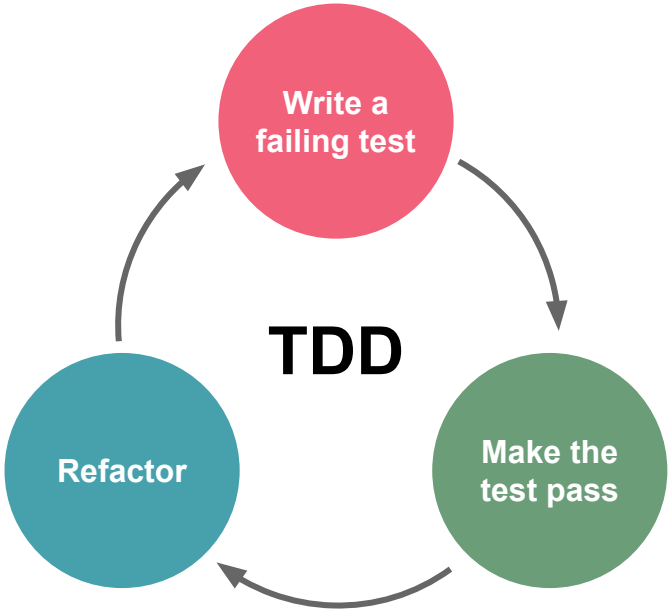
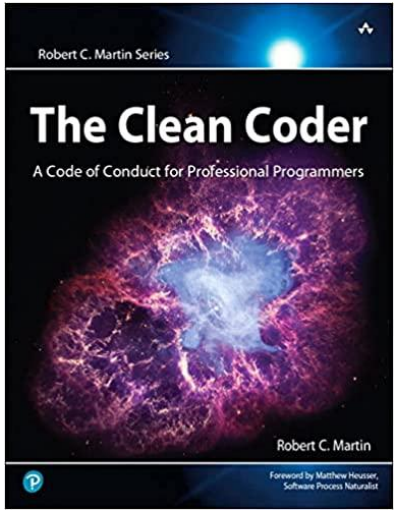
I’ve done TDD wrong all the time

/thoughtworks

Test driven development



The benefit that was promised



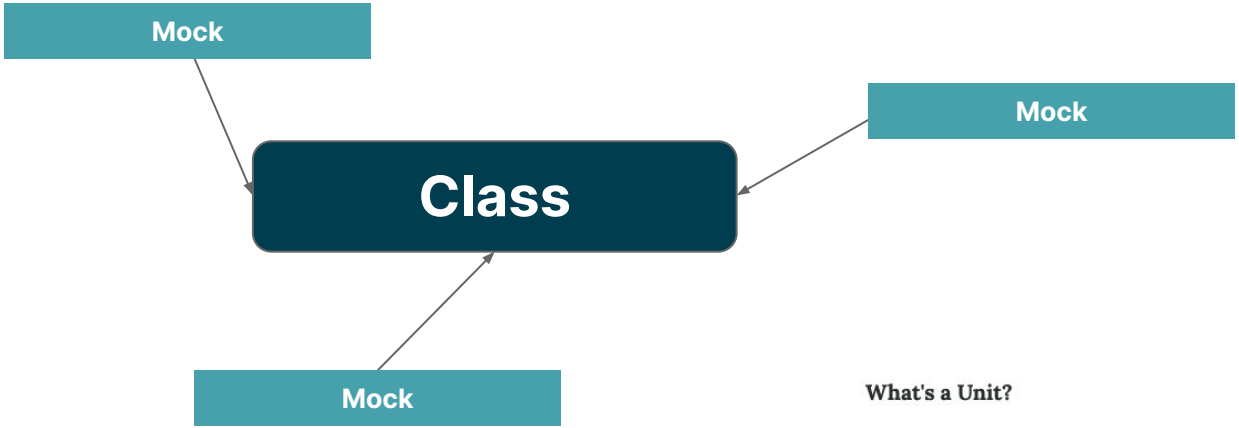
Remove the fear of refactoring

VS

Unit test

Mock

What are Unit tests and Mocks



What's a Unit?

If you're working in a functional language a *unit* will most likely be a single function. Your unit tests will call a function with different parameters and ensure that it returns the expected values. In an object-oriented language a unit can range from a single method to an entire class.

The Practical Test Pyramid
martinfowler.com

reference

Example

/thoughtworks

Bookstore

Has been asked to implement a ethical bookstore that that doesn't accept money

It has two operation:

- Withdraw a book
- Deposit a book

Withdraw a book it cost 2 credit points.
If we have at least 5 credit we get a **discount** of 1 credit point.

Deposit a book gives 1 credit point.
If we have 10 or more credit points he will receive 1 credit point as **extra bonus**.

