Developer Testing
Part 2: Advanced Topics

- Isolated Testing
- Contract Testing
- Property-Based Testing
- Acceptance Testing / Specification by Example

Microtesting in a world of dependencies

Microtests are supposed to check a component in isolation, but

- Components do not work in isolation
- Components collaborate with other components to fulfil their task

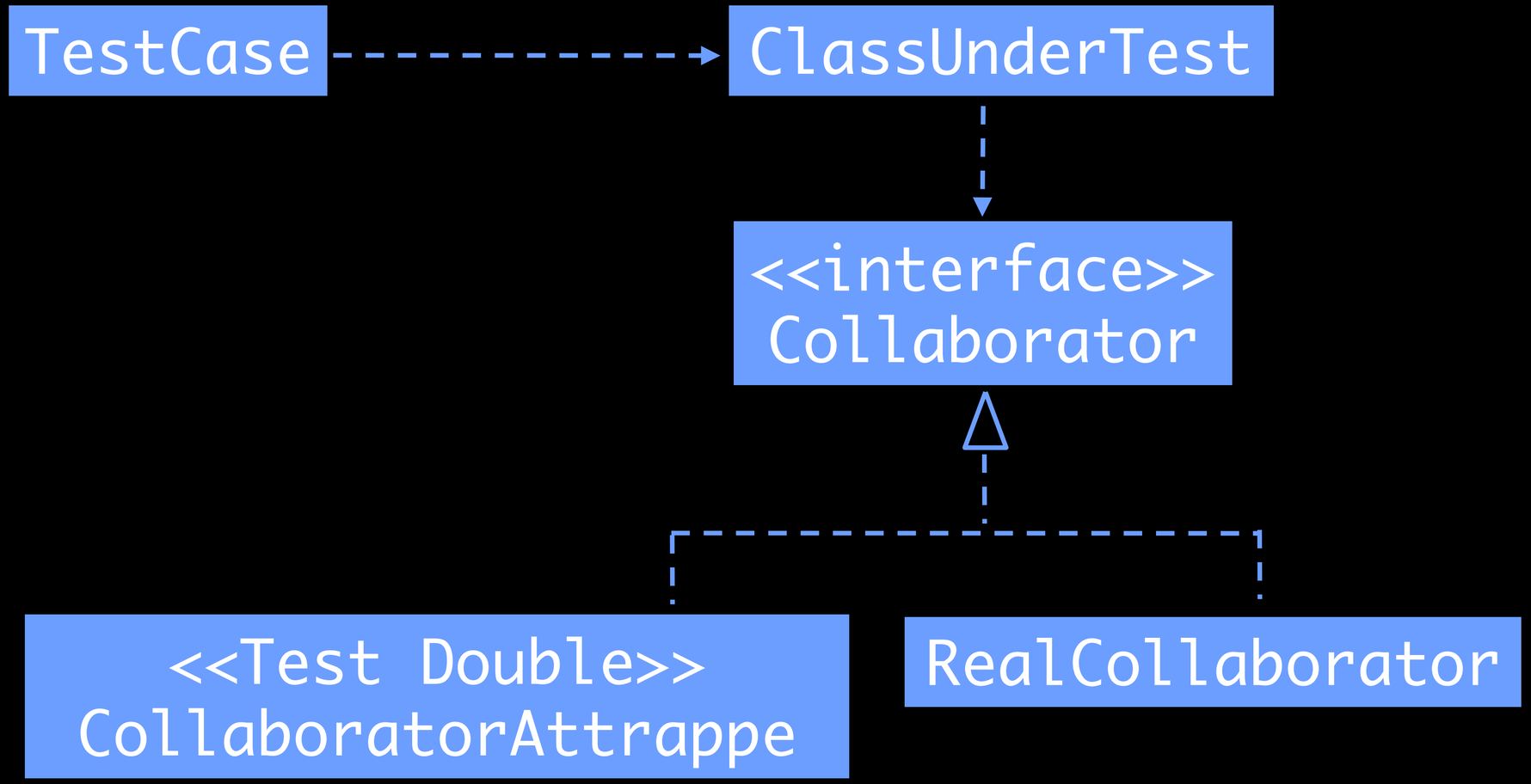
Isolated Testing

During automated testing we replace a component's dependencies by test doubles

Advantages:

- Tests run fast
- Occurring errors are easy to locate
- Testing all necessary combinations requires less effort than while testing with many integrated components at once

Test Doubles



How to test euroAmount()?

```
public class EuroConverter {
    private RateProvider provider;

    public EuroConverter(RateProvider provider) {
        this.provider = provider;
    }

    public double euroAmount(double amount, String curr) {
        return amount * provider.getRate(curr, "EUR");
    }
    ...
}
```

Weaker dependencies facilitate testing

Decoupling in test requires decoupled code:

```
public class RateProvider {
    public double getRate(String fromCurrency, String toCurrency) {
        // Make a long running web call
    }
}

public interface RateProvider {
    double getRate(String fromCurrency, String toCurrency);
}
```

