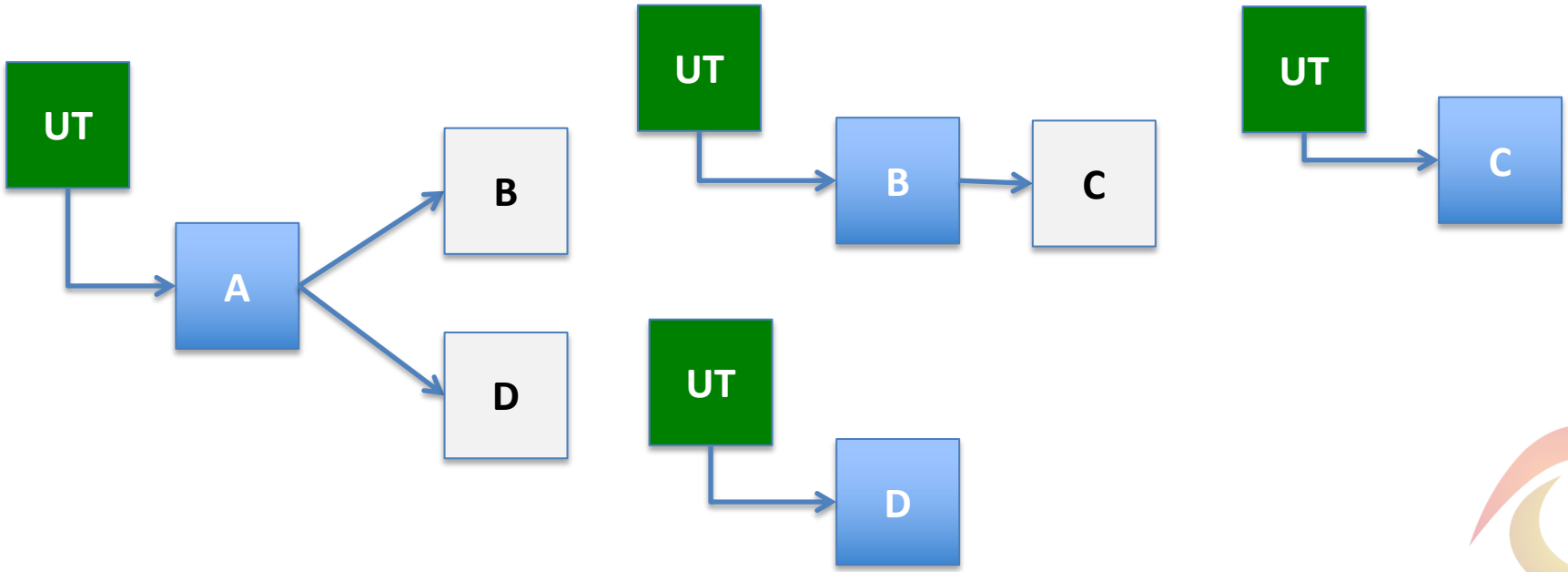
# Leap Year demo




PetraCross.com

# Outside-In TDD: Mocking



# E.4 – Mocking

**Problem description: Payment service**

Given a user wants to buy her selected items
When she submits her payment details
Then we should process her payment

**Acceptance criteria:**
- If the user is not valid, an exception should be thrown.
- Payment should be sent to the payment gateway only when user is valid.

Create a class with the following signature:

```
public class PaymentService {
    public void processPayment(User user,
                               PaymentDetails paymentDetails) {
        // your code goes here
    }
}
```