TDD - 1° cycle

```java
@AllArgsConstructor
public class Account {

    @Getter(AccessLevel.PROTECTED)
    private int points;

    public void deposit() {

    }
}
```

```java
class AccountTest {

    @Test
    void shouldIncreasePointsByDepositingABook() {
        Account account = new Account(0);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(1);
    }
}
```

TDD - 1° cycle

```java
@AllArgsConstructor
public class Account {

    @Getter(AccessLevel.PROTECTED)
    private int points;

    public void deposit() {
        points++;
    }
}
```

```java
class AccountTest {

    @Test
    void shouldIncreasePointsByDepositingABook() {
        Account account = new Account(0);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(1);
    }
}
```

TDD - 2° cycle

```java
@AllArgsConstructor
public class Account {

    @Getter(AccessLevel.PROTECTED)
    private int points;

    public void deposit() {
        points++;
    }
}
```

```java
class AccountTest {

    @Test
    void shouldIncreasePointsByDepositingABook() {
        Account account = new Account(0);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(1);
    }

    @Test
    void shouldGetBonusPointForLoyalCustomers() {
        Account account = new Account(10);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(12);
    }
}
```

TDD - 2° cycle

```java
@AllArgsConstructor
public class Account {

    @Getter(AccessLevel.PROTECTED)
    private int points;

    public void deposit() {
        if (points >= 10) {
            points += 2;
        } else {
            points++;
        }
    }
}
```

```java
class AccountTest {

    @Test
    void shouldIncreasePointsByDepositingABook() {
        Account account = new Account(0);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(1);
    }

    @Test
    void shouldGetBonusPointForLoyalCustomers() {
        Account account = new Account(10);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(12);
    }
}
```


TDD - 2° cycle

```java
@AllArgsConstructor
public class Account {
    public static final int BONUS_THRESHOLD = 10;
    public static final int BONUS = 1;

    @Getter(AccessLevel.PROTECTED)
    private int points;

    public void deposit() {
        if (points >= BONUS_THRESHOLD) {
            points += BONUS;
        }
        points++;
    }
}
```

```java
class AccountTest {

    @Test
    void shouldIncreasePointsByDepositingABook() {
        Account account = new Account(0);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(1);
    }

    @Test
    void shouldGetBonusPointForLoyalCustomers() {
        Account account = new Account(10);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(12);
    }
}
```

TDD - 5° cycle

```java
@AllArgsConstructor
public class Account {
    public static final int BONUS_THRESHOLD = 10;
    public static final int BONUS = 1;
    public static final int POINT_FOR_DEPOSIT = 1;
    public static final int DISCOUNT_THRESHOLD = 5;
    public static final int DISCOUNT = 1;
    public static final int POINT_FOR_WITHDRAW = 2;

    @Getter(AccessLevel.PROTECTED)
    private int points;

    public void deposit() {
        if (points >= BONUS_THRESHOLD) {
            points += BONUS;
        }
        points += POINT_FOR_DEPOSIT;
    }

    public void withdraw() throws WithdrawException {
        if (points < POINT_FOR_WITHDRAW) {
            throw new WithdrawException();
        }
        if (points >= DISCOUNT_THRESHOLD) {
            points += DISCOUNT;
        }

        points -= POINT_FOR_WITHDRAW;
    }
}
```

```java
class AccountTest {

    @Test
    void shouldIncreasePointsByDepositingABook() {
        Account account = new Account(0);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(1);
    }

    @Test
    void shouldGetBonusPointForLoyalCustomers() {
        Account account = new Account(10);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(12);
    }

    @Test
    void shouldWithdrawABook() throws WithdrawException {
        Account account = new Account(3);
        account.withdraw();

        assertThat(account.getPoints()).isEqualTo(1);
    }

    @Test
    void shouldWithdrawABookWithDiscount() throws WithdrawException {
        Account account = new Account(5);
        account.withdraw();

        assertThat(account.getPoints()).isEqualTo(4);
    }

    @Test
    void shouldAvoidToWithdrawWhenThereAreNotEnoughPoints() {
        Account account = new Account(1);

        ThrowableAssert throwableAssert = assertThatThrownBy(
          () -> account.withdraw()
        );
        throwableAssert.isInstanceOf(WithdrawException.class);

        assertThat(account.getPoints()).isEqualTo(1);
    }
}
```

TDD - 5° cycle

```java
@AllArgsConstructor
public class Account {
    public static final int BONUS_THRESHOLD = 10;
    public static final int BONUS = 1;
    public static final int POINT_FOR_DEPOSIT = 1;
    public static final int DISCOUNT_THRESHOLD = 5;
    public static final int DISCOUNT = 1;
    public static final int POINT_FOR_WITHDRAW = 2;

    @Getter(AccessLevel.PROTECTED)
    private int points;

    public void deposit() {
        if (points >= BONUS_THRESHOLD) {
            points += BONUS;
        }
        points += POINT_FOR_DEPOSIT;
    }

    public void withdraw() throws WithdrawException {
        if (points < POINT_FOR_WITHDRAW) {
            throw new WithdrawException();
        }
        if (points >= DISCOUNT_THRESHOLD) {
            points += DISCOUNT;
        }

        points -= POINT_FOR_WITHDRAW;
    }
}
```

Single Class Application

Design Patterns

Book store - State pattern

Contributor State

```java
public class ContributorState extends State {

    public ContributorState(Account account) {
        super(account);
    }

    @Override
    public void deposit() {
        this.account.setPoints(
                this.account.getPoints()
                        + POINT_FOR_DEPOSIT
        );
    }

    @Override
    public void withdraw() throws WithdrawException {
        throw new WithdrawException();
    }

}
```

```java
class ContributorStateTest {

    ContributorState state;
    Account account = mock(Account.class);

    @BeforeEach
    void setUp() {
        state = new ContributorState(account);
    }

    @Test
    void shouldDeposit() {
        when(account.getPoints()).thenReturn(1);

        state.deposit();

        verify(account).setPoints(2);
    }

    @Test
    void shouldWithdraw() {
        when(account.getPoints()).thenReturn(1);

        ThrowableAssert throwableAssert =
                assertThatThrownBy(
                        () -> state.withdraw()
                );
        throwableAssert
                .isInstanceOf(WithdrawException.class);
    }
}
```