Stubs

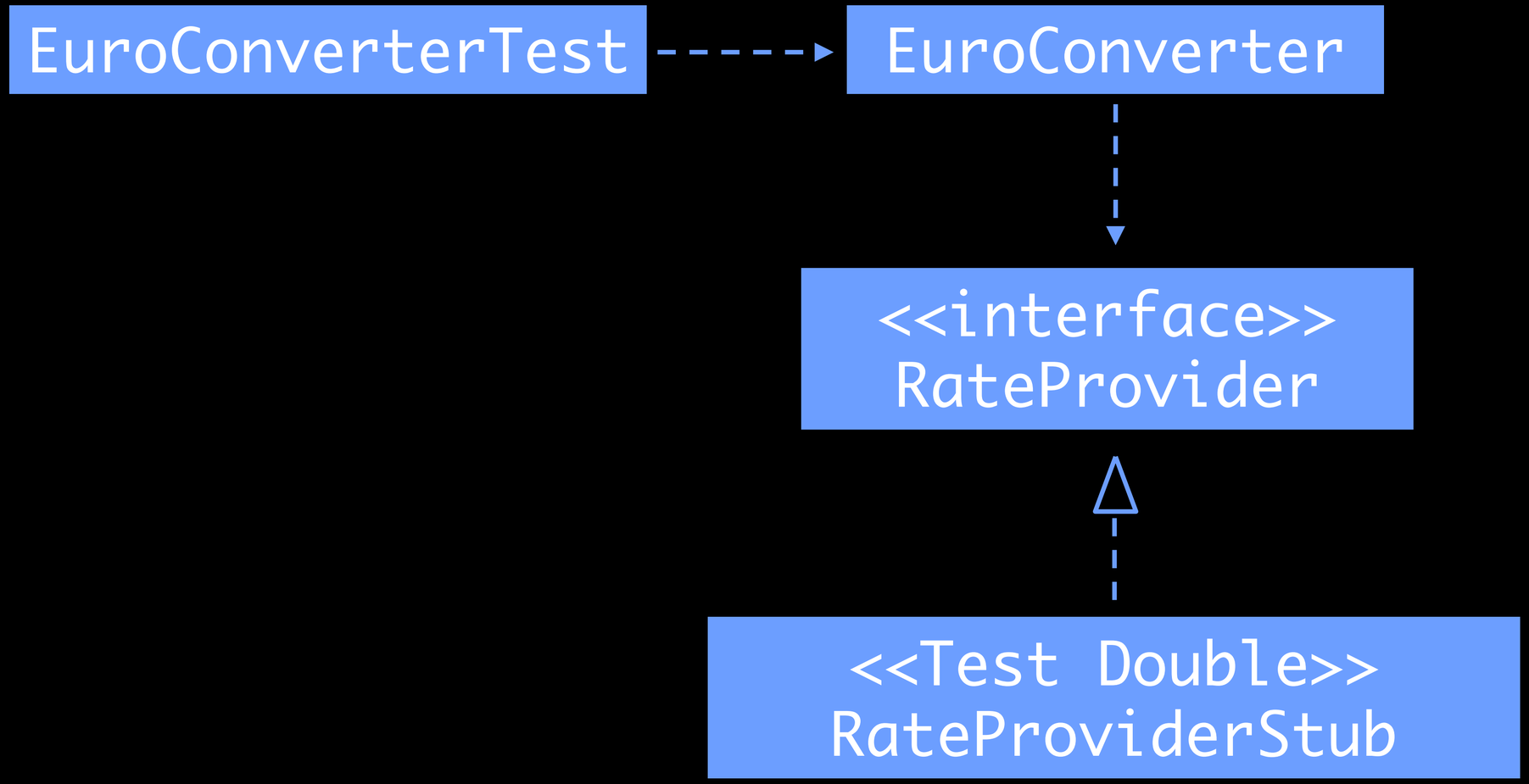
- The simplest type of test double
- Provide a simple implementation of a collaborator that returns fixed, predefines values

EuroConverterTest using a Stub

```
class EuroConverterTests {
    @Test
    void euroAmount() {
        RateProvider provider = new RateProviderStub();
        EuroConverter converter = new EuroConverter(provider);
        assertEquals(6.0, converter.euroAmount(3.0, "CHF"), 0.001);
    }

    static class RateProviderStub implements RateProvider {
        @Override
        public double getRate(String fromCurrency, String toCurrency) {
            return 2.0;
        }
    }
}
```

EuroConverterTest



Crash Test Dummies

```
@Test
void unknownCurrencyIsConvertedToZeroEuros() {
    RateProvider provider = (from, to) -> {
        throw new IllegalArgumentException();
    };
    EuroConverter converter = new EuroConverter(provider);
    assertEquals(0.0, converter.euroAmount(3.0, "XYZ"));
}
```

Mock Objects

- Stubs come with problems:
  - Method call and parameters are not checked!
- Mock objects are stubs with embedded checking
- Main types of mock objects
  - Endo mock (easymock, jmock)
  - Test Spy

Test Spy

- They allow stubbing by configuring defined answers to method calls
- They record all actual method calls
  - You can ask them later if the right call was made

Using a Test Spy

1. Create the spy object
2. Configure its stubbing behaviour
3. Inject spy in object under test
4. Run test
5. If necessary, verify that expected calls actually happened

Best known Java spy framework:
    Mockito: http://code.google.com/p/mockito

Mockito

Step 1: Create the spy object:

```
MyInterface mock = mock(MyInterface.class);
```

Step 2: Configure stubbing behaviour:

```
when(mock.myMethod(par1, par2)).thenReturn("result");
```

Step 3+4: Inject and run test...

