Parking Lot

	Day 1
09:00 - 09:45	• Welcome • Environment setup • Introduction to TDD
09:45 - 10:45	• TDD Lifecycle and Naming (exercise)
10:45 - 11:00	Break
11:00 - 12:00	• TDD Lifecycle and Naming (exercise)
12:00 - 13:00	Lunch break
13:00 - 14:15	• Classicist TDD • Test-driving algorithms (exercise)
14:15 - 14:30	Break
14:30 - 15:30	• Test-driving algorithms (exercise) cont. • Demo
15:30 - 15:45	Break
15:45 - 17:00	• Using mocks
17:00 - 17:20	• Wrap up
	Day 2
09:00 - 09:45	• Outside-In TDD introduction
09:45 - 10:30	• Outside-in TDD & Acceptance tests (exercise)
10:30 - 10:45	Break
10:45 - 12:00	• Outside-in TDD & Acceptance tests (cont.)
12:00 - 13:00	Lunch break
13:00 - 13:45	• Outside-in TDD & Acceptance tests (cont.)
13:45 - 14:30	• Outside-In Design explanation
14:30 - 14:45	Break
14:45 - 16:00	• Testing & Refactoring Legacy Code (exercise)
16:00 - 16:10	Break
16:10 - 17:00	Legacy Code demo
17:00 - 17:20	Wrap up

Welcome & Introductions

1. Who are you?
    a. What programming language do you use at work?
    b. What programming language will you use for the training?
    c. What is your experience with TDD?
2. A bit about myself

Why this training?

- The goal of this training is to learn how to learn how to craft code that is easy to maintain, enable business agility, and is a pleasure to work with.
- Although we will be talking about TDD for 2 days, this is not a TDD training. This training is about crafting code.
- We care about the process of building code as much as we care about the end result.
- The goal is to start on a long but pleasant journey to mastery.

Forget about what you know... just for 2 days

- There is no silver bullet
    - There is a lot of people I respect who do things differently.
    - I'm not a believer that there is just one way of doing things.
    - I'll be sharing with you what works for me.
- Be selfish with your learning
    - Feel free to ask me questions but don't fight me.
    - You do not learn much trying to convince someone you are right.
    - Take the opportunity to learn who others do things.
- This is a safe space to experiment.
- You can always do these exercises again

Agenda

- Starts slow, with simple exercises.
- Each exercise introduces more complexity
- Day 1 provides the foundation, with simpler and shorter exercises
- Day 2 has only 2 exercises that are closer to what you do on a daily basis

Environment setup

- Make sure everyone can create a new project and run a simple test
- Access Coding Board: http://codingboard.org/boards/craftingcode
- Access Miro Board: https://miro.com/app/board/o9J_kj8eILM=/
    - No need to login / create an account
    - Explain how it works
- Check everyone is on Slack: skills-matter.slack.com

Training mechanics (pairs & zoom breakout rooms)

- You will be working in pairs
- We will rotate pairs on each exercise
- Pairs will work in breakout rooms on each exercise

Running your first test

```
package com.codurance;

import org.junit.Test;

import static org.assertj.core.api.Assertions.assertThat;

public class MyClassShould {

 @Test public void
    my_first_test() {
        assertThat(true).isTrue();
    }

}
```

Naming convention

The combination of the name of the class and name of the test method should clearly describe the behaviour of the class.

Test method structure

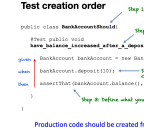
```
public class BankAccountShould{

    @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));
    }

}
```

The name of the public methods should be derived from the domain language used in the description of the test.

Even at a unit level, we should always strive to use the ubiquitous language (domain language).

Test methods describe the behaviour that is important for a client to know, not the internal behaviour of the class or method.

Test creation order



Production code should be created from the tests.

Bad naming used on tests

```
BankAccount testme = new BankAccount();
BankAccount ext = new BankAccount();
BankAccount ba = new BankAccount();
```

If it is a basic 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()
```

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 element)
+ Object pop()

Stack should throw an exception if popped when empty.

StackShould.java

```
package com.codurance.craftingcode.exercise_01_stack;

import org.junit.Before;
import org.junit.Test;

import java.util.EmptyStackException;

import static org.hamcrest.core.Is.is;
import static org.junit.Assert.assertThat;

public class StackShould {

 private static final Object ELEMENT_1 = "Some element";
   private static final Object ELEMENT_2 = "Another element";
   private Stack stack;

   @Before
   public void initialise() {
      stack = new Stack();
   }

   @Test(expected = EmptyStackException.class) public void
   throw_exception_if_popped_when_empty() {
      stack = new Stack();

      stack.pop();
   }

   @Test public void
   pop_the_last_element_pushed() {
      stack.push(ELEMENT_1);
      stack.push(ELEMENT_2);

      assertThat(stack.pop(), is(ELEMENT_2));
   }

   @Test public void
   pop_elements_in_the_reverse_order_they_were_pushed() {
      stack.push(ELEMENT_1);
      stack.push(ELEMENT_2);

      assertThat(stack.pop(), is(ELEMENT_2));
      assertThat(stack.pop(), is(ELEMENT_1));
   }

}
```

Stack.java

```
package com.codurance.craftingcode.exercise_01_stack;

import java.util.ArrayList;
import java.util.EmptyStackException;
import java.util.List;

public class Stack {
 private List<Object> elements = new ArrayList<>();

   public Object pop() {
      if (elements.isEmpty()) {
         throw new EmptyStackException();
      }
      return elements.remove(elements.size() - 1);
   }

   public void push(Object element) {
      this.elements.add(element);
   }
}
```

Test names

```
public class StackShould {

   @Test(expected = EmptyStackException.class) public void
   throw_exception_if_popped_when_empty() {...}

   @Test public void
   pop_the_last_element_pushed() {...}

   @Test public void
   pop_elements_in_the_reverse_order_they_were_pushed() {...}

}
```

- The test names describe the behaviour of the class, from a client's perspective. They do not describe internal behaviour (i.e. add item to the top of the list, removing item from the list, etc).
- Test names clearly explain the behaviour of a Stack, even for those who are not so technical.
- Domain language used in the name of the methods: **pop** and **push.** They are both used as verbs.

Exception name

```
  public Object pop() {
     if (elements.isEmpty()) {
        throw new EmptyStackException();
     }
     return elements.remove(elements.size() - 1);
  }
```

- Exceptions are part of the contract. It allows clients to understand what went wrong.
- Exceptions should be clear enough to avoid developers to have to debug the code. They should not look inside our code in order to understand what went wrong.

Constants & Clean Tests

```
   private static final Object ELEMENT_1 = "Some element";
   private static final Object ELEMENT_2 = "Another element";

   @Test public void
   pop_the_last_element_pushed() {
      stack.push(ELEMENT_1);
      stack.push(ELEMENT_2);

      assertThat(stack.pop(), is(ELEMENT_2));
   }

   @Test public void
   pop_elements_in_the_reverse_order_they_were_pushed() {
      stack.push(ELEMENT_1);
      stack.push(ELEMENT_2);

      assertThat(stack.pop(), is(ELEMENT_2));
      assertThat(stack.pop(), is(ELEMENT_1));
   }

}
```

- Constants are used to clarify intent, not to optimise the code.
- Variable declarations are removed from the test method. They create a lot of noise.
- Only code that is important for the test remains in the test. Everything else is removed.
- Tests must tell a story and be very focused.

Consistency

- **Elements**: Name of parameters, constants, internal attributes uses the same language.

How many tests?

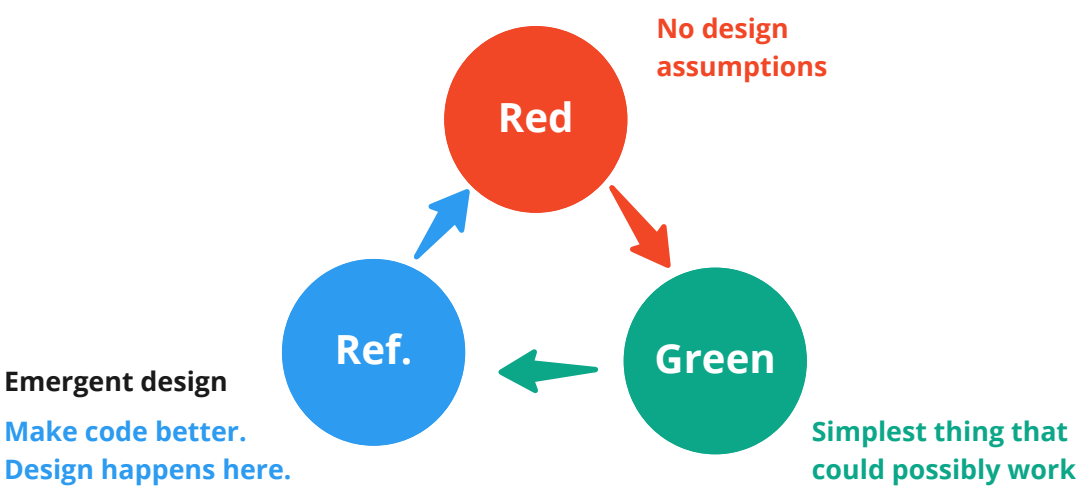
```
public class StackShould {

   @Test(expected = EmptyStackException.class) public void
   throw_exception_if_popped_when_empty() {...}