Community State

```java
public class CommunityState extends State {

    public CommunityState(Account account) {
        super(account);
    }

    @Override
    public void deposit() {
        this.account.setPoints(
                this.account.getPoints()
                        + POINT_FOR_DEPOSIT
        );
    }

    @Override
    public void withdraw() throws WithdrawException {
        this.account.setPoints(
                this.account.getPoints()
                        - POINT_FOR_WITHDRAW
        );
    }

}
```

```java
class CommunityStateTest {

    CommunityState state;
    Account account = mock(Account.class);

    @BeforeEach
    void setUp() {
        state = new CommunityState(account);
    }

    @Test
    void shouldDeposit() {
        when(account.getPoints()).thenReturn(2);

        state.deposit();

        verify(account).setPoints(3);
    }

    @Test
    void shouldWithdraw() throws WithdrawException {
        when(account.getPoints()).thenReturn(4);

        state.withdraw();

        verify(account).setPoints(2);
    }
}
```

Affiliate State

```java
public class AffiliateState extends State {
    public AffiliateState(Account account) {
        super(account);
    }

    @Override
    public void deposit() {
        this.account.setPoints(
                this.account.getPoints()
                        + POINT_FOR_DEPOSIT
        );
    }

    @Override
    public void withdraw() throws WithdrawException {
        this.account.setPoints(
                this.account.getPoints()
                        - POINT_FOR_WITHDRAW
                        + DISCOUNT
        );
    }

}
```

```java
class AffiliateStateTest {

    AffiliateState state;
    Account account = mock(Account.class);

    @BeforeEach
    void setUp() {
        state = new AffiliateState(account);
    }

    @Test
    void shouldDeposit() {
        when(account.getPoints()).thenReturn(5);

        state.deposit();

        verify(account).setPoints(6);
    }

    @Test
    void shouldWithdraw() throws WithdrawException {
        when(account.getPoints()).thenReturn(9);

        state.withdraw();

        verify(account).setPoints(8);
    }
}
```

Premium State

```java
public class PremiumState extends State {
    public PremiumState(Account account) {
        super(account);
    }

    @Override
    public void deposit() {
        this.account.setPoints(
                this.account.getPoints()
                        + POINT_FOR_DEPOSIT
                        + BONUS
        );
    }

    @Override
    public void withdraw() throws WithdrawException {
        this.account.setPoints(
                this.account.getPoints()
                        - POINT_FOR_WITHDRAW
                        + DISCOUNT
        );
    }

}
```

```java
class PremiumStateTest {

    PremiumState state;
    Account account = mock(Account.class);

    @BeforeEach
    void setUp() {
        state = new PremiumState(account);
    }

    @Test
    void shouldDeposit() {
        when(account.getPoints()).thenReturn(10);

        state.deposit();

        verify(account).setPoints(12);
    }

    @Test
    void shouldWithdraw() throws WithdrawException {
        when(account.getPoints()).thenReturn(15);

        state.withdraw();

        verify(account).setPoints(14);
    }
}
```

Account

```java
@AllArgsConstructor
public class Account {
    public static final int BONUS_THRESHOLD = 10;
    public static final int BONUS = 1;
    public static final int POINT_FOR_DEPOSIT = 1;
    public static final int DISCOUNT_THRESHOLD = 5;
    public static final int DISCOUNT = 1;
    public static final int POINT_FOR_WITHDRAW = 2;

    @Getter(AccessLevel.PROTECTED)
    private int points;

    public void deposit() {
        if (points >= BONUS_THRESHOLD) {
            points += BONUS;
        }
        points += POINT_FOR_DEPOSIT;
    }

    public void withdraw() throws WithdrawException {
        if (points < POINT_FOR_WITHDRAW) {
            throw new WithdrawException();
        }
        if (points >= DISCOUNT_THRESHOLD) {
            points += DISCOUNT;
        }

        points -= POINT_FOR_WITHDRAW;
    }
}
```

```java
class AccountTest {

    @Test
    void shouldIncreasePointsByDepositingABook() {
        Account account = new Account(0);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(1);
    }

    @Test
    void shouldGetBonusPointForLoyalCustomers() {
        Account account = new Account(10);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(12);
    }

    @Test
    void shouldWithdrawABook() throws WithdrawException {
        Account account = new Account(3);
        account.withdraw();

        assertThat(account.getPoints()).isEqualTo(1);
    }

    @Test
    void shouldWithdrawABookWithDiscount() throws WithdrawException {
        Account account = new Account(5);
        account.withdraw();

        assertThat(account.getPoints()).isEqualTo(4);
    }

    @Test
    void shouldAvoidToWithdrawWhenThereAreNotEnoughPoints() {
        Account account = new Account(1);

        ThrowableAssert throwableAssert = assertThatThrownBy(
          () -> account.withdraw()
        );
        throwableAssert.isInstanceOf(WithdrawException.class);

        assertThat(account.getPoints()).isEqualTo(1);
    }
}
```

