Effektives Contract Testing

JAX, Mainz
7. Mai 2019

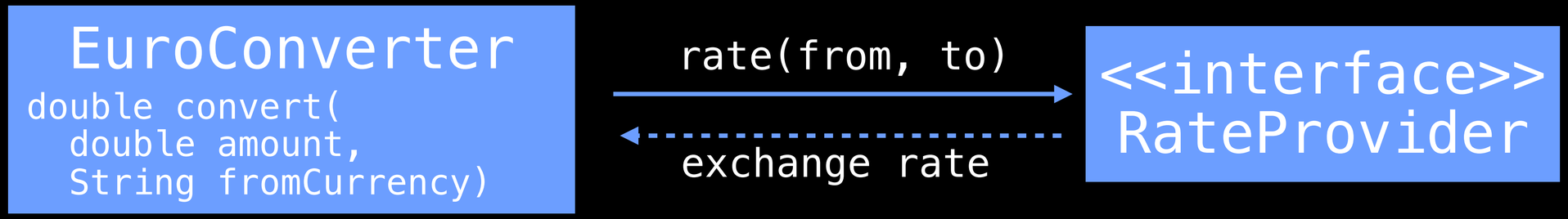
@johanneslink

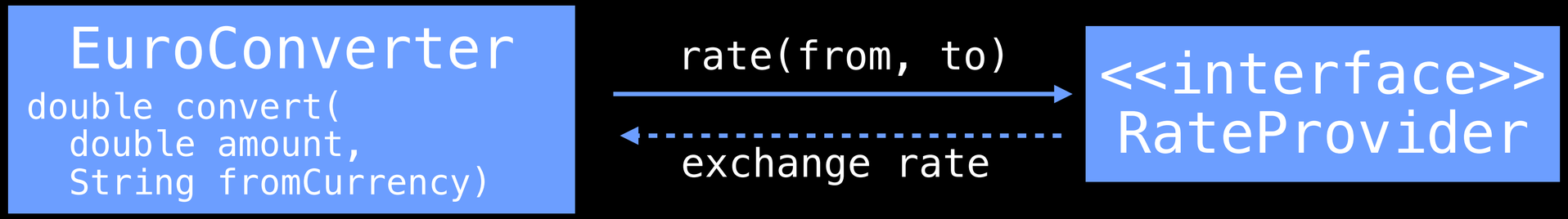
johanneslink.net

Softwaretherapeut

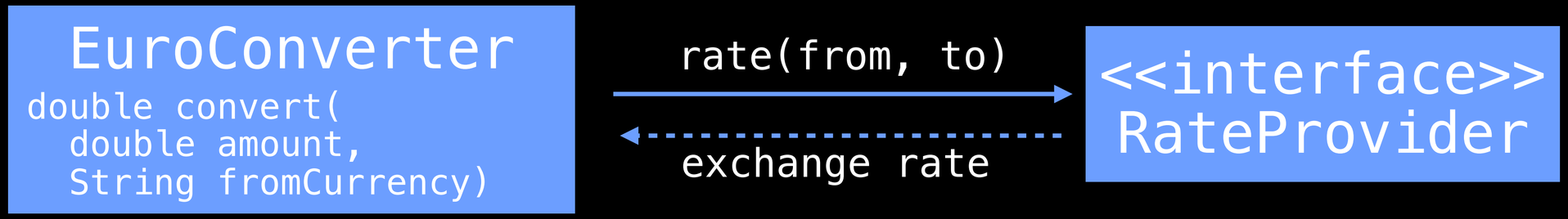
Softwaretherapeut

"In Deutschland ist die Bezeichnung Therapeut allein oder ergänzt mit bestimmten Begriffen gesetzlich nicht geschützt und daher kein Hinweis auf ein erfolgreich abgeschlossenes Studium oder auch nur fachliche Kompetenz." Quelle: Wikipedia


EuroConverter
double convert(
double amount,
String fromCurrency)
rate(from, to)
exchange rate
<<interface>>
RateProvider




```
interface RateProvider {
    double rate(String fromCurrency, String toCurrency)
            throws UnknownCurrency;
}
```




```
interface RateProvider {
    double rate(String fromCurrency, String toCurrency)
            throws UnknownCurrency;
}
```

Was kann beim Zusammenspiel zwischen EuroConverter und RateProvider schiefgehen?