   @Test public void
   pop_the_last_element_pushed() {...}

   @Test public void
   pop_elements_in_the_reverse_order_they_were_pushed() {...}

}
```

- Enough so that you and your pair are confident you have all the functionality covered and a bug cannot be introduced without breaking a test.
- Push 2, pop 3: This test should go green if introduced now. Every time a test goes green we should ask if we need the test. In doubt, write the test.
- Don't over do it. We can make the last test to create random elements and then pop, instead of using only 2 elements. It it worthy? What is the likelihood of someone having the wrong implementation with the existing solution?

Classicist	Outside-In
Chicago School	London School
Inside-out	Mockist
Design approach: Emergent design	Design approach: Just-in time design

- Story of the C3 project, XP, Kent Beck, Uncle Bob
- Explain the evolution in London, early 2000s

Classicist / Chicago School



Four Rules of Simple Design (4RSD)

1. Passes the tests
2. Reveals intention (should be easy to understand)
3. No duplication (DRY)
4. Fewest elements (remove anything that doesn't serve the three previous rules)

- Work in baby steps. One small step (test) at a time.
- Don't make design assumptions. Make it work and let the code tell you what needs to be improved.
- Let the tests and emergent design guide your progress
- Exploratory style to design.

E.2 – Test-Driving Algorithms

Problem description: Roman Numerals Converter

Implement a Roman numeral converter. The code must be able to take numbers from 1 to 3999 and convert to their roman equivalent.

Objective

- Grow an algorithm bit by bit, one test at a time.
- Do not design up-front. Let the tests and code guide you.
- Focus on simple structures first
- Focus on the general cases first. Delay exceptional cases.
- Intentionally cause duplication as it makes easier to refactor.

A few examples

1 - I	**5 - V**	**10 - X**	**50 - L**
100 - C	**500 - D**	**1000 - M**	76 - LXXVI
493 - CDXCIII	648 - DCXLVIII	2499 - MMCDXCIX	3949 - MMMCMXLIX

Parameterised tests

NUnit:
https://docs.nunit.org/articles/nunit/writing-tests/attributes/testcase.html?q=testcase

JUnit 5:
https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests-sources-CsvSource

JUnit 4 (JUnitParams)
https://github.com/Pragmatists/JUnitParams

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.
```

- Refactorings have counterparts called Transformations.
- Refactorings are simple operations that change the structure of code without changing its behavior.
- 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.

(from Robert C. Martin - Uncle Bob)

RomanNumeralConverterShould.java

```
package com.codurance;

import junitparams.JUnitParamsRunner;
import junitparams.Parameters;
import org.junit.Test;
import org.junit.runner.RunWith;

import static com.codurance.RomanNumeralConverter.romanFor;
import static org.assertj.core.api.Assertions.assertThat;

@RunWith(JUnitParamsRunner.class)
public class RomanNumeralConverterShould {

 @Test
    @Parameters({
            "1, I",
            "2, II",
            "3, III",
            "4, IV",
            "5, V",
            "7, VII",
            "9, IX",
            "10, X",
            "18, XVIII",
            "30, XXX",
            "40, XL",
            "50, L",
            "76, LXXVI",
            "90, XC",
            "100, C",
            "400, CD",
            "493, CDXCIII",
            "500, D",
            "648, DCXLVIII",
            "900, CM",
            "1000, M",
            "2497, MMCDXCVII",
            "2748, MMDCCXLVIII",
            "3949, MMMCMXLIX",
    })
    public void
    convert_arabic_numbers_into_roman_numerals(int arabic, String
roman) {
        assertThat(romanFor(arabic)).isEqualTo(roman);
    }

}
```

RomanNumeralConverter.java

```
package com.codurance;

public class RomanNumeralConverter {
 public static String romanFor(int arabic) {
        String roman = "";

        for (ArabicToRoman arabicToRoman : ArabicToRoman.values()) {
            while (arabic >= arabicToRoman.arabic) {
                roman += arabicToRoman.roman;
                arabic -= arabicToRoman.arabic;
            }
        }

        return roman;
    }

    enum ArabicToRoman {
        THOUSAND(1000, "M"),
        NINE_HUNDRED(900, "CM"),
        FIVE_HUNDRED(500, "D"),
        FOUR_HUNDRED(400, "CD"),
        HUNDRED(100, "C"),
        NINETY(90, "XC"),
        FIFTY(50, "L"),
        FORTY(40, "XL"),
        TEN(10, "X"),
        NINE(9, "IX"),
        FIVE(5, "V"),
        FOUR(4, "IV"),
        ONE(1, "I");

        private final int arabic;
        private final String roman;

        ArabicToRoman(int arabic, String roman) {
            this.arabic = arabic;
            this.roman = roman;
        }
    }
}
```

Tests

- The class has only one behaviour from a client's perspective. If that is the case, it should have a single test.
- The test name should express the behaviour of the class—the general case, not examples of the behaviour.
- Duplication in tests should be removed, similar to production.
- In cases like this, use a parameterised test. They are good for when you have multiple examples of the same behaviour.
- Here we assume that Roman Numerals is a common domain and the mechanics of the numeric system does not need to be explained by tests.
- If we were creating our own numeric system, we probably would create separate tests explaining its different characteristics.

Classicist TDD mechanics

- Good for exploratory work, when we know inputs and outputs but we don't know how the code will look like.
- Perfect for growing algorithms, data transformation, parsers, etc, bit-by-bit safely.
- Design emerges from the code. No design upfront needed.
- Design is guided by simple concepts like 4RSD, DRY, Clean Code and SOLID.

Caveats

- This style helps you to come up with a solution, but not necessarily the best solution.
- Imagine if this was a sorting exercise. This TDD style would probably lead to a Bubble Sort, but almost certainly not to Merge or Quick sort.
- Performance, parallelisation, immutability need to be designed upfront. This style of TDD will not lead you to these concepts.

E.4 – Mocking

Problem description: Payment service

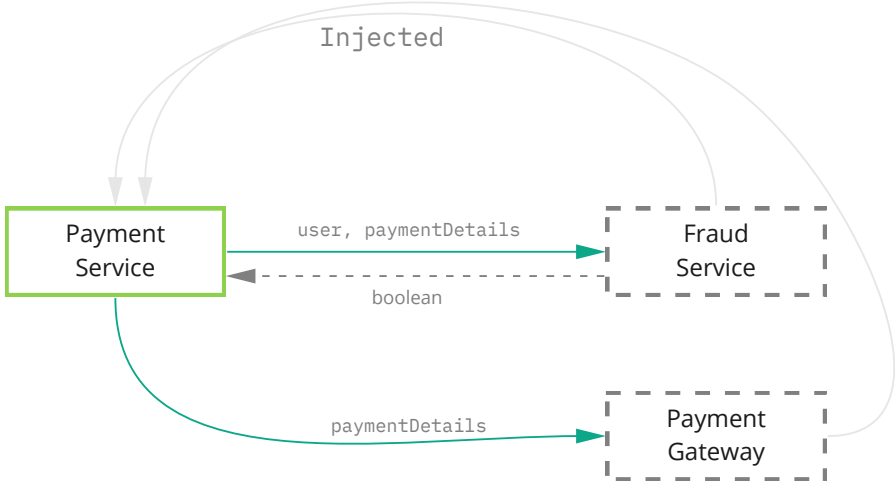
Given a user wants to buy her selected items
When she submits her payment details
Then we should process her payment

Acceptance criteria:

- If payment is fraudulent, an exception should be thrown.
- Payment should only be sent to the payment gateway when payment is legit.

Create a class with the following signature:

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



Testing approach

- Test drive a working implementation of PaymentService.
- Write the tests assuming that all the classes you need already exist.
- Create all the production code from your test code, using your IDE if possible.
- Do not add any implementation for the methods on FraudService and PaymentGateway. Make the methods throw an exception, just to make sure you are not using the real classes in your PaymentService tests.

A bit of context before jumping onto the next exercise.

- For large applications, we cannot use the approach we used until now.
- A large application is composed by multiple modules and it would be very difficult to build a full enterprise app from a single class, just relying on refactoring.
- We need to design a bit up-front.
- We need to isolate the "module" we are building from other modules we need to use or build.
- We need to keep the unit under test small and under control.
- Testing large units make the diagnostic very difficult when the test breaks.
- It is possible to build large applications one test at a time. But it is much easier if we define our modules before we write the tests and production code.
- We can always refactor later, in case we got the design wrong.