Account

```java
@AllArgsConstructor
public class Account {
    public static final int BONUS_THRESHOLD = 10;
    public static final int BONUS = 1;
    public static final int POINT_FOR_DEPOSIT = 1;
    public static final int DISCOUNT_THRESHOLD = 5;
    public static final int DISCOUNT = 1;
    public static final int POINT_FOR_WITHDRAW = 2;

    @Getter(AccessLevel.PROTECTED)
    private int points;

    public void deposit() {
        if (points >= BONUS_THRESHOLD) {
            points += BONUS;
        }
        points += POINT_FOR_DEPOSIT;
    }

    public void withdraw() throws WithdrawException {
        if (points < POINT_FOR_WITHDRAW) {
            throw new WithdrawException();
        }
        if (points >= DISCOUNT_THRESHOLD) {
            points += DISCOUNT;
        }

        points -= POINT_FOR_WITHDRAW;
    }
}
```

```java
class AccountTest {
    private final State state = mock(State.class);
    private Account account;

    @BeforeEach
    void setUp() {
        account = new Account(0);
        account.setState(state);
    }

    @Test
    void shouldCallDeposit() {
        account.deposit();

        verify(state).deposit();
    }

    @Test
    void shouldCallWithdraw() throws WithdrawException {
        account.withdraw();

        verify(state).withdraw();
    }

}
```

Account

```java
public class Account {

    @Getter(AccessLevel.PROTECTED)
    @Setter(AccessLevel.PROTECTED)
    private int points;

    @Setter(AccessLevel.PROTECTED)
    private State state;

    public Account(int points) {
        this.points = points;
    }

    public void deposit() {
        state.deposit();
    }

    public void withdraw() throws WithdrawException {
        state.withdraw();
    }
}
```

```java
class AccountTest {
    private final State state = mock(State.class);
    private Account account;

    @BeforeEach
    void setUp() {
        account = new Account(0);
        account.setState(state);
    }

    @Test
    void shouldCallDeposit() {
        account.deposit();

        verify(state).deposit();
    }

    @Test
    void shouldCallWithdraw() throws WithdrawException {
        account.withdraw();

        verify(state).withdraw();
    }

}
```

What we have now

- The tests **don’t speak about “What”** the application does but **they are speaking about “How”**
- Tests are **far to be a design tool**
- **Fragile tests**
- For a few moment **we gave up our safety-net**

I’ve made a mistake

Account

```java
public class Account {

    @Getter(AccessLevel.PROTECTED)
    @Setter(AccessLevel.PROTECTED)
    private int points;

    @Setter(AccessLevel.PROTECTED)
    private State state = new ContributorState(this);

    public Account(int points) {
        this.points = points;
        this.state = state.getCurrentState();
    }

    public void deposit() {
        state.deposit();
        state.stateChangeCheck();
    }

    public void withdraw() throws WithdrawException {
        state.withdraw();
        state.stateChangeCheck();
    }
}
```

State

```java
public abstract class State {
    public static final int BONUS_THRESHOLD = 10;
    public static final int BONUS = 1;
    public static final int POINT_FOR_DEPOSIT = 1;
    public static final int DISCOUNT_THRESHOLD = 5;
    public static final int DISCOUNT = 1;
    public static final int POINT_FOR_WITHDRAW = 2;

    protected Account account;

    public abstract void deposit();

    public abstract void withdraw() throws
WithdrawException;

    public void stateChangeCheck() {
        this.account.setState(getCurrentState());
    }

    public State getCurrentState() {
        int balance = this.account.getPoints();
        if (balance < POINT_FOR_WITHDRAW) {
            return new ContributorState(account);
        } else if (balance < DISCOUNT_THRESHOLD) {
            return new CommunityState(account);
        } else if (balance < BONUS_THRESHOLD) {
            return new AffiliateState(account);
        } else {
            return new PremiumState(account);
        }
    }

}
```

Is TDD really helpful?

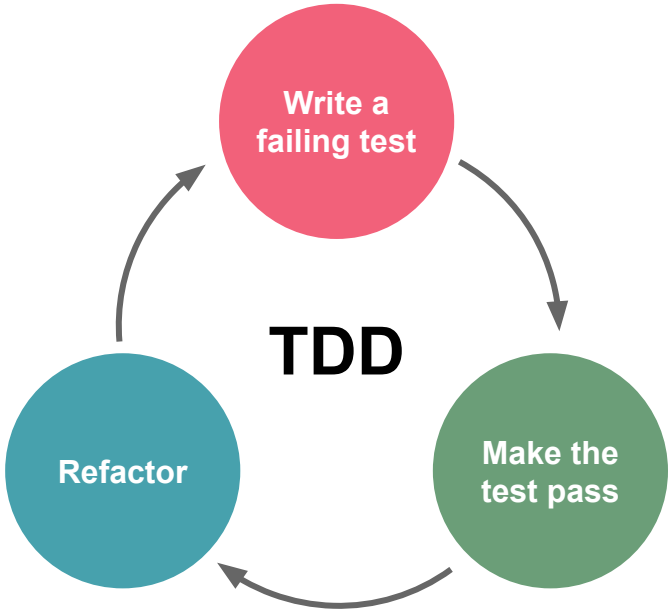
… or are we missing something?