# End of Day 1

# Agenda

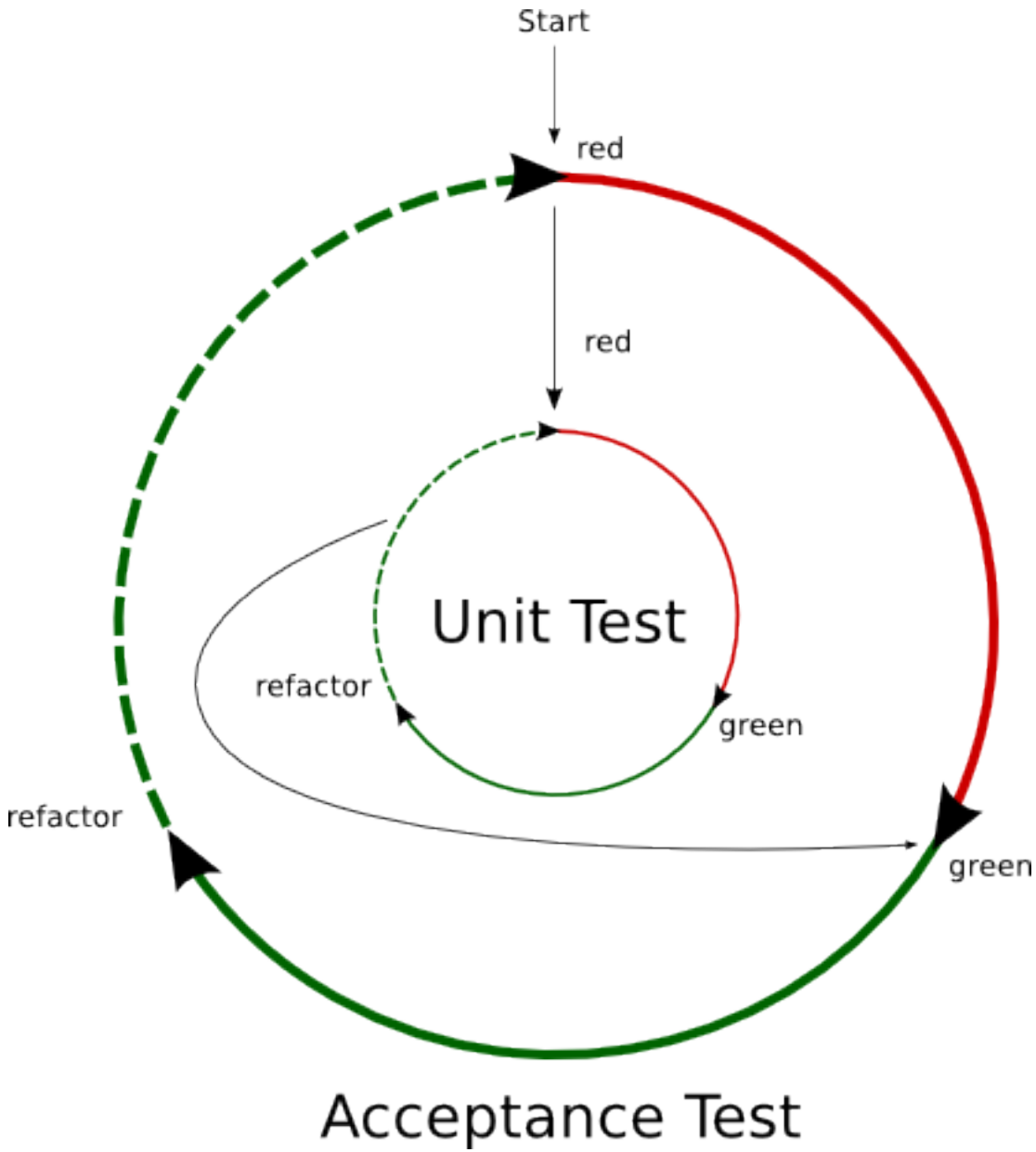
## Day 1

- Introduction to TDD
- TDD Lifecycle and Naming
- Test-Driving algorithms
- Expressing business rules
- Mocking