Mockito docs
https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html

MOQ docs:
https://github.com/Moq/moq4/wiki/Quickstart

PaymentServiceShould.java

```
package com.codurance.craftingcode.exercise_04_make_payment_fraud_gateway;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

import static org.assertj.core.api.Assertions.assertThatExceptionOfType;
import static org.mockito.BDDMockito.given;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.verifyZeroInteractions;

@RunWith(MockitoJUnitRunner.class)
public class PaymentServiceShould {

    private static final User USER = new User();
    private static final PaymentDetails PAYMENT_DETAILS =
                                            new PaymentDetails();
    private static final PaymentDetails FRAUDULENT_PAYMENT_DETAILS = new PaymentDetails();

    @Mock FraudService fraudService;
    @Mock PaymentGateway paymentGateway;

    private PaymentService paymentService;

    @Before
    public void initialise() {
      paymentService = new PaymentService(fraudService, paymentGateway);
    }

    @Test public void
    throw_exception_when_fraud_is_detected() {
        given(fraudService.isFraudulent(USER, FRAUDULENT_PAYMENT_DETAILS)).willReturn(true);

        assertThat(exceptionOfType(FraudulentPaymentException.class)
                .isThrownBy(() -> paymentService.processPayment(USER, FRAUDULENT_PAYMENT_DETAILS));

        verify(fraudService).isFraudulent(USER, FRAUDULENT_PAYMENT_DETAILS);
    }

    @Test public void
    process_payment_details_when_payment_is_legit() throws FraudulentPaymentException {
        given(fraudService.isFraudulent(USER, PAYMENT_DETAILS)).willReturn(false);

        paymentService.processPayment(USER, PAYMENT_DETAILS);

        verify(paymentGateway).payWith(PAYMENT_DETAILS);
    }

    @Test
    public void
    not_process_payment_when_fraud_is_detected() {
        given(fraudService.isFraudulent(USER, FRAUDULENT_PAYMENT_DETAILS)).willReturn(true);

        assertThat(exceptionOfType(FraudulentPaymentException.class)
                .isThrownBy(() -> paymentService.processPayment(USER, FRAUDULENT_PAYMENT_DETAILS));

        verifyZeroInteractions(paymentGateway);
    }

}
```

PaymentService.java

```
package com.codurance.craftingcode.exercise_04_make_payment_fraud_gateway;

public class PaymentService {

    private FraudService fraudService;
    private PaymentGateway paymentGateway;

    public PaymentService(FraudService fraudService,
                          PaymentGateway paymentGateway) {
        this.fraudService = fraudService;
        this.paymentGateway = paymentGateway;
    }

    public void processPayment(User user,
                               PaymentDetails paymentDetails) throws FraudulentPaymentException {
        if (fraudService.isFraudulent(user, paymentDetails)) {
            throw new FraudulentPaymentException();
        }
        paymentGateway.payWith(paymentDetails);
    }
}
```

FraudService.java

```
package com.codurance.craftingcode.exercise_04_make_payment_fraud_gateway;

public class FraudService {

    public boolean isFraudulent(User user, PaymentDetails paymentDetails) {
        throw new UnsupportedOperationException();
    }

}
```

PaymentGateway.java

```
package com.codurance.craftingcode.exercise_04_make_payment_fraud_gateway;

public class PaymentGateway {

    public void payWith(PaymentDetails paymentDetails) {
        throw new UnsupportedOperationException();
    }

}
```

User.java

```
package com.codurance.craftingcode.exercise_04_make_payment_fraud_gateway;

public class User {
}
```

PaymentDetails.java

```
package com.codurance.craftingcode.exercise_04_make_payment_fraud_gateway;

public class PaymentDetails {
}
```

FraudulentPaymentException.java

```
package com.codurance.craftingcode.exercise_04_make_payment_fraud_gateway;

public class FraudulentPaymentException extends Exception {
}
```

Strict and non-strict mocking frameworks

```
@Test
public void
not_process_payment_when_fraud_is_detected() {
   given(fraudService.isFraudulent(USER, FRAUDULENT_PAYMENT_DETAILS)).willReturn(true);

   assertThatExceptionOfType(FraudlentPaymentException.class)
           .isThrownBy(() -> paymentService.processPayment(USER, FRAUDULENT_PAYMENT_DETAILS));

   verifyZeroInteractions(paymentGateway);
}
```

Without the test above, production code below would not break any test.

```
public void processPayment(User user, PaymentDetails paymentDetails) throws FraudlentPaymentException {
    paymentGateway.makePayment(user, paymentDetails);
    paymentGateway.makePayment(user, paymentDetails);
    if (fraudService.isFraudulent(user, paymentDetails)) {
        throw new FraudlentPaymentException();
    }
    paymentGateway.payWith(paymentDetails);
    paymentGateway.makePayment(user, paymentDetails);
    paymentGateway.makePayment(user, paymentDetails);
    paymentGateway.makePayment(user, paymentDetails);
}
```

Without the test above, calling the PaymentGateway before checking for fraud would not break any tests.

```
public void processPayment(User user, PaymentDetails paymentDetails) throws FraudlentPaymentException {
    paymentGateway.payWith(paymentDetails);
    if (fraudService.isFraudulent(user, paymentDetails)) {
        throw new FraudlentPaymentException();
    }
}
```

Where to throw the exception?

Case 1: Throw it inside the PaymentService:

```
public void processPayment(User user,
                           PaymentDetails paymentDetails) throws FraudlentPaymentException {
    if (fraudService.isFraudulent(user, paymentDetails)) {
        throw new FraudlentPaymentException();
    }
    paymentGateway.payWith(paymentDetails);
}
```

Case 2: Throw it inside the FraudService:

```
public void processPayment(User user,
                           PaymentDetails paymentDetails) throws FraudlentPaymentException {
    fraudService.isFraudulent(user, paymentDetails);
    paymentGateway.payWith(paymentDetails);
}
```

- Feature-wise, there is no difference. Design-wise, there are significant differences.
- Exceptions are a flow control construct. When thrown, they interrupt the flow and move the control up the stack.
- As a guideline, exceptions should be thrown by classes that have the responsibility to control the flow. In this case, PaymentService is the class controlling the flow.
- Classes at the bottom of the flow should not throw exceptions. They should provide the information for the client classes and let them decide what to do.
- In this case, the FraudService should tell its clients if there is or not a fraud. It's up to the clients to decide what they are going to do with that information. It should not be up to the FraudService to break its clients flows or force them to catch exceptions.
- This approach makes FraudService more reusable and "user" friendly.

Mocks are a design tool

- Mocks are a design tool, not a testing tool.
- They are used to design the collaboration between two classes.
- Mocks define the amount of responsibility you want to keep on it class and how they are going to collaborate.
- Mocks allow us to keep the tests focused and build large applications in very small steps.
- Tests will only break (as they should) if we change how two classes collaborate.

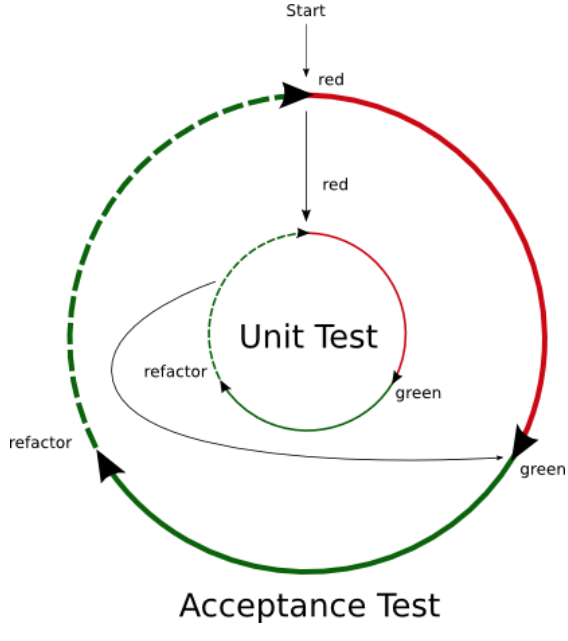
A bit of history: What is the London school of TDD? Why Outside-In TDD?

- C3 project, London connection
- Challenges of building enterprise systems with large teams
- Agile / XP group of in London
- User stories, feature description, automation
- Early days of BDD and the origins of the outside-in approach
    - Start with acceptance test, continue with unit test
- History of Cucumber

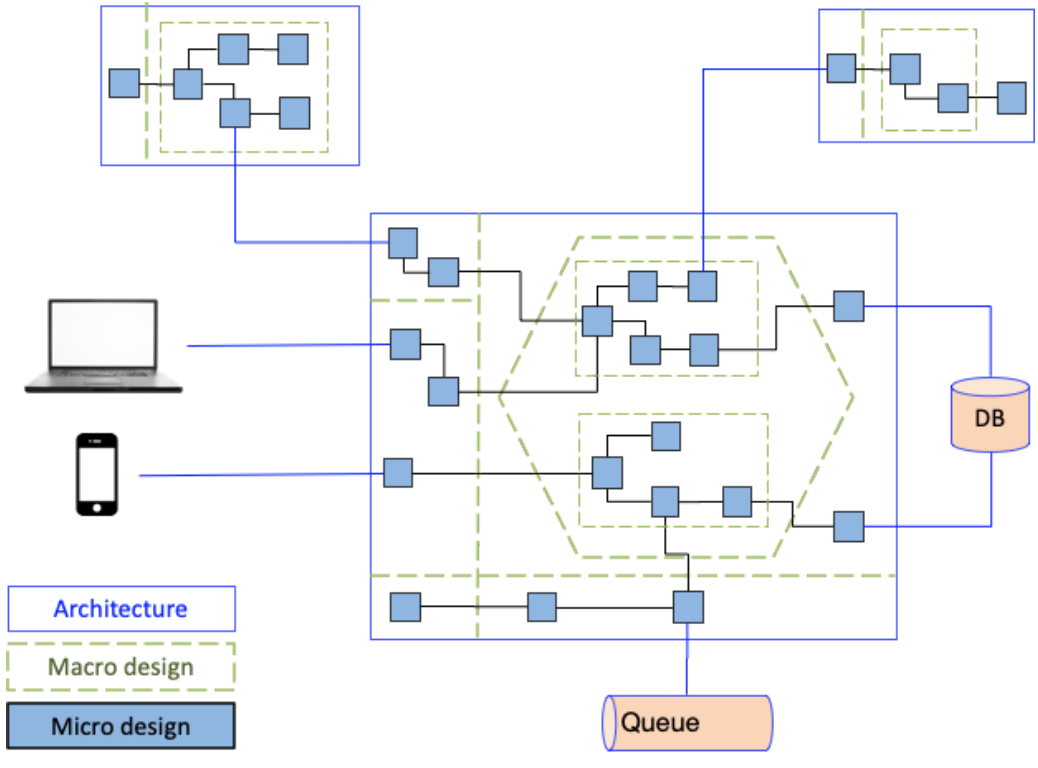
A few lessons learned

- For large applications, we cannot use the approach we used until now.
- A large application is composed by multiple modules and it would be very difficult to build a full enterprise app from a single class, just relying on refactoring.
- We need to design a bit up-front.
- We need to isolate the "module" we are building from other modules we need to use or build.
- We need to keep the unit under test small and under control.
- Testing large units make the diagnostic very difficult when the test breaks.
- It is possible to build large applications one test at a time. But it is much easier if we define our modules before we write the tests and production code.
- We can always refactor later, in case we got the design wrong.