/thoughtworks

What happened

We **changed** our code for some **implementation details**

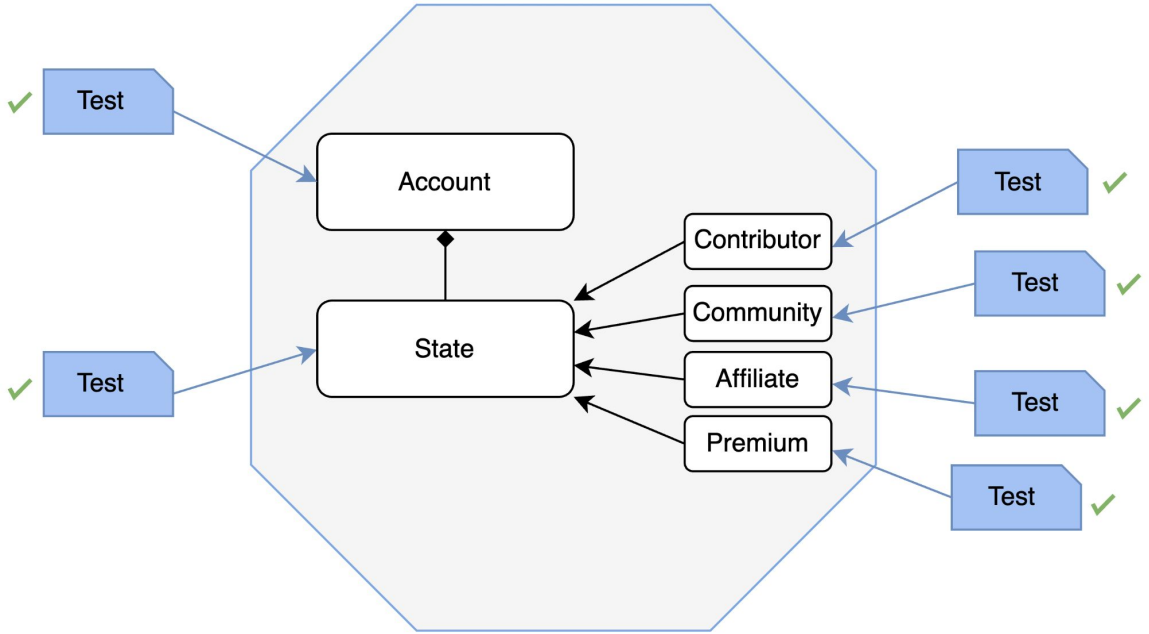
Tests/mocks became an **obstacle to the refactoring**



What do we do when we write a test?

We are defining a **contract**

that the code will fulfill

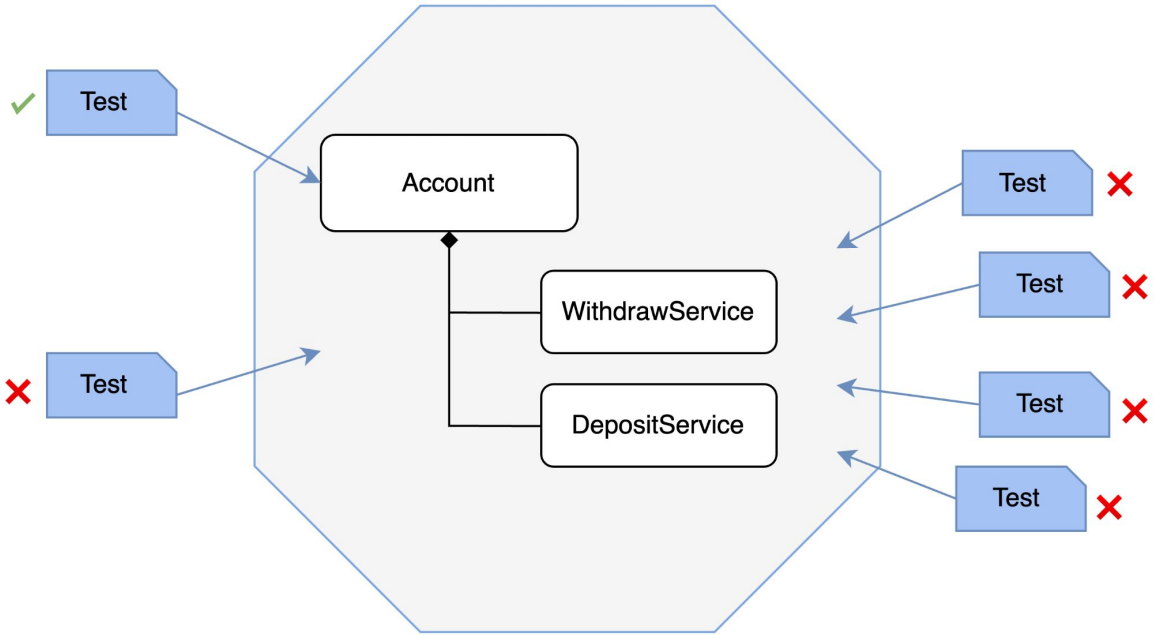


What do we do when we write a test?

We are defining a **contract**

that the code will fulfill

If it's **tight to implementation** details it will **easily breaks**



What is a Unit test?