Step 5: Verify expected calls:

```
verify(mock).myMethod(par1, par2); //notwendig?
```

EuroConverterTest using Mockito

```
import static org.mockito.Mockito.*;

class EuroConverterTests {
    @Test
    void euroAmountWithMockito() {
        RateProvider mockProvider = mock(RateProvider.class);
        when(mockProvider.getRate("CHF", "EUR")).thenReturn(2.0);
        EuroConverter converter = new EuroConverter(mockProvider);
        assertEquals(6.0, converter.euroAmount(3.0, "CHF"), 0.001);
        //not really necessary here:
        verify(mockProvider).getRate("CHF", "EUR");
    }
}
```

Test Double Glossary

according to [Meszaros07]

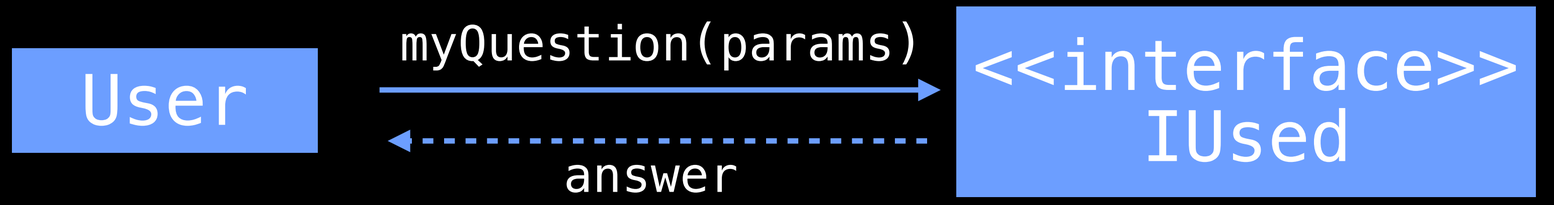
- General Term: Test Double
  - Test Stub
    - Mock Object
      - Test Spy
  - Fake Object
    - Simulates (part of) the real functionality
    - No embedded testing

Do we need Integrated Tests?

- Common wisdom:
  - Isolated tests do not find bugs related to faulty integration of several components
- Is that really true?

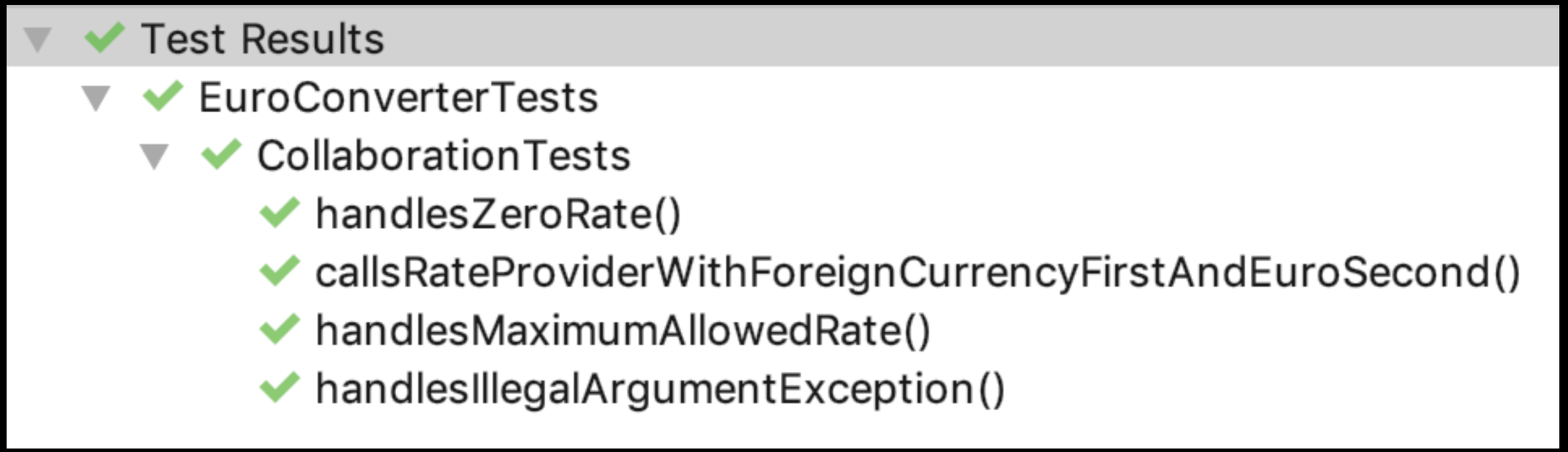
Can we test
integration aspects
without actually integrating
components?