Benötigen wir integrierte Tests um das Zusammenspiel zweier Komponenten ausreichend abzusichern?

Benötigen wir integrierte Tests um das Zusammenspiel zweier Komponenten ausreichend abzusichern?

- Entwickler-Folklore:
  "Isolierte Tests finden keine Fehler, die durch das fehlerhafte Zusammenspiel von Komponenten zustande kommen"

Benötigen wir integrierte Tests um das Zusammenspiel zweier Komponenten ausreichend abzusichern?

- Entwickler-Folklore:
  "Isolierte Tests finden keine Fehler, die durch das fehlerhafte Zusammenspiel von Komponenten zustande kommen"
- Probleme von integrierten Tests
  - Hoher Erstellungsaufwand
  - Längere Laufzeiten
  - Größerer Analyseaufwand
  - Zunehmender Indeterminismus

Wie testen wir die
Integrationsaspekte
ohne die Komponenten
tatsächlich zu integrieren?

Design by Contract

Consumer und Supplier eines Dienstes müssen sich bei ihrer Zusammenarbeit (aka Integration) an einen Vertrag halten. Der Vertrag wird beschrieben durch:

- Vorbedingungen
- Nachbedingungen
- Invarianten

Vorbedingung

- Eine Vorbedingung muss erfüllt sein, damit der Aufruf einer Funktion überhaupt sinnvoll ist
- Für das Einhalten der Vorbedingung ist der Consumer verantwortlich

Nachbedingung

- Der Consumer kann sich darauf verlassen, dass die Nachbedingungen eines Vertrags erfüllt werden
- Der Supplier ist für das Erfüllen der Nachbedingungen eines Vertrags verantwortlich

Invariante

- Eine Invariante beschreibt eine Eigenschaft einer (zustandsbehafteten) Komponente, die zu jeder beobachtbaren Zeit erfüllt sein muss
- Der Supplier ist für das Einhalten von Invarianten verantwortlich

RateProvider Contract

```
interface RateProvider {
    double rate(String fromCurrency, String toCurrency) throws RateNotAvailable;
}
```

RateProvider Contract

```
interface RateProvider {
    double rate(String fromCurrency, String toCurrency) throws RateNotAvailable;
}
```

- Vorbedingungen

RateProvider Contract

```
interface RateProvider {
    double rate(String fromCurrency, String toCurrency) throws RateNotAvailable;
}
```

- Vorbedingungen
  - Nur 3-Buchstaben-Codes als Währungen erlaubt

RateProvider Contract

```
interface RateProvider {
    double rate(String fromCurrency, String toCurrency) throws RateNotAvailable;
}
```

- Vorbedingungen
  - Nur 3-Buchstaben-Codes als Währungen erlaubt
  - Währungen in Reihenfolge: `fromCurrency, toCurrency`

RateProvider Contract

```
interface RateProvider {
    double rate(String fromCurrency, String toCurrency) throws RateNotAvailable;
}
```

- Vorbedingungen
  - Nur 3-Buchstaben-Codes als Währungen erlaubt
  - Währungen in Reihenfolge: `fromCurrency, toCurrency`
- Nachbedingungen

RateProvider Contract

```
interface RateProvider {
    double rate(String fromCurrency, String toCurrency) throws RateNotAvailable;
}
```

- Vorbedingungen
  - Nur 3-Buchstaben-Codes als Währungen erlaubt
  - Währungen in Reihenfolge: `fromCurrency, toCurrency`
- Nachbedingungen
  - Unterstützte Währungen:
    minimum exchange rate <= rate <= maximum exchange rate