Unit test Behaviour

Not methods or classes

/thoughtworks

Tests should be
coupled to the behavior of code and
decoupled from the structure of code

Kent Beck

/thoughtworks

TDD is not really about test but more about behavior and requirement

TDD // BDD // ATDD // Module test

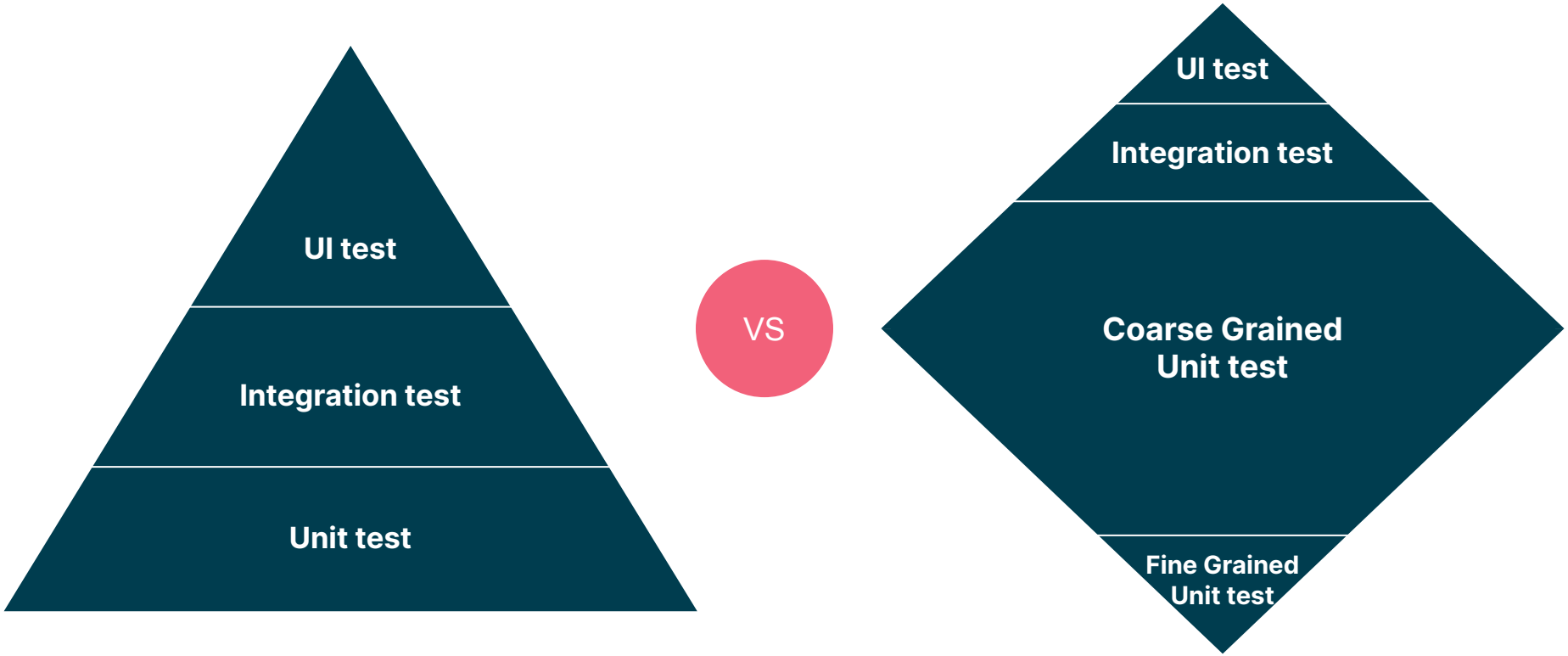
/thoughtworks

Lots of confusion has been made around the name Unit test

Fine grained unit test
Tests are coupled to the implementation

Coarse grained unit test
Tests are coupled to the behaviours

The Test Diamond



Combinatory logics

Give a free book if: **A and (B or C)**
i.e.

A = Deposit operation
B = Is user's birthday
C = User has visited the shop 3 times in a week

⚠️ trade-off

Write all possible scenario

All possible cases from a requirement point of view are:

- 2^3 = 9 **(2^n)**

Write tests for the logic condition

All possible cases from the implementation point of view:

- 3 + 1 = 4 **(n+1)**

TDD - Refactoring

If the **requirement doesn’t change**

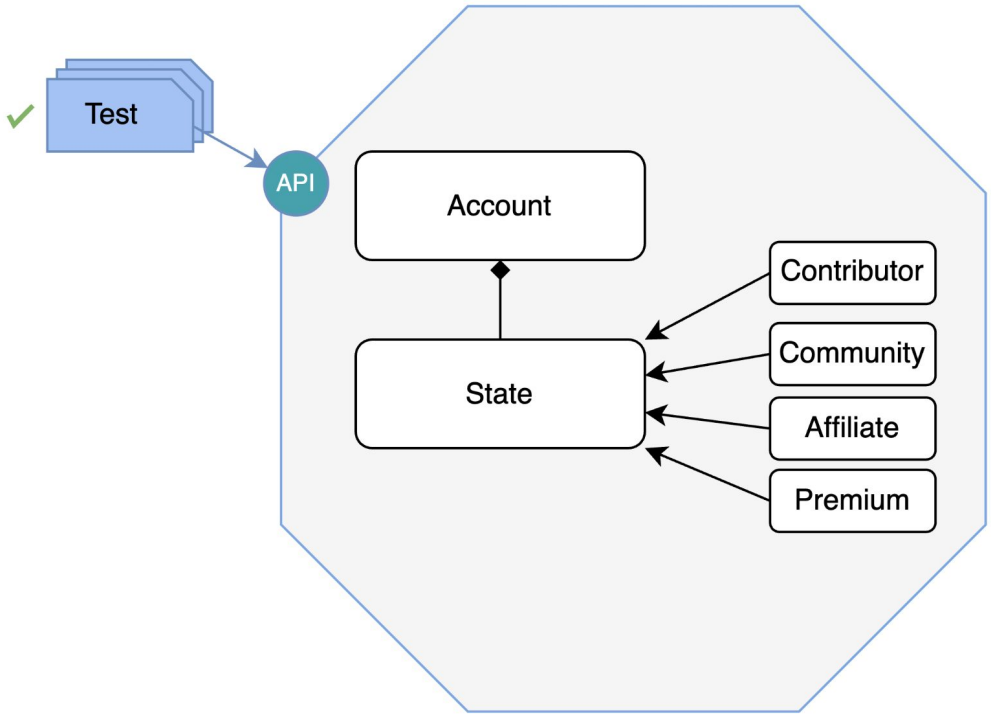
The **test shouldn't change** either

```
class AccountTest {

    @Test
    void shouldIncreasePointsByDepositingABook() {
        Account account = new Account(0);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(1);
    }

    @Test
    void shouldGetBonusPointForLoyalCustomers() {
        Account account = new Account(10);
        account.deposit();

        assertThat(account.getPoints()).isEqualTo(12);
    }

    @Test
    void shouldWithdrawABook() throws WithdrawException {
        Account account = new Account(3);
        account.withdraw();

        assertThat(account.getPoints()).isEqualTo(1);
    }

    @Test
    void shouldWithdrawABookWithDiscount() throws WithdrawException {
        Account account = new Account(5);
        account.withdraw();

        assertThat(account.getPoints()).isEqualTo(4);
    }

    @Test
    void shouldAvoidToWithdrawWhenThereAreNotEnoughPoints() {
        Account account = new Account(1);

        ThrowableAssert throwableAssert = assertThatThrownBy(
          () -> account.withdraw()
        );
        throwableAssert.isInstanceOf(WithdrawException.class);

        assertThat(account.getPoints()).isEqualTo(1);
    }
}
```

What should we do when we write a test?

Put yourself in the user's shoes, tests should be **executable specification**

Develop against an **interface**
Outside-in



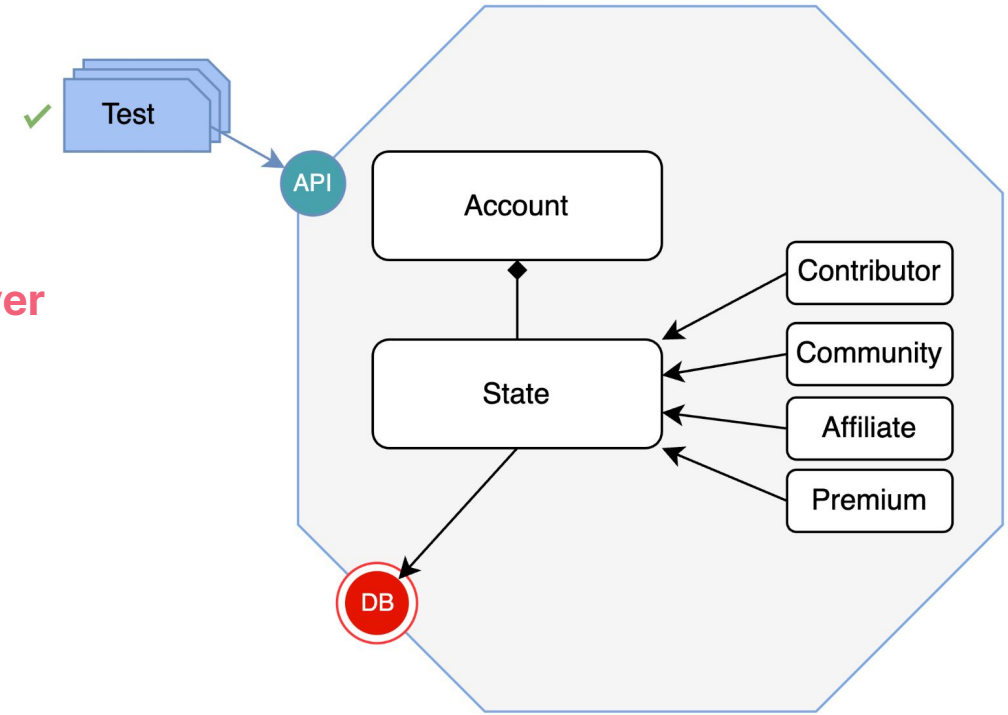
Are Mocks useless then?