Collaboration Tests

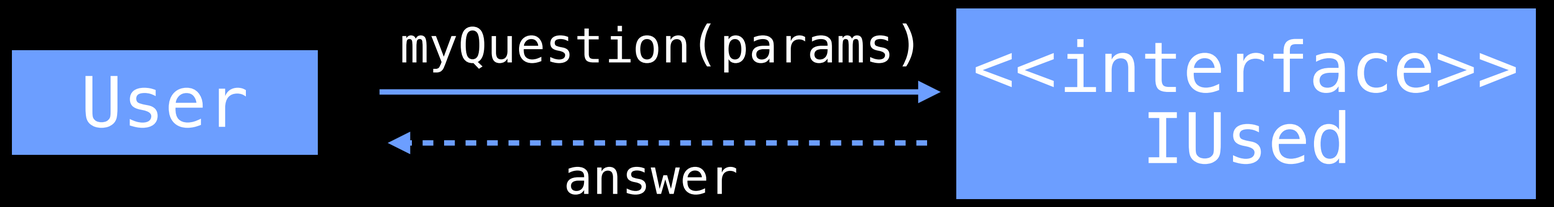


A. Test the User object

1. Does it ask the right questions?
2. Can it handle all allowed answers?

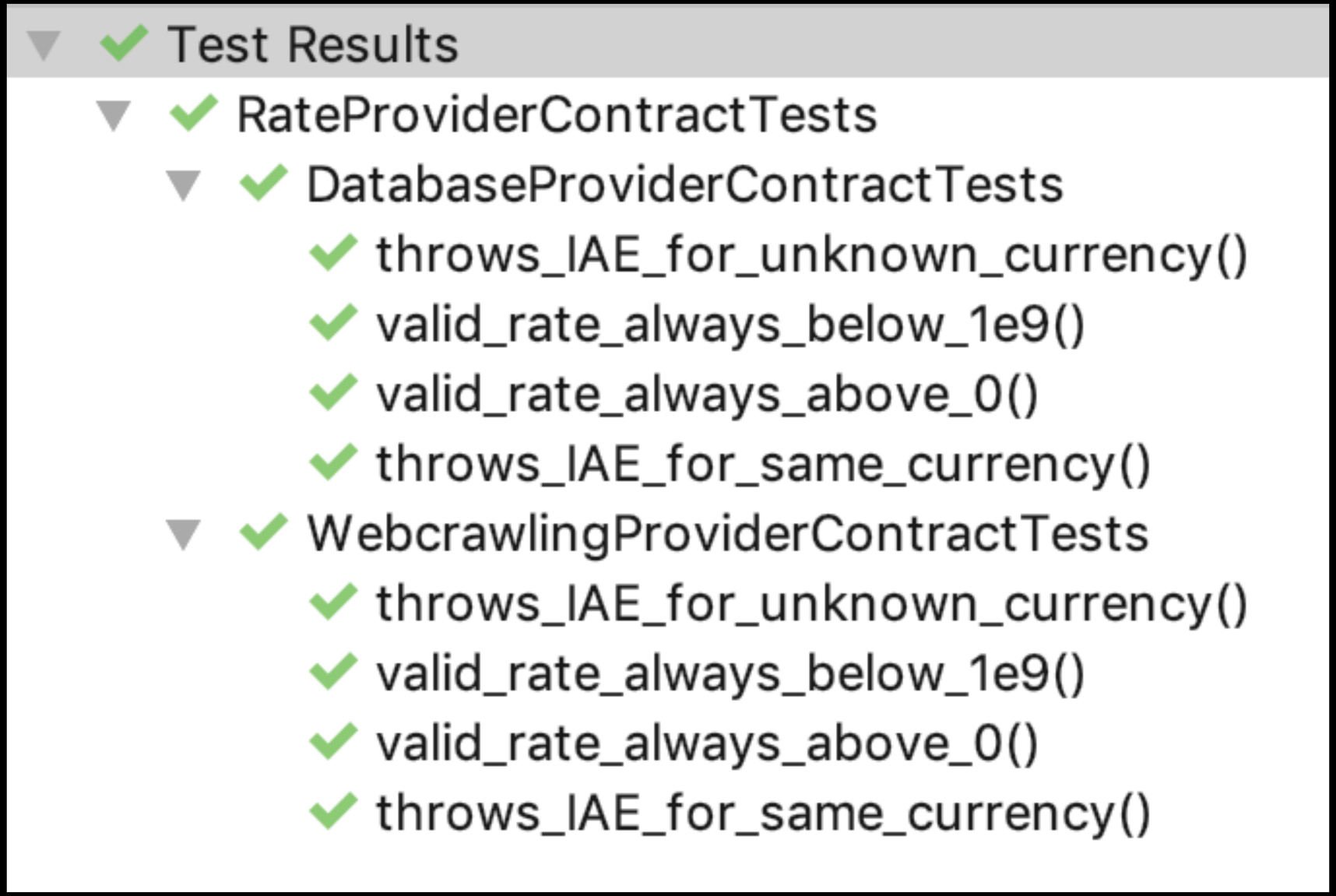

Test Results
EuroConverterTests
CollaborationTests
handlesZeroRate()
callsRateProviderWithForeignCurrencyFirstAndEuroSecond()
handlesMaximumAllowedRate()
handlesIllegalArgumentException()

Contract Tests



B. Test all implementations of `IUsed`:

1. Can they handle all questions?
2. Do they come up with the expected answers?


Test Results
RateProviderContractTests
DatabaseProviderContractTests
throws_IAE_for_unknown_currency()
valid_rate_always_below_1e9()
valid_rate_always_above_0()
throws_IAE_for_same_currency()
WebcrawlingProviderContractTests
throws_IAE_for_unknown_currency()
valid_rate_always_below_1e9()
valid_rate_always_above_0()
throws_IAE_for_same_currency()

```
class RateProviderContractTests {

    interface RateProviderContract {
        RateProvider createProvider();
        @Test
        default void throws_IAE_for_unknown_currency() { }
        @Test
        default void throws_IAE_for_same_currency() { }
        @Test
        default void valid_rate_always_above_0() { }
        @Test
        default void valid_rate_always_below_1e9() { }
    }

    @Nested
    class WebcrawlingProviderContractTests implements RateProviderContract {
        @Override
        public RateProvider createProvider() {
            return new WebcrawlingRateProvider();
        }
    }

    @Nested
    class DatabaseProviderContractTests implements RateProviderContract…
}
```

Basic Correctness

"If I ran the system on perfect technology, would it (eventually) compute the right answer every time?" (J.B. Rainsberger)

- Complete collaboration and contract testing can assure basic correctness
- With basic correctness present we now have time for the remaining technology-dependent problems

Do we need Integrated Tests?