The double loop of TDD

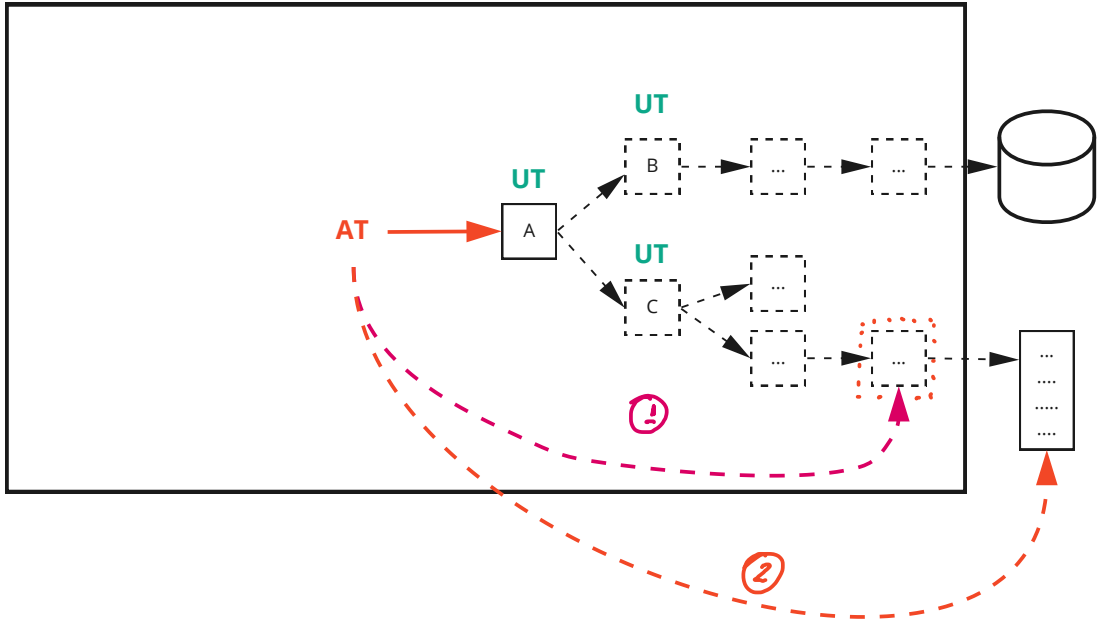


How would we test-drive a system to look like this?

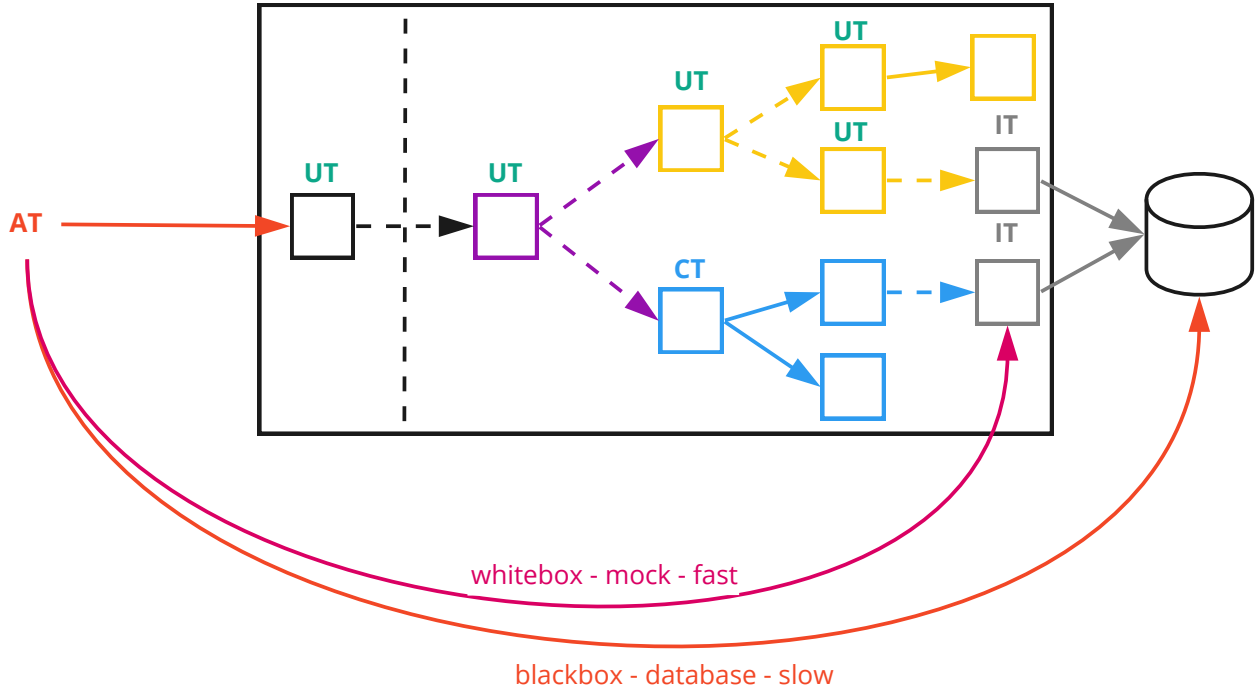


There is a need for some design thinking.

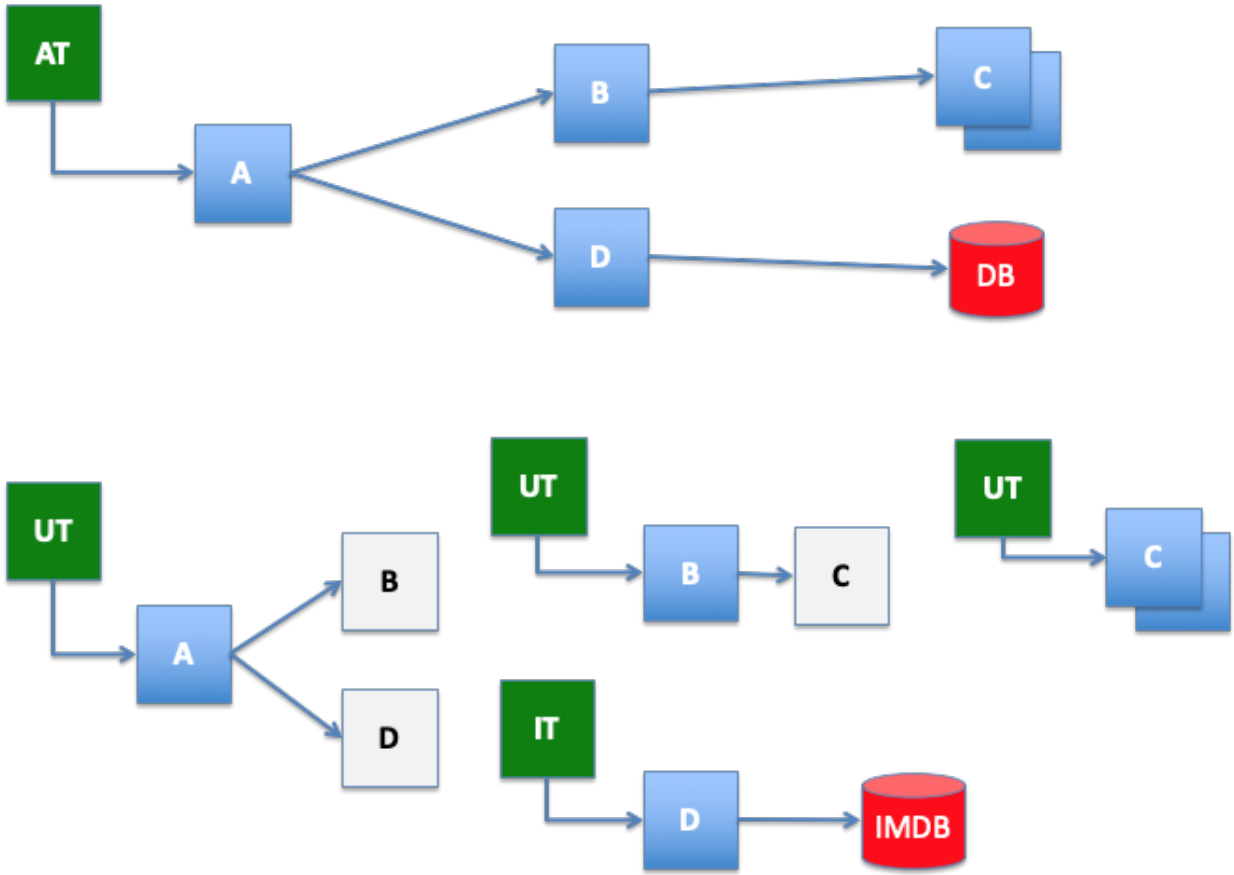
For the next exercise...