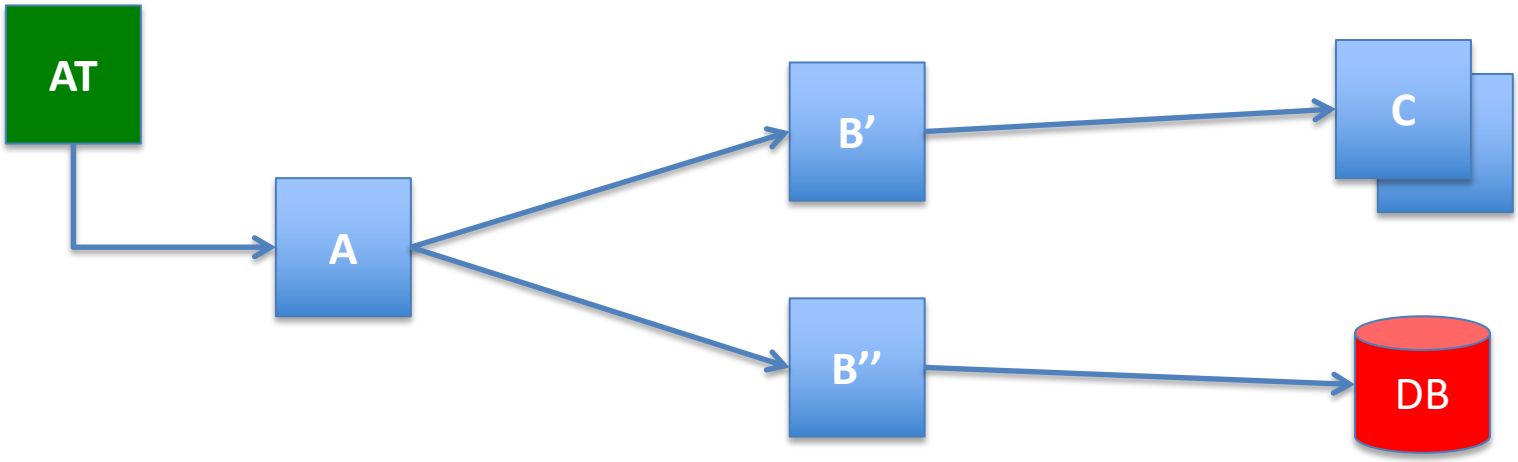
## Day 2

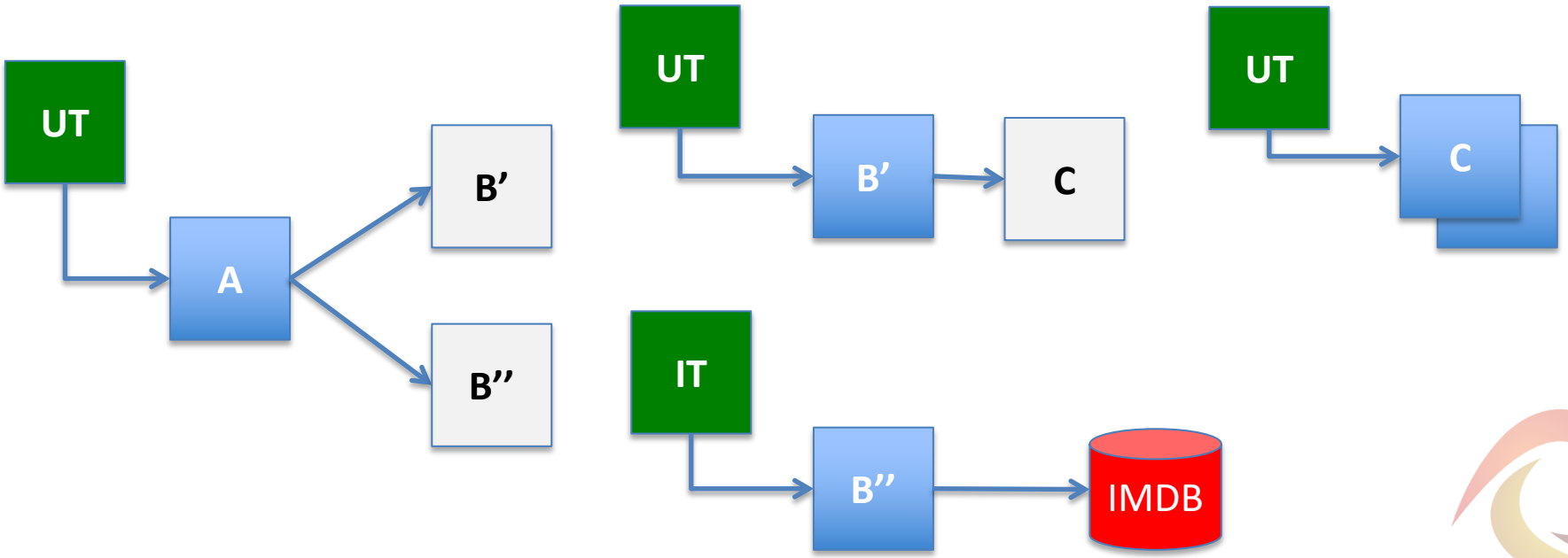
- Outside-In TDD with Acceptance Tests
- Testing and Refactoring Legacy Code

# Outside-In TDD


Start
red
red
Unit Test
refactor
green
refactor
green
Acceptance Test

# Outside-In TDD: Mocking



# E.6 – Outside-In TDD with Acceptance Tests

**Objective:**
Learn and practice the double loop of TDD
Test application from outside, according to side effect

**Problem description: Bank kata**

Create a simple bank application with the following features:

- Deposit money
- Withdraw money
- Print a bank statement to the console.