- Collaboration and Contract tests are microtests:
  They scale much better and are less costly
- We still need integrated tests
  - to verify technological complications,
    e.g. concurrency and networks
  - to verify integration with external components and libraries,
    e.g. databases and web services
- Consumer-driven Contracts:
  Get rid of integrated tests with micro-services

Property-Based Testing

Example-based Tests

An *example* shows that the code delivers
a specific result

```
@Example
void reverseList() {
   List<Integer> aList = Arrays.asList(1, 2, 3);
   Collections.reverse(aList);
   assertThat(aList).containsExactly(3, 2, 1);
}
```

Does *reverse()* only work for the tested examples?

How representative are our examples?

How many examples does it take to create enough trust?

```
@Example void emptyList() {
    List<Integer> aList = Collections.emptyList();
    assertThat(Collections.reverse(aList)).isEmpty();
}

@Example void oneElement() {
    List<Integer> aList = Collections.singletonList(1);
    assertThat(Collections.reverse(aList)).containsExactly(1);
}

@Example void manyElements() {
    List<Integer> aList = asList(1, 2, 3, 4, 5, 6);
    assertThat(Collections.reverse(aList)).containsExactly(6, 5, 4, 3, 2,
1);
}

@Example void duplicateElements() {
    List<Integer> aList = asList(1, 2, 2, 4, 6, 6);
```

Properties

A *Property* shows that
for a class of inputs (aka *preconditions*)

```
@Property
void reverseList() {
    // preconditions?
    // postconditions and invariants?
}
```

```
Collections.reverse(List aList):
   // preconditions?
   // postconditions and invariants?
```

Preconditions

- Any non-null list

Invariants

- Size of list remains the same
- All elements stay in list
- After reversing the first element becomes the last
- Applying reverse twice produces the original list

A Property in Java Code

```
boolean theSizeRemainsTheSame(List<Integer> original) {
    List<Integer> reversed = reverse(original);
    return original.size() == reversed.size();
}


private <T> List<T> reverse(List<T> original) {
    List<T> clone = new ArrayList<>(original);
    Collections.reverse(clone);
    return clone;
}
```

Jqwik

```
@Property
boolean theSizeRemainsTheSame(@ForAll List<Integer>
original) {
   List<Integer> reversed = reverse(original);
   return original.size() == reversed.size();
}
```

Demo

- `pbt.reverse.ReverseListTests`
- `pbt.reverse.ReverseListProperties`
- Integration in Gradle & IntelliJ

What jqwik is…

https://jqwik.net

- Test engine for the JUnit 5 platform
- Generator for test cases creating
  - random and typical input data
  - sometimes even an exhaustive set of all possible input combinations
- Current version: 0.9.2

What jqwik is not…

- It's not a fully randomized testing tool, which can be applied on your software without thinking
- Properties cannot be proven, they can only be falsified

```java
@Property
void squareOfRootIsOriginalValue(@ForAll double aNumber) {
    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}
```

```
java.lang.AssertionError:
Expecting:
  <NaN>
to be close to:
  <-1.0>
by less than 1% but difference was NaN%.
(a difference of exactly 1% being considered valid)
```

Constraining Value Generation

Often a Property is only valid for a constrained subset of a given type

```java
@Property
void squareOfRootIsOriginalValue(
    @ForAll @Positive double aNumber
) {
    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}
```

```
timestamp = 2017-10-20T17:23:53.351,
  tries = 1000,
  checks = 1000,
  seed = 7890962728489990406
```

```
@Property
void squareOfRootIsOriginalValue(
    @ForAll("positiveDoubles") double aNumber
) {
    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}

@Provide
Arbitrary<Double> positiveDoubles() {
    return Arbitraries.doubles().between(0, Double.MAX_VALUE);
}
```

```
timestamp = 2017-10-20T17:23:53.351,
  tries = 1000,
  checks = 1000,
  seed = 7890962728489990406
```

How to Generate Values

Fluent Interfaces

```
@Provide
StringArbitrary fluentString() {
   return Arbitraries.strings()
                     .alpha()
                     .numeric()
                     .withChars('?', '!', '.')
                     .ofMinLength(2)
                     .ofMaxLength(10);
}
```

Changing Generated Values

- Sometimes you want to filter generate values yourself
- Sometimes you want to map generated values to others
- Sometimes you want to combine generated values with each other

Filtering

```
@Property
boolean evenNumbersAreEven(@ForAll("evenUpTo10000") int anInt) {
    return anInt % 2 == 0;
}

@Provide
Arbitrary<Integer> evenUpTo10000() {
    return Arbitraries.integers()
                .between(0, 10000)
                .filter(i -> i % 2 == 0);
}
```

Mapping

```
@Provide
Arbitrary<Integer> evenUpTo10000() {
    return Arbitraries.integers()
                .between(0, 5000)
                .map(i -> i * 2);
}


@Provide
Arbitrary<Integer> evenUpTo10000() {
    return Arbitraries.integers()
                .between(0, 10000)
                .filter(i -> i % 2 == 0);
}
```