Mock

Mocks are useful for external dependency

We should mock the **infrastructure layer** (through an adapter)

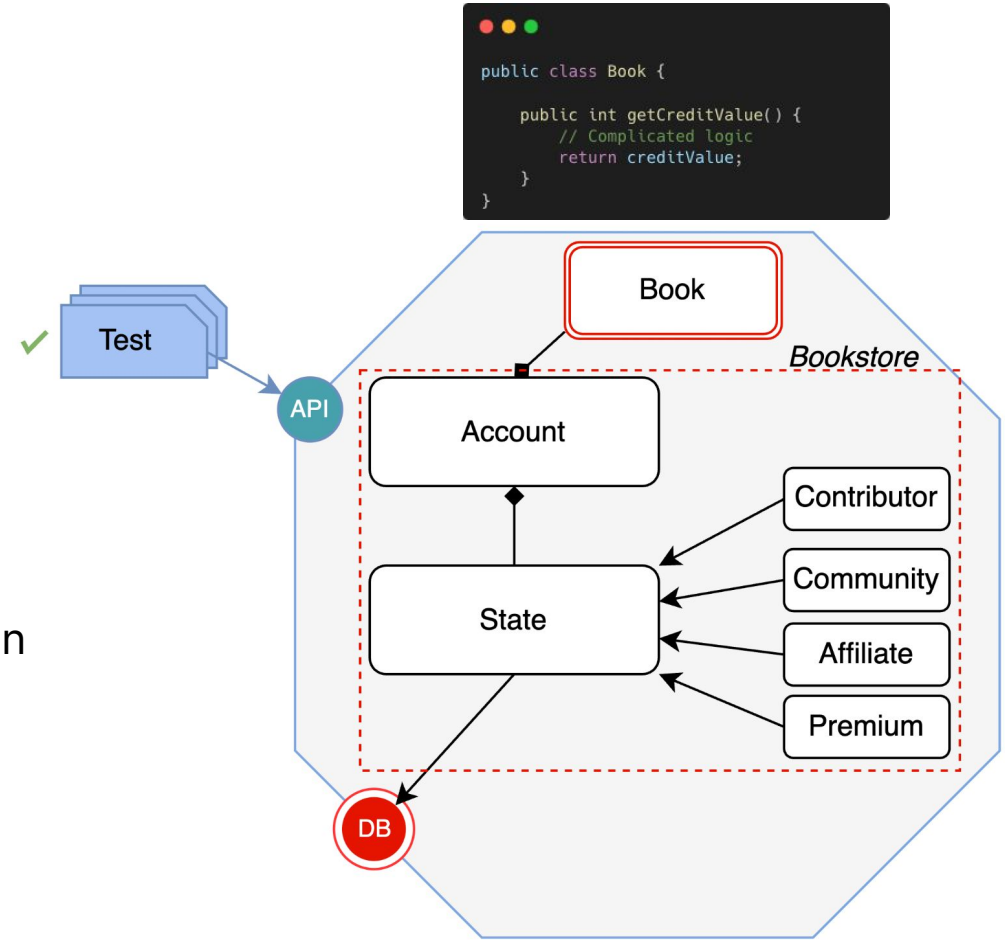
i.e. DB, Kafka, HTTP call



Mock

Mocks are useful for isolate pieces of the application that have **complex logics**

or isolate subdomains of our application



⚠️ trade-off

I’ve found my balance

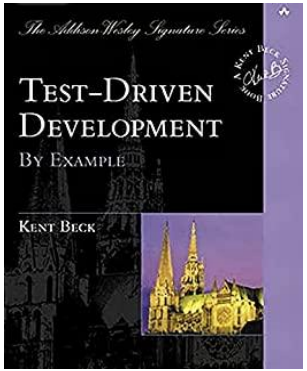
That’s how I’ve removed the
fear of refactoring

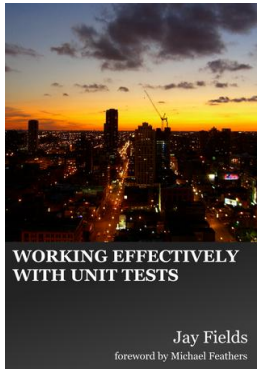
- Unit tests based on **behaviours**
- Test against an **interface**
- **Do not mock logic**, mock the infrastructure

Your balance might be different 🤸

References

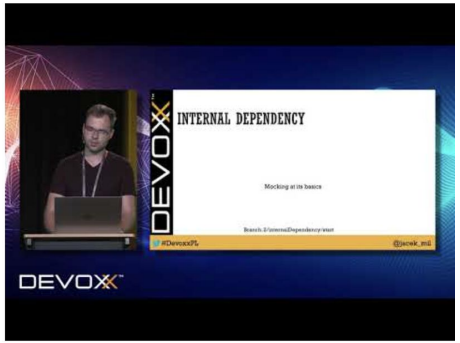
Books:



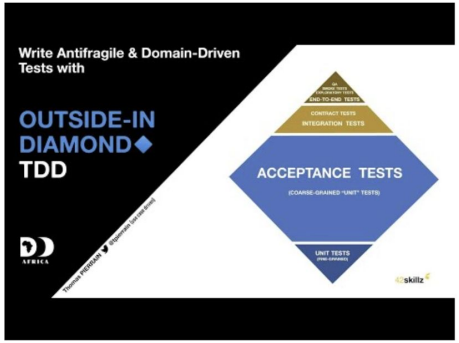


Videos:

TDD, Where Did It All Go Wrong
Ian Cooper



Automated tests: You won't find it in a book!
Jacek Milewski



Outside-in diamond ◆ TDD
Thomas Pierrain

TDD and Clean Architecture - Driven by Behaviour
Valentina Cupać

Thank you

Luca Giuberti
Senior Developer Consultant

lucagiuberti
@gix_lg

/thoughtworks

Questions?