RateProvider Contract

```
interface RateProvider {
    double rate(String fromCurrency, String toCurrency) throws RateNotAvailable;
}
```

- Vorbedingungen
  - Nur 3-Buchstaben-Codes als Währungen erlaubt
  - Währungen in Reihenfolge: `fromCurrency, toCurrency`
- Nachbedingungen
  - Unterstützte Währungen:
    minimum exchange rate <= rate <= maximum exchange rate
  - Nichtunterstützte Währung: RateNotAvailable

RateProvider Contract

```
interface RateProvider {
    double rate(String fromCurrency, String toCurrency) throws RateNotAvailable;
}
```

- Vorbedingungen
  - Nur 3-Buchstaben-Codes als Währungen erlaubt
  - Währungen in Reihenfolge: `fromCurrency, toCurrency`
- Nachbedingungen
  - Unterstützte Währungen:
    minimum exchange rate <= rate <= maximum exchange rate
  - Nichtunterstützte Währung: RateNotAvailable
  - Unbekannte Währung: IllegalArgumentException

RateProvider Contract

```
interface RateProvider {
    double rate(String fromCurrency, String toCurrency) throws RateNotAvailable;
}
```

- Vorbedingungen
  - Nur 3-Buchstaben-Codes als Währungen erlaubt
  - Währungen in Reihenfolge: `fromCurrency, toCurrency`
- Nachbedingungen
  - Unterstützte Währungen:
    minimum exchange rate <= rate <= maximum exchange rate
  - Nichtunterstützte Währung: RateNotAvailable
  - Unbekannte Währung: IllegalArgumentException
- Invarianten

RateProvider Contract

```
interface RateProvider {
    double rate(String fromCurrency, String toCurrency) throws RateNotAvailable;
}
```

- Vorbedingungen
  - Nur 3-Buchstaben-Codes als Währungen erlaubt
  - Währungen in Reihenfolge: `fromCurrency, toCurrency`
- Nachbedingungen
  - Unterstützte Währungen:
    minimum exchange rate <= rate <= maximum exchange rate
  - Nichtunterstützte Währung: RateNotAvailable
  - Unbekannte Währung: IllegalArgumentException
- Invarianten
  - rate(X, Y)*rate(Y, X) < 1.0

RateProvider Contract

```
interface RateProvider {
    double rate(String fromCurrency, String toCurrency) throws RateNotAvailable;
}
```

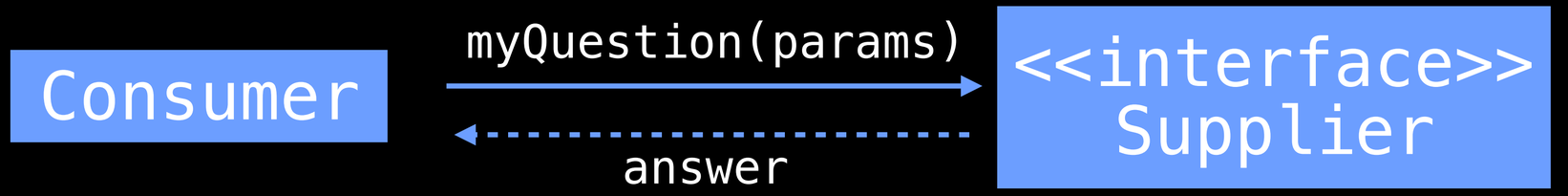
- Vorbedingungen
  - Nur 3-Buchstaben-Codes als Währungen erlaubt
  - Währungen in Reihenfolge: `fromCurrency, toCurrency`
- Nachbedingungen
  - Unterstützte Währungen:
    minimum exchange rate <= rate <= maximum exchange rate
  - Nichtunterstützte Währung: RateNotAvailable
  - Unbekannte Währung: IllegalArgumentException
- Invarianten
  - rate(X, Y)*rate(Y, X) < 1.0
  - rate(X, Y) == rate(X, Y) ?