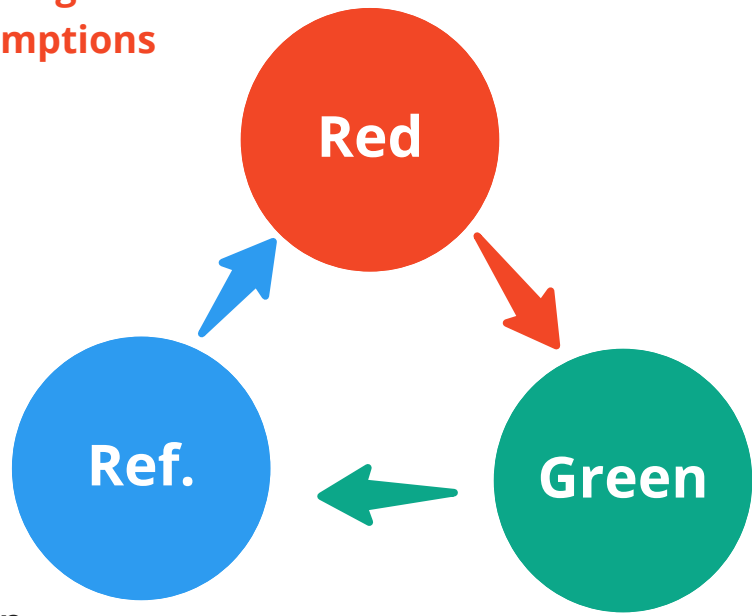
The outside-in approach



Test boundaries - Outside-In



Classicist / Chicago School



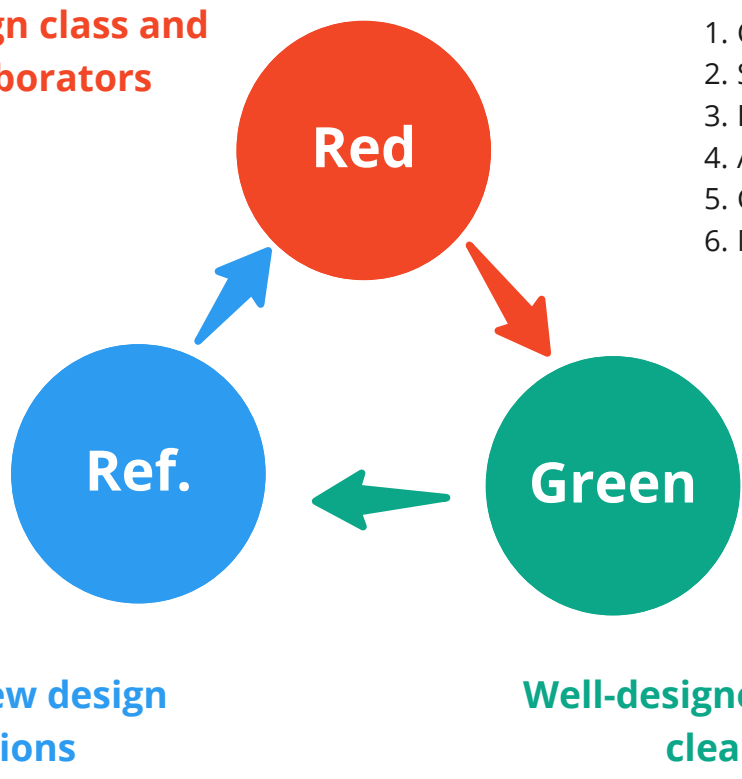
Relies on Four Rules of Simple Design (4RSD)

1. Passes the tests
2. Reveals intention (should be easy to understand)
3. No duplication (DRY)
4. Fewest elements (remove anything that doesn't serve the three previous rules)

- Work in baby steps. One small step (test) at a time.
- Don't make design assumptions. Make it work and let the code tell you what needs to be improved.
- Let the tests and emergent design guide your progress
- Exploratory style to design.

Outside-In / London School

Just-in-time design



Relies on:

1. Coupling & Cohesion
2. SOLID principles
3. DDD
4. Architectural Patterns
5. Clean Architecture
6. Modularisation strategies

- Work in larger design steps.
- Use the test to design the class and respective interactions.
- Let the tests and your design experience guide your progress.
- Assertive style to design.

E.5 – 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:

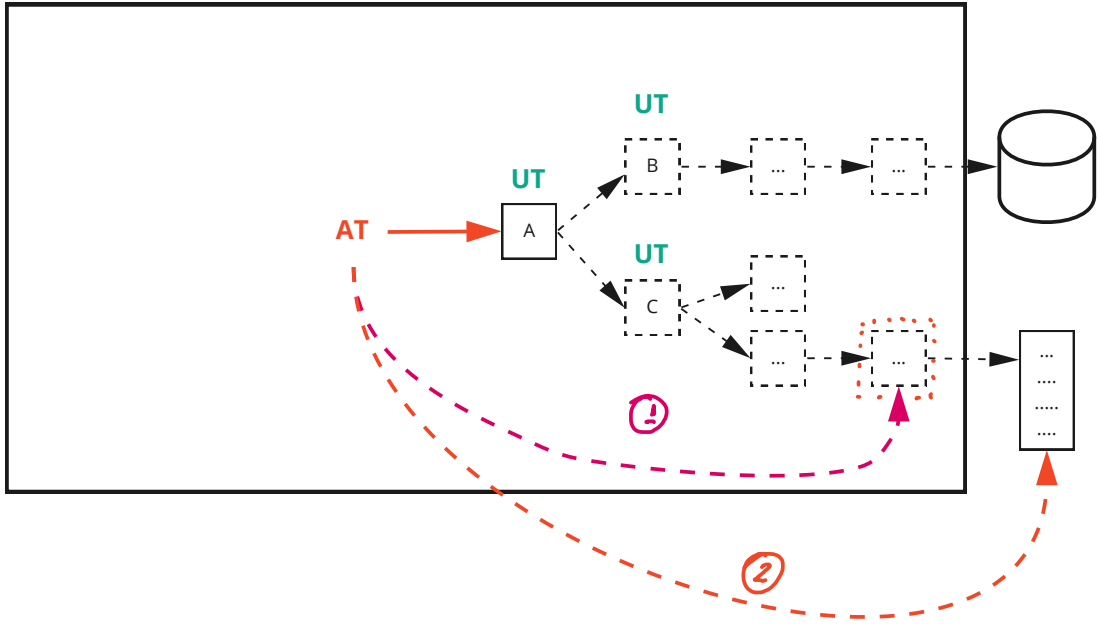
```
  DATE       | AMOUNT  | BALANCE
  10/04/2020 | 500.00  | 1400.00
  02/04/2020 | -100.00 | 900.00
  01/04/2020 | 1000.00 | 1000.00
```

Note: Start with an acceptance test

Starting class:

```
   public class AccountService {

       public void deposit(int amount);

       public void withdraw(int amount);

       public void printStatement();

   }
```

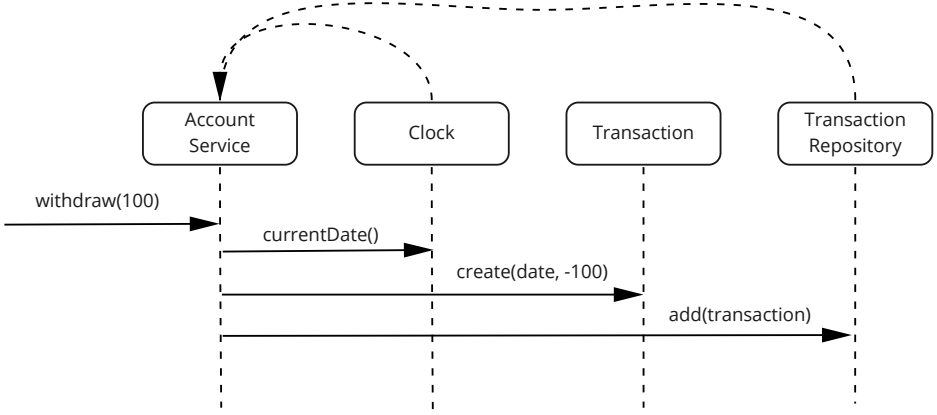
For the next exercise...



Proposed acceptance test starting point

```
import org.junit.runner.RunWith;
import org.mockito.InOrder;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.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("10/04/2019 | 500.00 | 1400.00");
      inOrder.verify(console).printLine("02/04/2019 | -100.00 | 900.00");
      inOrder.verify(console).printLine("01/04/2019 | 1000.00 | 1000.00");
   }
}
```

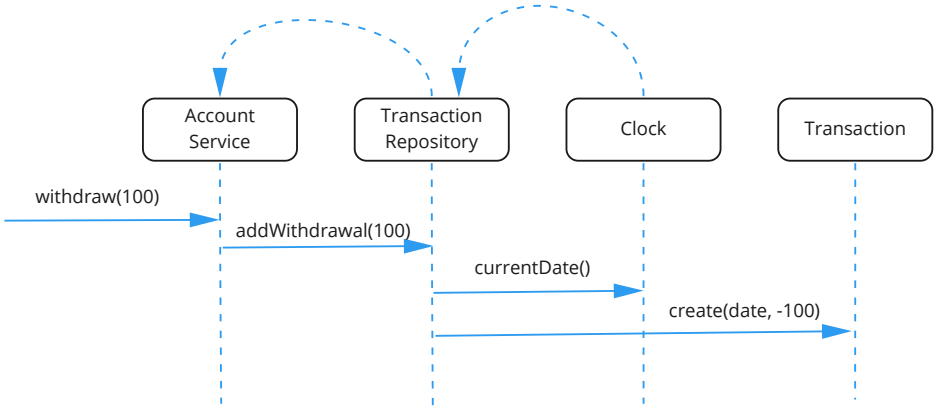
Black approach: natural and simple



Issues with this approach:

- Leak in abstraction. Client (AccountService) needs to know that it needs to flip the sign in order to represent a withdraw transaction.
  - Type system does not inform that.
- Transaction is a domain concept. It is likely that the Transaction will have more attributes in the future. Once the constructor changes, all clients will be broken.
- AccountService has two collaborators. The only reason it needs the Clock is so we can test the Transaction creation.