Combining

```
public class Person {
    public Person(String firstName, String lastName) {…}
    public String fullName() {return firstName + " " + lastName;}
}

    Arbitrary<Character> initialChar =
Arbitraries.chars().between('A', 'Z');
    Arbitrary<String> firstName = Arbitraries.strings()… ;
    Arbitrary<String> lastName = Arbitraries.strings()… ;
    return Combinators.combine(initialChar, firstName,
lastName)
            .as((initial, first, last) -> new Person(initial + first,
last));
}
```

```
static <E> List<E> brokenReverse(List<E> aList) {
    if (aList.size() < 4) {
        aList = new ArrayList<>(aList);
        reverse(aList);
    }
    return aList;
}

@Property(shrinking = ShrinkingMode.OFF)
boolean reverseShouldSwapFirstAndLast(@ForAll List<Integer> aList) {
    Assume.that(!aList.isEmpty());
    List<Integer> reversed = brokenReverse(aList);
    return aList.get(0) == reversed.get(aList.size() - 1);
}
```

```
org.opentest4j.AssertionFailedError:
Property [reverseShouldSwapFirstAndLast] falsified with sample
   [[0, -2147483648, 2147483647, -7997, 7997, -3223, -6474, 1915, -7151,
   3102, 4362, 714, 3053, 1919, -445, 7498, -2424, 3016, -5127, -7401, -7946,
   -3801, -305]]
```

```
static <E> List<E> brokenReverse(List<E> aList) {
    if (aList.size() < 4) {
        aList = new ArrayList<>(aList);
        reverse(aList);
    }
    return aList;
}

@Property
boolean reverseShouldSwapFirstAndLast(@ForAll List<Integer> aList) {
    Assume.that(!aList.isEmpty());
    List<Integer> reversed = brokenReverse(aList);
    return aList.get(0) == reversed.get(aList.size() - 1);
}
```

```
org.opentest4j.AssertionFailedError:
Property [reverseShouldSwapFirstAndLast] falsified with sample
   [[0, 0, 0, -1]]
```

```java
static <E> List<E> brokenReverse(List<E> aList) {
    if (aList.size() < 4) {
        aList = new ArrayList<>(aList);
        reverse(aList);
    }
    return aList;
}

@Property
boolean reverseShouldSwapFirstAndLast(@ForAll List<Integer> aList) {
    Assume.that(!aList.isEmpty());
    List<Integer> reversed = brokenReverse(aList);
    return aList.get(0) == reversed.get(aList.size() - 1);
}
```

```
org.opentest4j.AssertionFailedError:
Property [reverseShouldSwapFirstAndLast] falsified with sample
   [[0, 0, 0, -1]]
```

The Importance of Being Shrunk

- Shrinking of falsified property: Trying to find the simplest set of inputs to make the property fail
- Sometimes there is no "simplest" failing example or finding it would take very long
- Use heuristics to shrink values
  - try integer closer to 0
  - try collection with fewer elements
- Requires full determinism of property method

Patterns of PBT

- Obvious Property
- Fuzzying
- Inverse functions
- Idempotent functions
- Commutativity
- Black-box testing
- Induction
- Test oracle
- Invariant properties
- Stateful Testing

Demo: Fizz Buzz

- Count from 1 to 100
- "Normal" numbers are given their "normal" name
- Multiples of 3 are named "Fizz"
- Multiples of 5 are named "Buzz"
- Multiples of 3 and 5 are named "FizzBuzz"

PBT: Summary

- Property-Based Testing is an additional tool in a developer's tool box
- Property Tests can detect
  - Bugs in the implementation
  - Gaps and Misunderstandings in the specification
- Sometimes concrete examples are more helpful for understanding and describing a system's behaviour

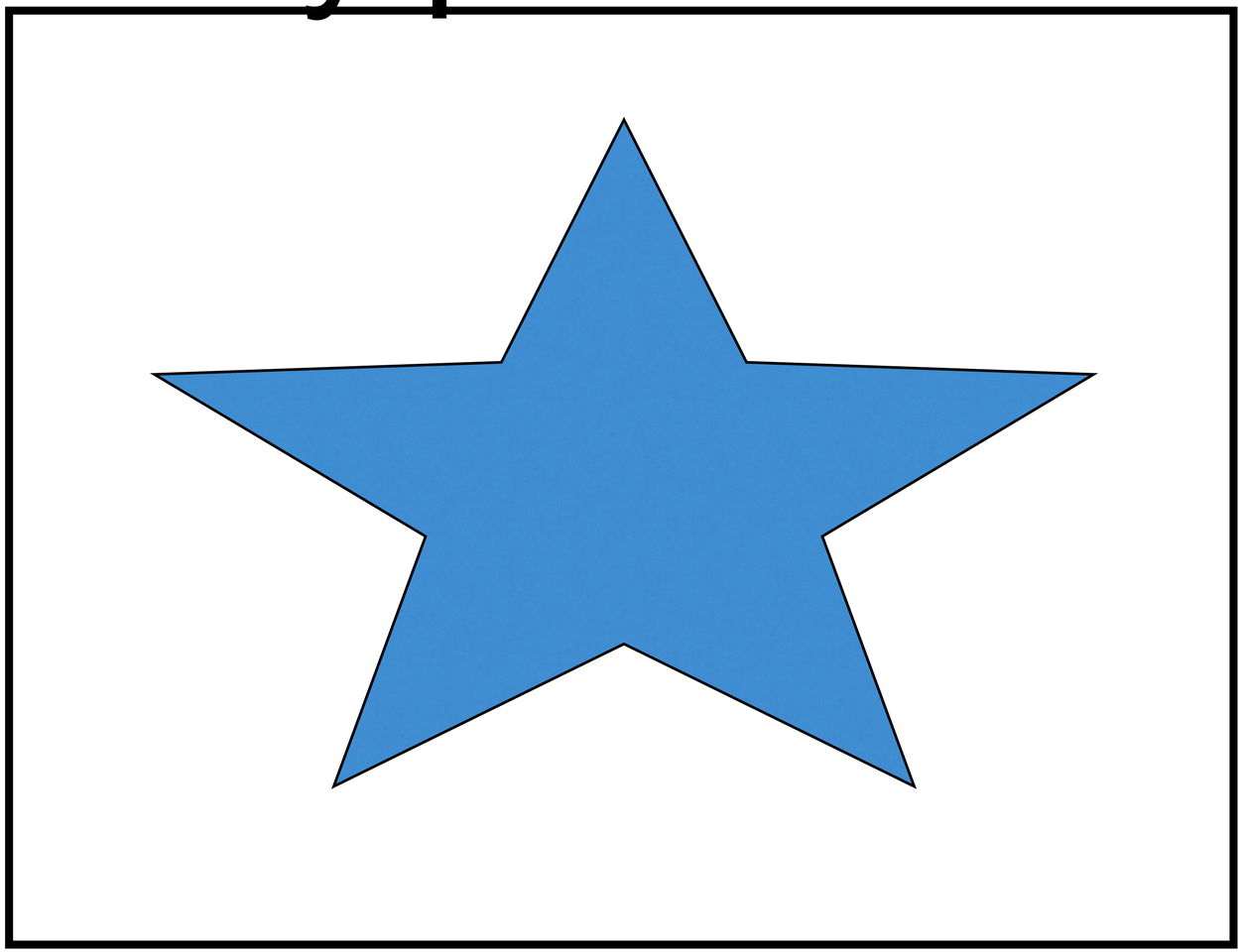
Acceptance Testing
Specification by Example