Wie testen wir einen Vertrag?

ohne die Komponenten tatsächlich zu integrieren

- Collaboration Tests
- Contract Tests

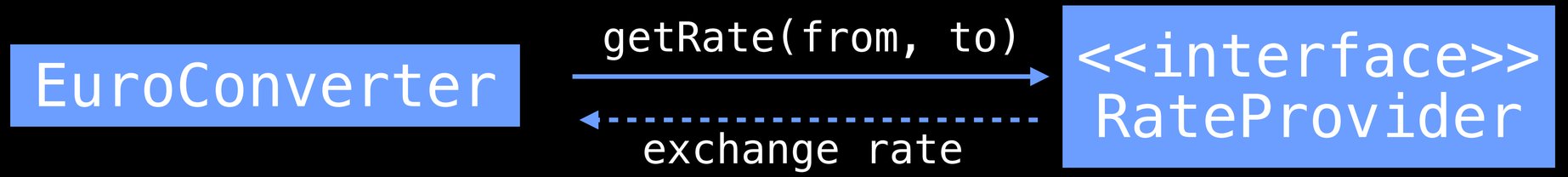
Collaboration Tests

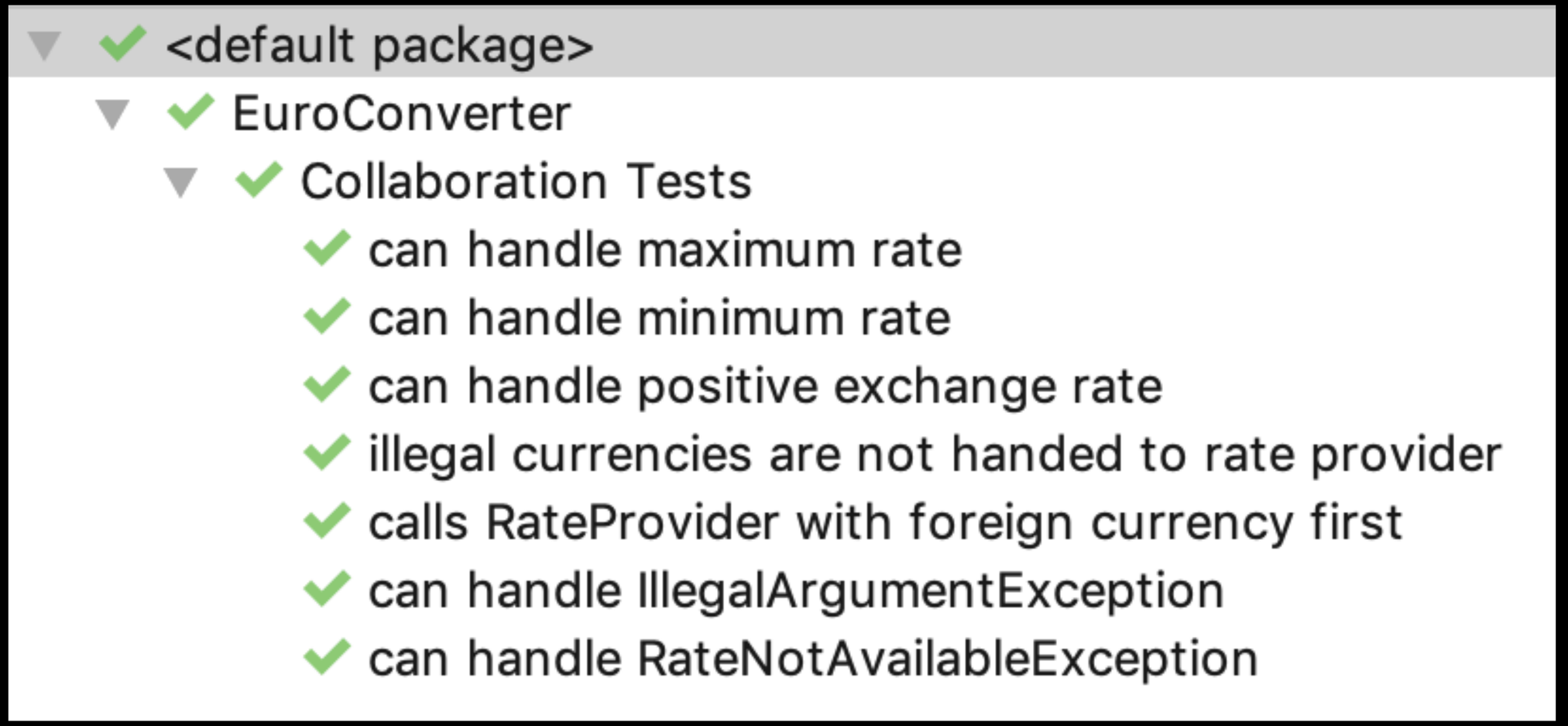


A. Test the Consumer object

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

Collaboration Tests

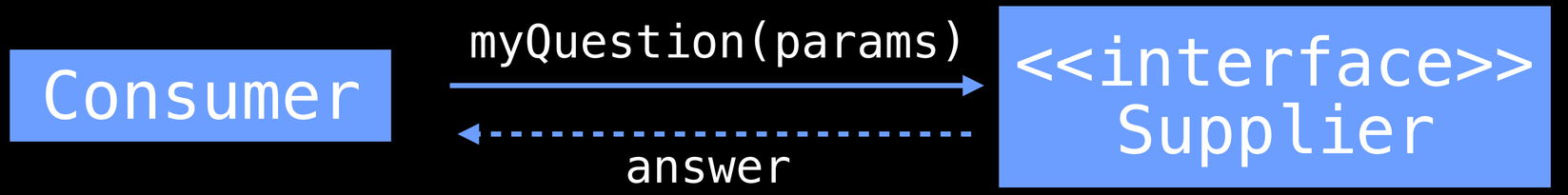



<default package>
EuroConverter
Collaboration Tests
can handle maximum rate
can handle minimum rate
can handle positive exchange rate
illegal currencies are not handed to rate provider
calls RateProvider with foreign currency first
can handle IllegalArgumentException
can handle RateNotAvailableException

```java
@Example
void calls_RateProvider_with_foreign_currency_first() throws RateNotAvailable {
    RateProvider provider = Mockito.mock(RateProvider.class);
    Mockito.when(provider.rate("USD", "EUR")).thenReturn(0.5);

    double euroAmount = new EuroConverter(provider).convert(10.0, "USD");
    Assertions.assertThat(euroAmount).isCloseTo(5.0, Offset.offset(0.01));
}
```