Blue approach: push behaviour down



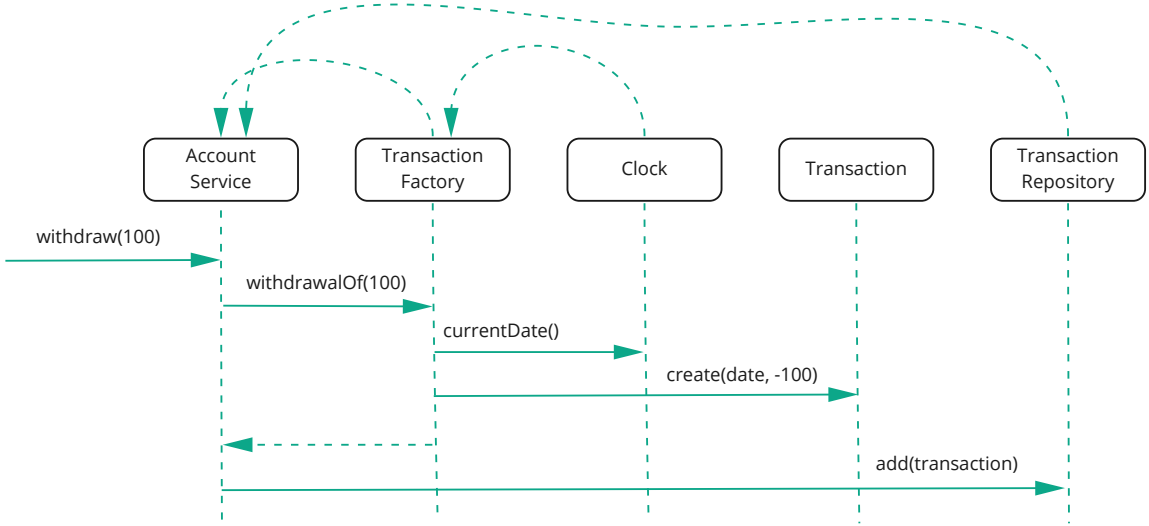
Advantages of this approach:

- It encapsulates the creation of the transaction. Now there is only a single place where transactions are created.
- Removes the leak abstraction. The TransactionRepository provides two clear methods to its clients: addWithdrawal(amount) and addDeposit(amount), both receiving positive amounts.
- Keeps fan-out coupling under control, with classes having one or less collaborators.

Big disadvantage

- Bad design approach. A repository should not have business logic and creating a transaction is business logic. A repository should just receive a domain object (entity, value object) and persist it to the database. Or read data from database, populate a domain object and retrieve it.

Green approach: addressing design issues



Advantages of this approach:

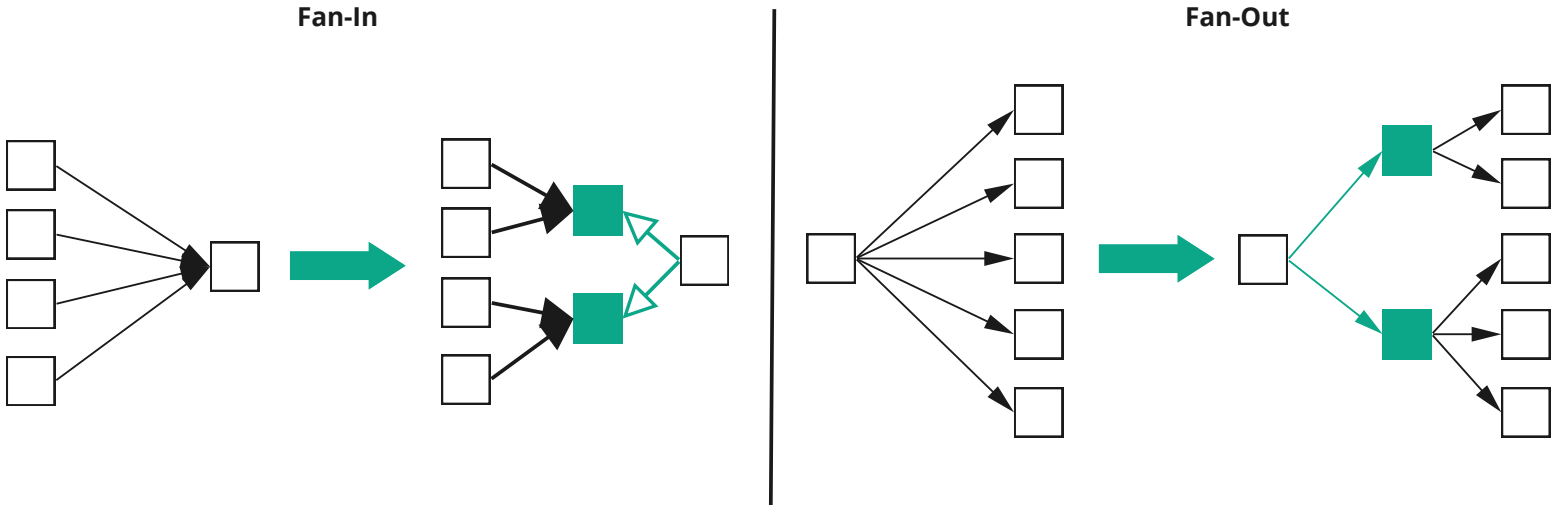
- Solves all the main design issues.
  - Encapsulates the creation of Transactions, allowing the domain concept to evolve without an impact on the rest of the system
  - Makes it clear when a withdrawal or deposit transaction is being created
- TransactionRepository remains business logic free.

Disadvantage

- Solution is over-engineered: 5 classes created to solve a very simple problem.

Advice

- Go for the black approach and refactor towards the green approach if we ever have the need to create transactions in multiple places.



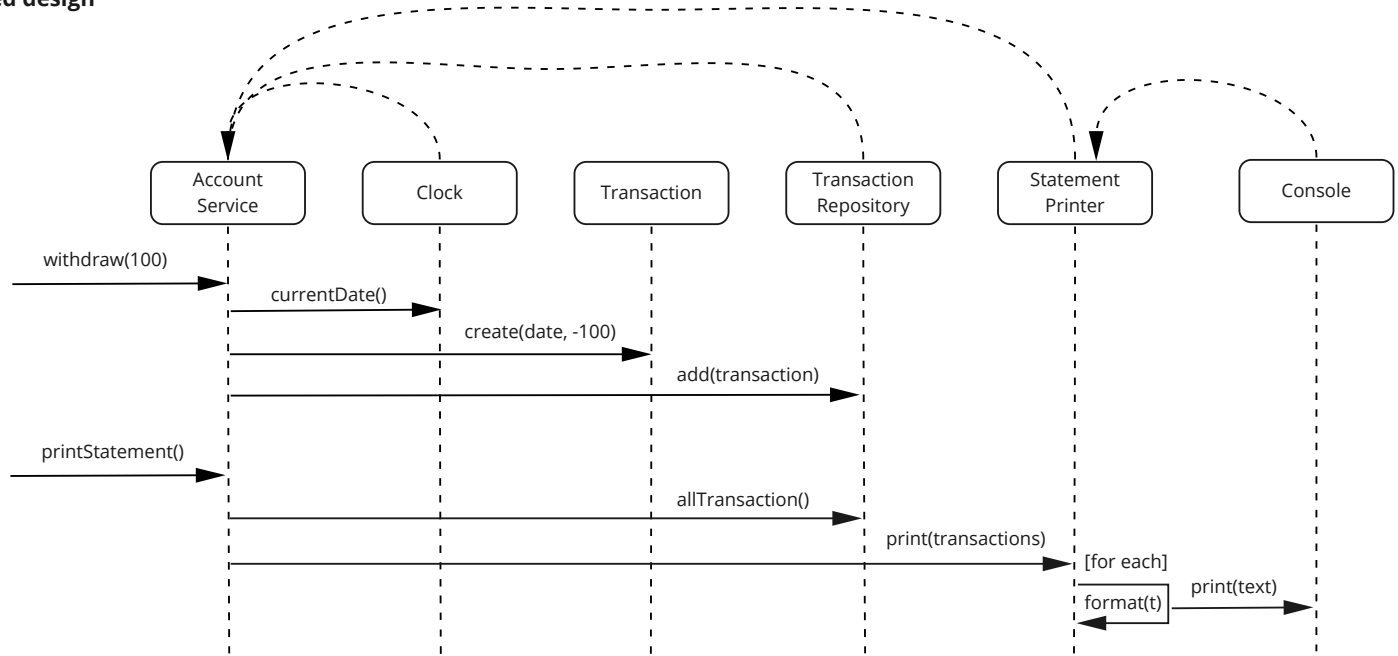
Fan-In

- If the class API changes, it causes a ripple effect upwards, which is normally harder to fix.
- Normally not an issue when the module is stable. Imagine all the Java classes like String, HashMap, etc.
    - Repositories and small utility classes should keep their APIs stable.
- A big fan-in might be an indication of a God class, in case is a class with a lot of business/domain logic.