**Acceptance criteria**
Statement should have the following the format:

```
DATE        | AMOUNT   | BALANCE
10/04/2014 | 500.00   | 1400.00
02/04/2014 | -100.00 | 900.00
01/04/2014 | 1000.00 | 1000.00
```

**Note: Start with an acceptance test**

# Account Service

```
public class AccountService {

    public void deposit(int amount);

    public void withdraw(int amount);

    public void printStatement();

}
```

# E.6 – Outside-In TDD with Acceptance Tests

**Proposed acceptance test starting point;**

```
@RunWith(MockitoJUntiRunner.class)
public class PrintStatementFeature {

    @Mock Console console;

    @Test public void
    print_statement_containing_transactions_in_reverse_chronological_order() {
        AccountService accountService = new AccountService();

        accountService.deposit(1000);
        accountService.withdraw(100);
        accountService.deposit(500);

        accountService.printStatement();

        InOrder inOrder = inOrder(console);
        inOrder.verify(console).printLine("DATE | AMOUNT | BALANCE");
        inOrder.verify(console).printLine("10/04/2014 | 500.00 | 1400.00");
        inOrder.verify(console).printLine("02/04/2014 | -100.00 | 900.00");
        inOrder.verify(console).printLine("01/04/2014 | 1000.00 | 1000.00");
    }
}
```

# Legacy Code

Tight Coupling, Low Cohesion, Feature Envy, Anaemic Domain, and SRP violation

# What does this code do?

```
public List<Trip> getTripsByUser(User user) throws UserNotLoggedInException {
    List<Trip> tripList = new ArrayList<Trip>();
    User loggedUser = UserSession.getInstance().getLoggedUser();
    boolean isFriend = false;
    if (loggedUser != null) {
        for (User friend : user.getFriends()) {
            if (friend.equals(loggedUser)) {
                isFriend = true;
                break;
            }
        }
        if (isFriend) {
            tripList = TripDAO.findTripsByUser(user);
        }
        return tripList;
    } else {
        throw new UserNotLoggedInException();
    }
}
```

# Exercise

## Testing and Refactoring Legacy Code

https://github.com/sandromancuso/trip-service-kata/

# Business Requirements

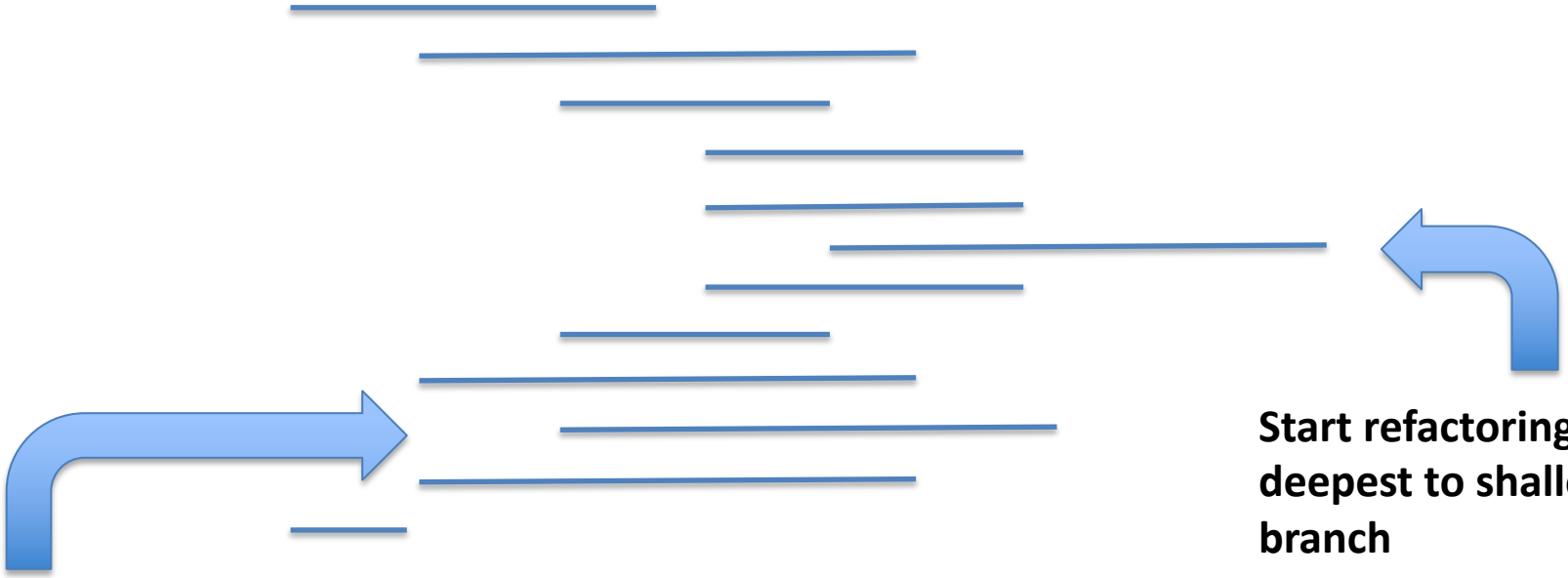
Imagine a social networking website for travellers