- Developer Testing: Have we developed what we want to develop?
- How do we make sure that we develop systems that match the customers' and users' expectations?

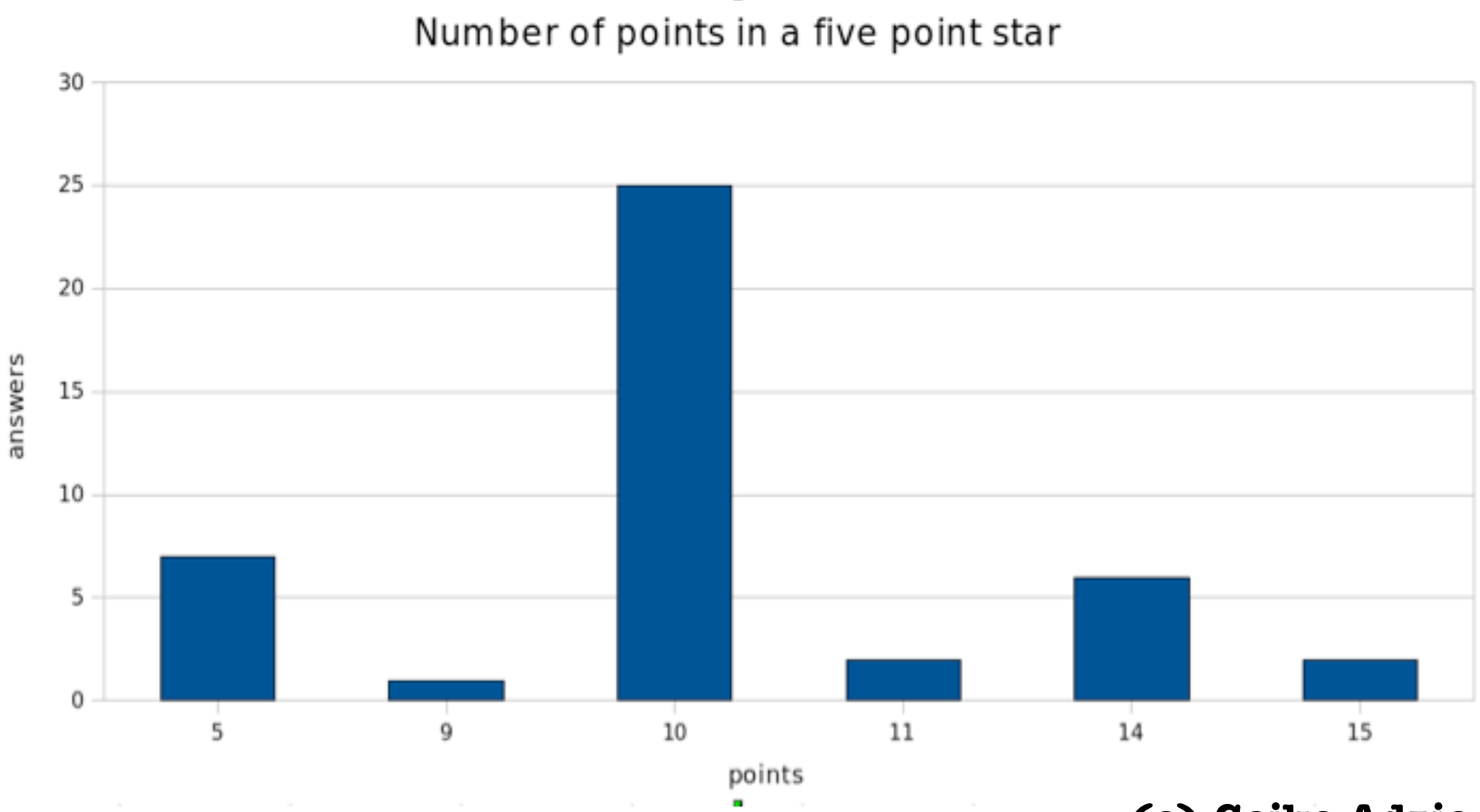
Requirements Documents

… are always ambiguous

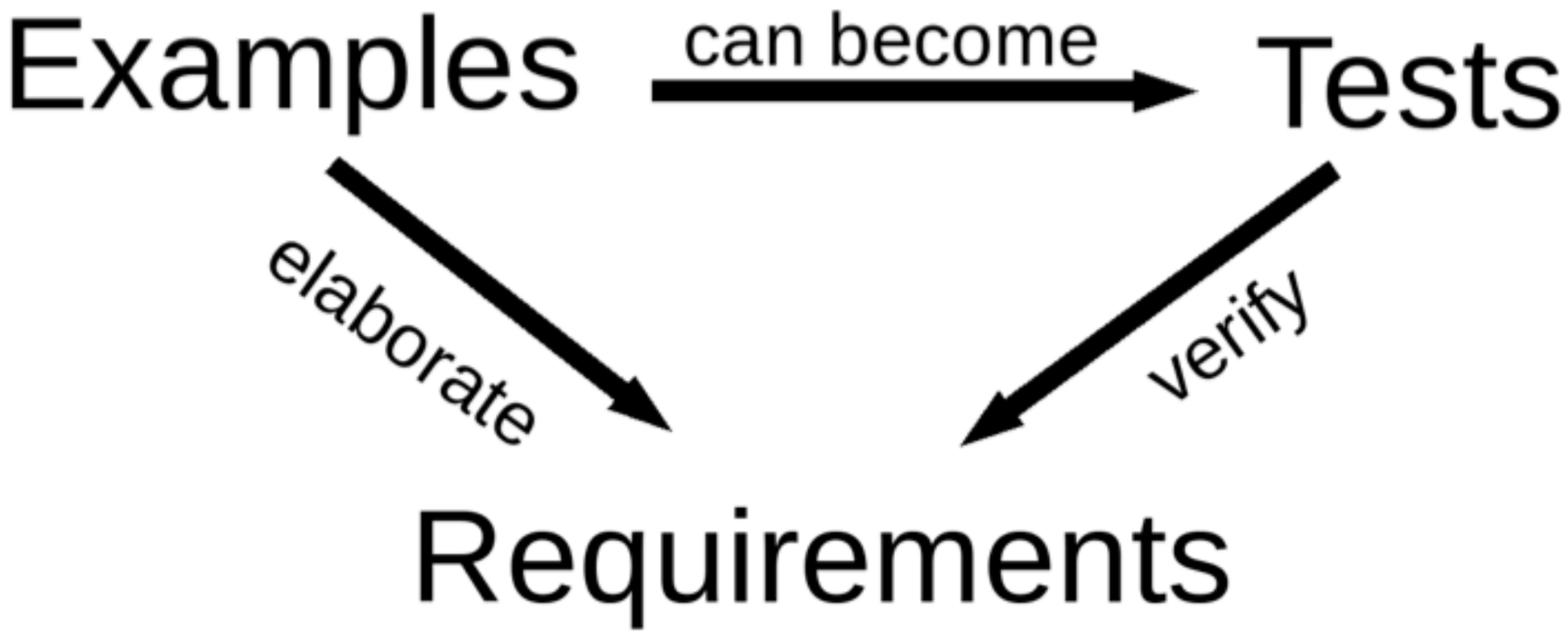
How many points are there?



How many points are there?



Vision:
A requirements document that
can be verified automatically



Real-world examples help to flush out incorrect **assumed** rules and to find **real** business rules!

Key Practices

- Discuss real-world examples to build a shared understanding of the domain
- Choose the crucial subset of discussed examples as acceptance criteria
- Automate examples as acceptance tests
- Use the tests as live specification to facilitate change

Three Amigos

- Specification workshops require at least the presence of three roles
  - Business Expert
  - Developer
  - Tester
- Run workshops shortly before implementation not in the beginning of a project
- The whole team should understand the selected crucial examples

Automation

- Domain language must be supported
- Abstractions and modularisation
- Version spec with program code
- Supports collaboration of all people involved

Cucumber

```
Feature: Creating New Accounts
  In order to allow financial transactions for a customer