Fan-Out

- A class in a high level of abstraction is controlling multiple classes in a much lower level of abstraction.
- This coordination of low level behaviour is normally a sign of anaemic domain, that means, there is a lot of behaviour hidden inside a class that would be better off if represented on its own class.
- Classes with big fan-out normally violate SRP and are very difficult to test.

Proposed design



Pros and cons:

- Code is still relatively simple and appropriate for the complexity of the problem.
- AccountService has 3 collaborators. I would consider split the AccountService in two classes, keeping the deposit and withdraw methods together and move the printStatement method to another class.
- Account Service has some duplication on the deposit and withdraw methods that could be removed with the TransactionFactory as described in the green approach.
- No design is perfect. It is all about trade-offs.

statement.feature

```
Feature: Printing Statement

  Scenario: Empty statement

        Given a new bank account with no transactions
        When I print a statement
        Then A statement with no transaction is printed

    Scenario: An account with deposits and withdrawals

        Given a bank account with a deposit of 100.00 on "01/04/2019"
        And a deposit of 200.00 on "04/04/2019"
        And a withdrawal of 50.00 on "05/04/2019"
        And a withdrawal of 80.00 on "10/04/2019"
        When I print a statement
        Then the statement should contain:
          """
          DATE | AMOUNT | BALANCE+
          01/04/2019 | 100.00 | 100.00+
          04/04/2019 | 200.00 | 300.00+
          05/04/2019 | -50.00 | 250.00+
          10/04/2019 | -80.00 | 170.00+
          """
```

StatementStepdefs.java

```
package com.codurance.craftingcode.exercise_05_bank_kata_with_junit.steps;

import com.codurance.craftingcode.exercise_05_bank_kata_with_junit.*;
import cucumber.api.Format;
import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;
import cucumber.api.java.en.When;

import java.text.SimpleDateFormat;
import java.util.Date;

import static
com.codurance.craftingcode.exercise_05_bank_kata_with_junit.StatementPrinter.STATEMENT_HEADER;
import static java.util.Arrays.asList;
import static org.mockito.BDDMockito.given;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;

public class StatementStepdefs {

 private static final SimpleDateFormat DD_MM_YYYY = new SimpleDateFormat("dd/MM/yyyy");

   @Given("^a new bank account with no transactions$")
   public void a_new_bank_account_with_no_transactions() {
      accountService = createNewAccount();
   }

   @Given("^a bank account with a deposit of (\\d+\\.\\d+) on \"(.+)\"$")
   public void a_bank_account_with_a_deposit_of_on(double amount, @Format("dd/MM/yyyy") Date date) {
      accountService = createNewAccount();
      deposit(amount, date);
   }

   @Given("^a deposit of (\\d+\\.\\d+) on \"(.+)\"$")
   public void a_deposit_of_on(double amount, @Format("dd/MM/yyyy") Date date) {
      deposit(amount, date);
   }

   @Given("^a withdrawal of (\\d+\\.\\d+) on \"(.+?)\"$")
   public void a_withdrawal_of_on(double amount, @Format("dd/MM/yyyy") Date date) {
      withdraw(amount, date);
   }

   @When("^I print a statement$")
   public void i_print_a_statement() {
      accountService.printStatement();
   }

   @Then("^A statement with no transaction is printed$")
   public void a_statement_with_no_transaction_is_printed() {
      verify(console).println(STATEMENT_HEADER);
   }

   @Then("^the statement should contain:$")
   public void the_statement_should_contain(String statement) {
      String[] statementLines = statement.replace("\n", "").split("\\+");
      asList(statementLines).forEach(line -> verify(console).println(line));
   }

   private AccountService createNewAccount() {
      TransactionRepository transactionRepository = new TransactionRepository();
      StatementPrinter statementPrinter = new StatementPrinter(console);
      return new AccountService(clock, transactionRepository, statementPrinter);
   }

   private void deposit(double amount, Date date) {
      given(clock.todayAsString()).willReturn(DD_MM_YYYY.format(date));
      accountService.deposit((int) amount);
   }

   private void withdraw(double amount, Date date) {
      given(clock.todayAsString()).willReturn(DD_MM_YYYY.format(date));
      accountService.withdraw((int) amount);
   }

   private Clock clock = mock(Clock.class);
   private Console console = mock(Console.class);
   private AccountService accountService;

}
```

StatementFeature.java

```
package feature;

import com.codurance.bankkata.*;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InOrder;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

import static org.mockito.BDDMockito.given;
import static org.mockito.Mockito.inOrder;

@RunWith(MockitoJUnitRunner.class)
public class PrintStatementFeature {

    @Mock Console console;
    @Mock Clock clock;

    private AccountService accountService;

    @Before
    public void initialise() {
        TransactionRepository transactionRepository = new TransactionRepository(clock);
        StatementPrinter statementPrinter = new StatementPrinter(console);
        accountService = new AccountService(transactionRepository, statementPrinter);
    }

    @Test public void
    print_statement_containing_all_transactions() {
        given(clock.todayAsString()).willReturn("01/04/2019", "02/04/2019", "10/04/2019");

        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("10/04/2019 | 500.00 | 1400.00");
        inOrder.verify(console).printLine("02/04/2019 | -100.00 | 900.00");
        inOrder.verify(console).printLine("01/04/2019 | 1000.00 | 1000.00");
    }
}
```

StatementPrinter.java

```
package com.codurance.bankkata;

import java.text.DecimalFormat;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.stream.Collectors;

public class StatementPrinter {

    public static final String STATEMENT_HEADER = "DATE | AMOUNT | BALANCE";

    private DecimalFormat decimalFormatter = new DecimalFormat("#.00");

    private Console console;

    public StatementPrinter(Console console) {
        this.console = console;
    }

    public void print(List<Transaction> transactions) {
        console.printLine(STATEMENT_HEADER);
        printStatementLines(transactions);
    }

    private void printStatementLines(List<Transaction> transactions) {
        AtomicInteger runningBalance = new AtomicInteger(0);
        transactions.stream()
                    .map(transaction -> statementLine(transaction, runningBalance))
                    .collect(Collectors.toCollection(LinkedList::new))
                    .descendingIterator()
                    .forEachRemaining(console::printLine);
    }

    private String statementLine(Transaction transaction, AtomicInteger runningBalance) {
         return transaction.date()
                 + " | "
                 + decimalFormatter.format(transaction.amount())
                 + " | "
                 + decimalFormatter.format(runningBalance.addAndGet(transaction.amount()));
    }
}
```

AccountServiceShould.java

```
package com.codurance.craftingcode.exercise_05_bank_kata_with_junit.unit;

import com.codurance.craftingcode.exercise_05_bank_kata_with_junit.*;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

import java.util.List;

import static java.util.Arrays.asList;
import static org.mockito.BDDMockito.given;
import static org.mockito.Mockito.verify;

@RunWith(MockitoJUnitRunner.class)
public class AccountServiceShould {

 private static final List<Transaction> ALL_TRANSACTIONS =
         asList(new Transaction("15/01/2019", 100));

   @Mock StatementPrinter statementPrinter;
   @Mock TransactionRepository transactionRepository;
   @Mock Clock clock;

   private AccountService accountService;

   @Before
   public void initialise() {
      accountService = new AccountService(clock, transactionRepository, statementPrinter);
   }

   @Test public void
   accept_a_deposit() {
      given(clock.todayAsString()).willReturn("01/09/2019");

      accountService.deposit(1000);

       verify(transactionRepository).add(transaction("01/09/2019", 1000));
   }

   @Test public void
   accept_a_withdrawal() {
      given(clock.todayAsString()).willReturn("02/09/2019");

      accountService.withdraw(500);

       verify(transactionRepository).add(transaction("02/09/2019", -500));
   }

   @Test public void
   print_a_statement_containing_all_transactions() {
      given(transactionRepository.all()).willReturn(ALL_TRANSACTIONS);

      accountService.printStatement();

      verify(statementPrinter).print(ALL_TRANSACTIONS);
   }

   private Transaction transaction(String date, int amount) {
      return new Transaction(date, amount);
   }

}
```

AccountService.java

```
package com.codurance.craftingcode.exercise_05_bank_kata_with_junit;

public class AccountService {

 private Clock clock;
   private TransactionRepository transactionRepository;
   private StatementPrinter statementPrinter;

   public AccountService(Clock clock, TransactionRepository transactionRepository,
StatementPrinter statementPrinter) {
      this.clock = clock;
      this.transactionRepository = transactionRepository;
      this.statementPrinter = statementPrinter;
   }

   public void printStatement() {
      statementPrinter.print(transactionRepository.all());
   }

   public void  deposit(int amount) {
      transactionRepository.add(transactionWith(amount));
   }

   public void withdraw(int amount) {
      transactionRepository.add(transactionWith(-amount));
   }

   private Transaction transactionWith(int amount) {
      return new Transaction(clock.todayAsString(), amount);
   }
}
```