- You need to be logged in to see the content
- You need to be a friend to see someone else's trips

# Legacy Code Rules

- You cannot change production code if not covered by tests
  - Just automated refactoring (via IDE) is allowed, in case it is needed for writing a test

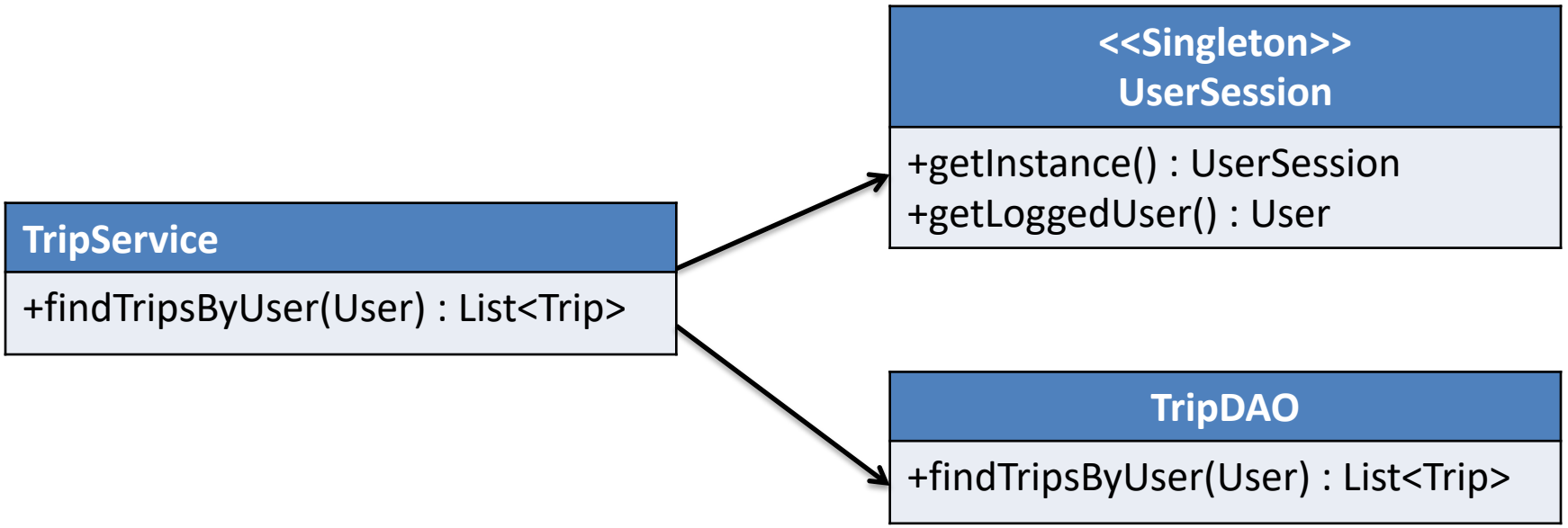
# Working with Legacy Code Tips



**Start refactoring from deepest to shallowest branch**

**Start testing from shallowest to deepest branch**

# Trip Service - Problems



# Have fun!!!

https://github.com/sandromancuso/trip-service-kata/

# Trip Service Kata Tips

**You will need the following three tests**

- `validate_logged_in_user // throws UserNotLoggedInException`
- `does_not_return_any_trips_when_users_are_not_friends`
- `return_trips_when_users_are_friends`

**TIPS**

- Create Seams to isolate dependencies
- When refactoring, ask yourself if the behaviour belongs to the TripService
- There are design issues that must be fixed. Dependencies should be injected.
- As you fix the design, tests become simpler