  As a bank manager
  I want to create an account for a customer

  Scenario: Create Customer's First Account
    Given there are no accounts
    When I create an account for "Johannes"
    Then the new account has owner "Johannes"
      And the new account has id "0000001"
      And the new account has balance EUR 0.0
```

Fit/Fitness
Table-oriented and Wiki-style Collaboration

eg.Division		
numerator	denominator	quotient?
10	2	5.0
12.6	3	4.2
22	7	~=3.14
9	3	<5
11	2	4<_<6
100	4	33

Points of Attack

- Most Acceptance Tests are no end-to-end tests
- Automated User Examples can be plugged into the system at different levels
  - UI
  - API
  - Domain Layer

Acceptance Testing can go wrong…

- No collaboration between developers, testers and domain experts
- ATs are abused as replacement for good developer tests
- Maintenance of ATs is neglected

➡ Communication is everything

➡ It's only worth it if domain experts use them

Code:

http://github.com/jlink/tdd-fau

Slides:

http://github.com/jlink/tdd-fau/slides

Sources

- Meszaros:
  xUnit Test Patterns - Refactoring Test Code
- JB Rainsberger:
  Integrated Tests are a Scam
  https://blog.thecodewhisperer.com/permalink/integrated-tests-are-a-scam
- Gojko Adzic:
  Bridging the Communication Gap - Specification by Example and Agile Acceptance Testing