StatementPrinterShould.java

```
package com.codurance.craftingcode.exercise_05_bank_kata_with_junit.unit;

import com.codurance.craftingcode.exercise_05_bank_kata_with_junit.Console;
import com.codurance.craftingcode.exercise_05_bank_kata_with_junit.StatementPrinter;
import com.codurance.craftingcode.exercise_05_bank_kata_with_junit.Transaction;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InOrder;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

import java.util.List;

import static java.util.Arrays.asList;
import static java.util.Collections.EMPTY_LIST;
import static org.mockito.Mockito.inOrder;
import static org.mockito.Mockito.verify;

@RunWith(MockitoJUnitRunner.class)
public class StatementPrinterShould {

 private static final List<Transaction> NO_TRANSACTIONS = EMPTY_LIST;

   @Mock Console console;

   private StatementPrinter statementPrinter;

   @Before
   public void initialise() {
      statementPrinter = new StatementPrinter(console);
   }

   @Test public void
   always_print_the_header() {
      statementPrinter.print(NO_TRANSACTIONS);

      verify(console).printLine("DATE | AMOUNT | BALANCE");
   }

   @Test public void
   print_transactions_in_reverse_chronological_order() {
      statementPrinter.print(transactionsContaining(
                            deposit(   "01/04/2019", 1000),
                            withdrawal("02/04/2019", 100),
                            deposit(   "10/04/2019", 500)));

      InOrder inOrder = inOrder(console);
      inOrder.verify(console).printLine("DATE | AMOUNT | BALANCE");
      inOrder.verify(console).printLine("10/04/2019 | 500.00 | 1400.00");
      inOrder.verify(console).printLine("02/04/2019 | -100.00 | 900.00");
      inOrder.verify(console).printLine("01/04/2019 | 1000.00 | 1000.00");
   }

   private List<Transaction> transactionsContaining(Transaction... transactions) {
      return asList(transactions);
   }

   private Transaction withdrawal(String date, int amount) {
      return new Transaction(date, -amount);
   }

   private Transaction deposit(String date, int amount) {
      return new Transaction(date, amount);
   }

}
```

TransactionRepository.java

```
package com.codurance.craftingcode.exercise_05_bank_kata_with_junit;

import java.util.ArrayList;
import java.util.List;

import static java.util.Collections.unmodifiableList;

public class TransactionRepository {
 private List<Transaction> transactions = new ArrayList<>();

   public void add(Transaction transaction) {
      transactions.add(transaction);
   }

   public List<Transaction> all() {
      return unmodifiableList(transactions);
   }
}
```

TransactionRepositoryShould.java

```
package com.codurance.craftingcode.exercise_05_bank_kata_with_junit.unit;

import com.codurance.craftingcode.exercise_05_bank_kata_with_junit.Transaction;
import com.codurance.craftingcode.exercise_05_bank_kata_with_junit.TransactionRepository;
import org.junit.Before;
import org.junit.Test;

import java.util.List;

import static org.assertj.core.api.Assertions.assertThat;

public class TransactionRepositoryShould {

 public static final String TODAY = "12/05/2019";
    private static final String YESTERDAY = "11/05/2019";;
    private static final Transaction DEPOSIT_TRANSACTION = new Transaction(TODAY, 100);
    private static final Transaction WITHDRAWAL_TRANSACTION = new Transaction(YESTERDAY, -50);

    private TransactionRepository transactionRepository;

    @Before
    public void initialise() {
        transactionRepository = new TransactionRepository();
    }

    @Test public void
    store_transactions() {
        transactionRepository.add(DEPOSIT_TRANSACTION);
        transactionRepository.add(WITHDRAWAL_TRANSACTION);

        List<Transaction> transactions = transactionRepository.all();

        assertThat(transactions).containsExactlyInAnyOrder(DEPOSIT_TRANSACTION,
WITHDRAWAL_TRANSACTION);
    }
}
```

TripService.java

```
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();
    }
}
```

Testing and refactoring legacy code

Please close the following repository:

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

Note: If you don't have git, download as a zip file.

Import the folder related to your programming language (not the root folder) into your IDE.

- You may need to fix some configurations depending on the version of the language you are using.
- You may want to change the versions of the testing and mocking frameworks.
- If using Java, import the project as a Maven project.

Open a class called TripService[.java|.cs|.etc]

- Make sure there are no errors in your IDE.

TripService - Business Rules

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
- If you are not logged in, the code throws an exception.

Exercise Rules

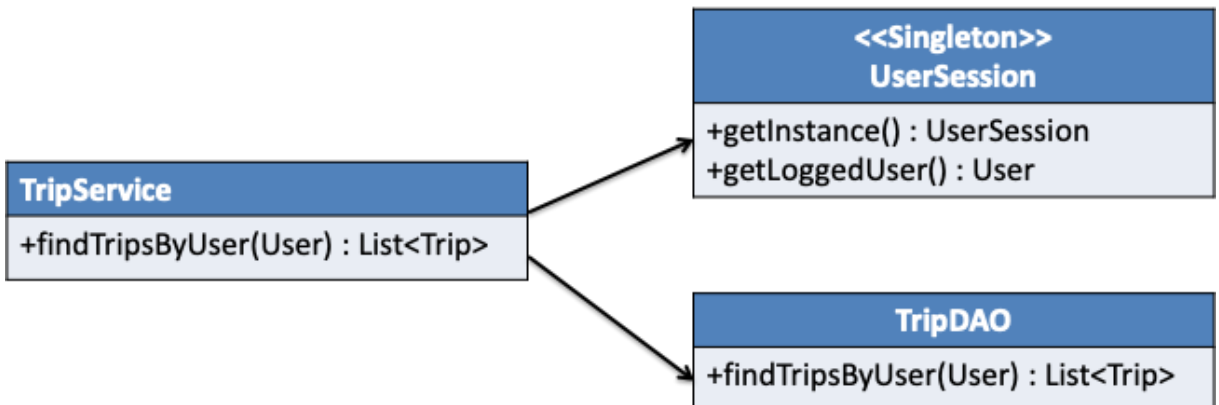
- Your job is to write tests for the TripService class until you have 100% test coverage.
- Once you have 100% test coverage, you need to refactor and make the code better.
- At the end of the refactoring, both tests and production code should clearly describe the business rules.

Exercise Constraints

- You cannot manually change production code if not covered by tests, that means:
    - You cannot type of the TripService class while still not covered by tests.
- If you need to change the TripService class in order to test, you can do so using automated refactorings (via IDE).
- You CANNOT change the public interface of TripService, that means:
    - You cannot change its constructor
    - You cannot change the method signature
    - Both changes above might cause other classes to change, which is not desirable now.
- You CANNOT introduce state in the TripService:
    - TripService is stateless. Introducing state may cause multi-thread issues.

Working with legacy code tips:

Problems to resolve



Unit tests for TripService must not call the real UserSession and TripDAO. It should solely focus on the TripService. The real classes have dependencies on resources (HTTP session, database) that are not available at unit test level. Unit tests will break if involving the real collaborators.

TripServiceShould.java

```
package org.craftedsw.tripservicekata.trip;

import org.craftedsw.tripservicekata.exception.UserNotLoggedInException;
import org.craftedsw.tripservicekata.user.User;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

import java.util.List;

import static org.assertj.core.api.Assertions.assertThat;
import static org.craftedsw.tripservicekata.trip.UserBuilder.aUser;
import static org.mockito.BDDMockito.given;

@RunWith(MockitoJUnitRunner.class)
public class TripServiceShould {

    private static final User GUEST = null;
    private static final User USER = new User();
    private static final User REGISTERED_USER = new User();
    private static final User ANOTHER_USER = new User();
    private static final Trip IASI = new Trip();
    private static final Trip LONDON = new Trip();

    @Mock TripDAO tripDAO;

    private TripService tripService;

    @Before
    public void initialise() {
        tripService = new TripService(tripDAO);
    }

    @Test(expected = UserNotLoggedInException.class) public void
    validate_the_logged_in_user() {
        tripService.getTripsByUser(USER, GUEST);
    }

    @Test public void
    return_trip_when_users_are_friends() {
        User friend = aUser()
                .friendsWith(ANOTHER_USER, REGISTERED_USER)
                .withTripsTo(IASI, LONDON)
                .build();
        given(tripDAO.tripsBy(friend)).willReturn(friend.trips());

        List<Trip> trips = tripService.getTripsByUser(friend, REGISTERED_USER);

        assertThat(trips).containsExactlyInAnyOrder(IASI, LONDON);
    }

    @Test public void
    return_no_trips_when_users_are_not_friends() {
        User stranger = aUser()
                        .friendsWith(ANOTHER_USER)
                        .withTripsTo(IASI)
                        .build();

        List<Trip> trips = tripService.getTripsByUser(stranger, REGISTERED_USER);

        assertThat(trips).isEmpty();
    }

}
```

TripService.java

```
package org.craftedsw.tripservicekata.trip;

import org.craftedsw.tripservicekata.exception.UserNotLoggedInException;
import org.craftedsw.tripservicekata.user.User;

import java.util.ArrayList;
import java.util.List;

public class TripService {

    private TripDAO tripDAO;

    public TripService() {
        tripDAO = new TripDAO();
    }

    public TripService(TripDAO tripDAO) {
        this.tripDAO = tripDAO;
    }

    public List<Trip> getTripsByUser(User user, User loggedInUser) throws UserNotLoggedInException {
        validate(loggedInUser);

        return user.isFriendsWith(loggedInUser)
                        ? tripsBy(user)
                        : noTrips();
    }

    private ArrayList<Trip> noTrips() {
        return new ArrayList<Trip>();
    }

    private void validate(User loggedInUser) {
        if (loggedInUser == null) {
            throw new UserNotLoggedInException();
        }
    }

    private List<Trip> tripsBy(User user) {
        return tripDAO.tripsBy(user);
    }

}
```

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?

Other resources

Clean Coders - London vs Chicago




https://cleancoders.com/series/comparativeDesign

Codurance

Services: https://codurance.com/services/
Publications: https://codurance.com/publications/
YouTube: https://www.youtube.com/codurance
Twitter: https://twitter.com/codurance
Podcasts: https://codurancetalks.podbean.com/
Katalyst: https://katalyst.codurance.com/

Books

Personal contacts

email: sandro@codurance.com
Twitter: https://twitter.com/sandrodmancuso