# DEMO

# What did we cover

## Day 1

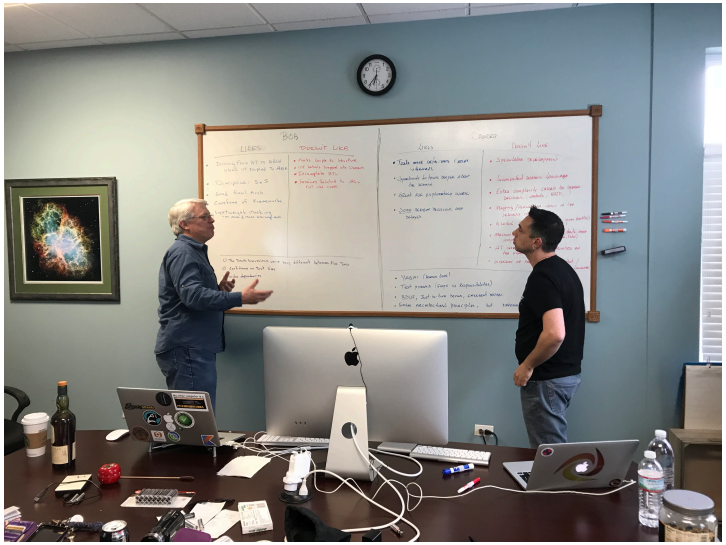
- Introduction to TDD
- TDD Lifecycle and Naming
- Test-Driving algorithms
- Expressing business rules
- Mocking

## Day 2

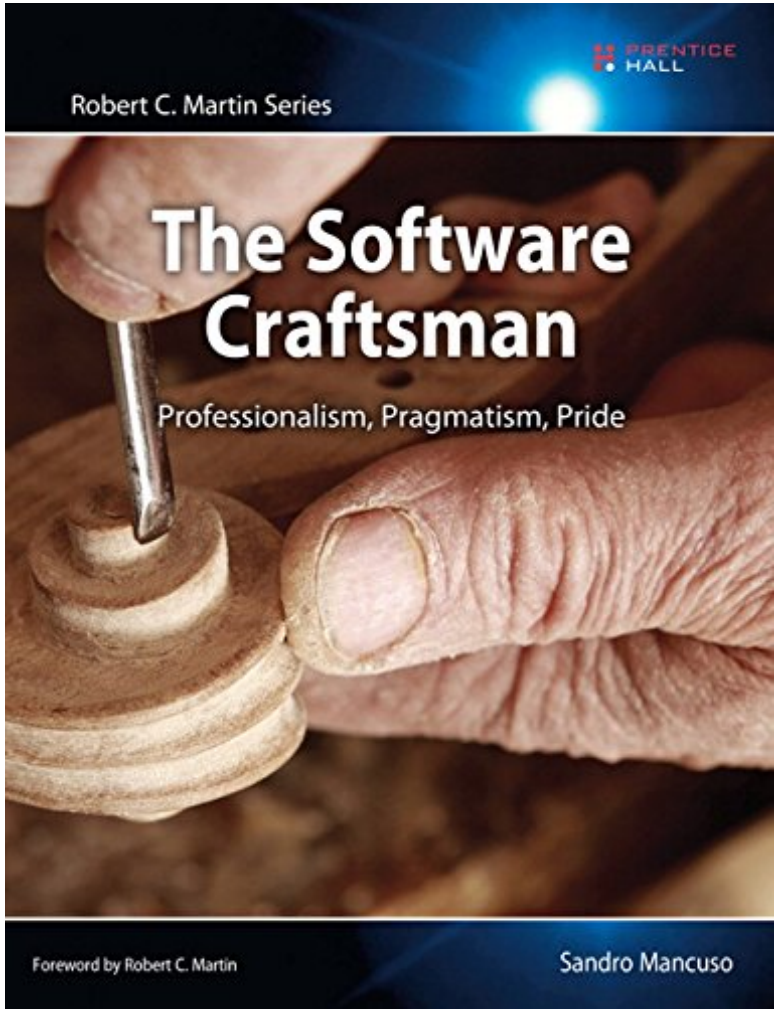
- Outside-In TDD with Acceptance Tests
- Testing and Refactoring Legacy Code

# Retrospective

What did you learn?
What did you like?
Is there anything you can apply tomorrow?
Anything you didn't like?
Any general comments?
Any suggestions?



https://cleancoders.com/videos/comparativeDesign




# Thank you

sandro@codurance.com
@sandromancuso

codurance
craft at heart
http://codurance.com