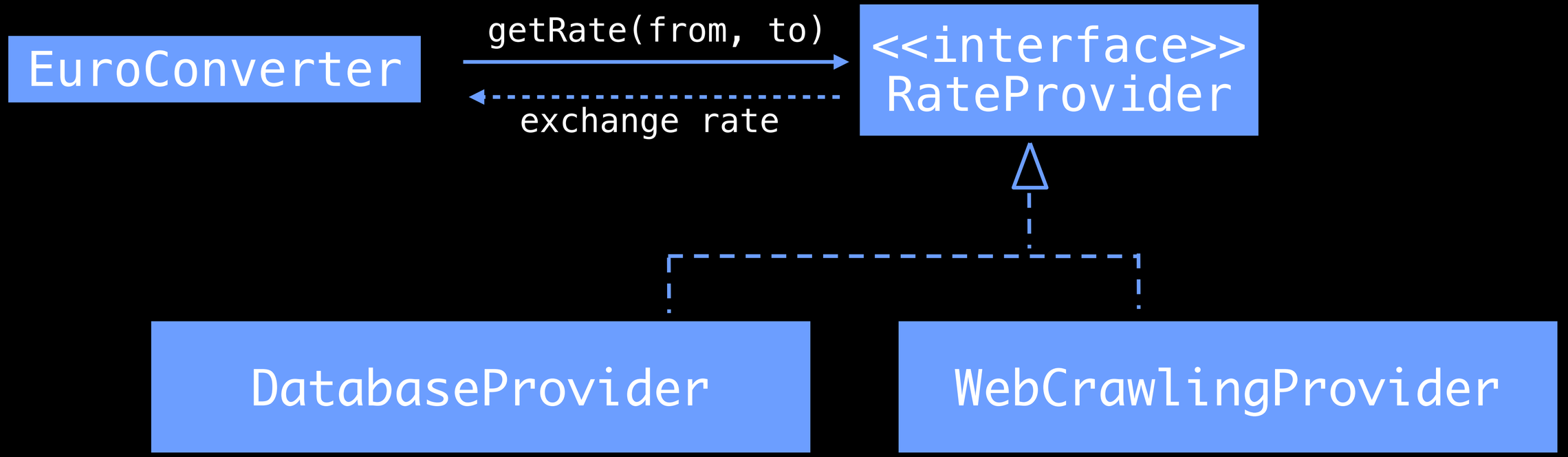
Contract Tests

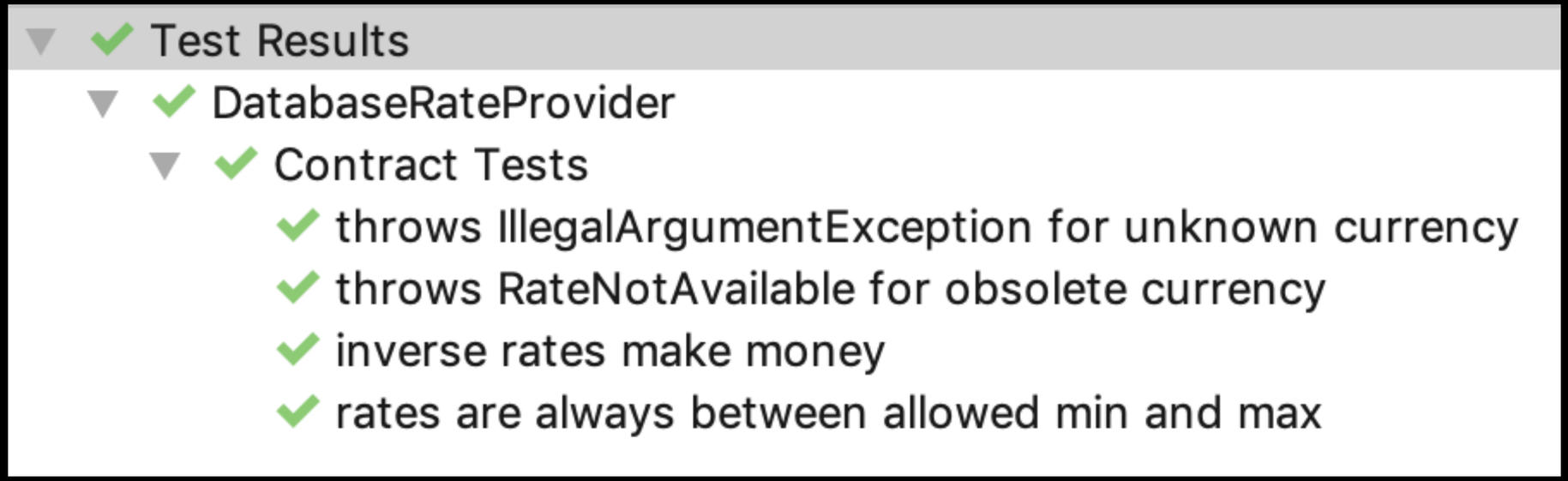


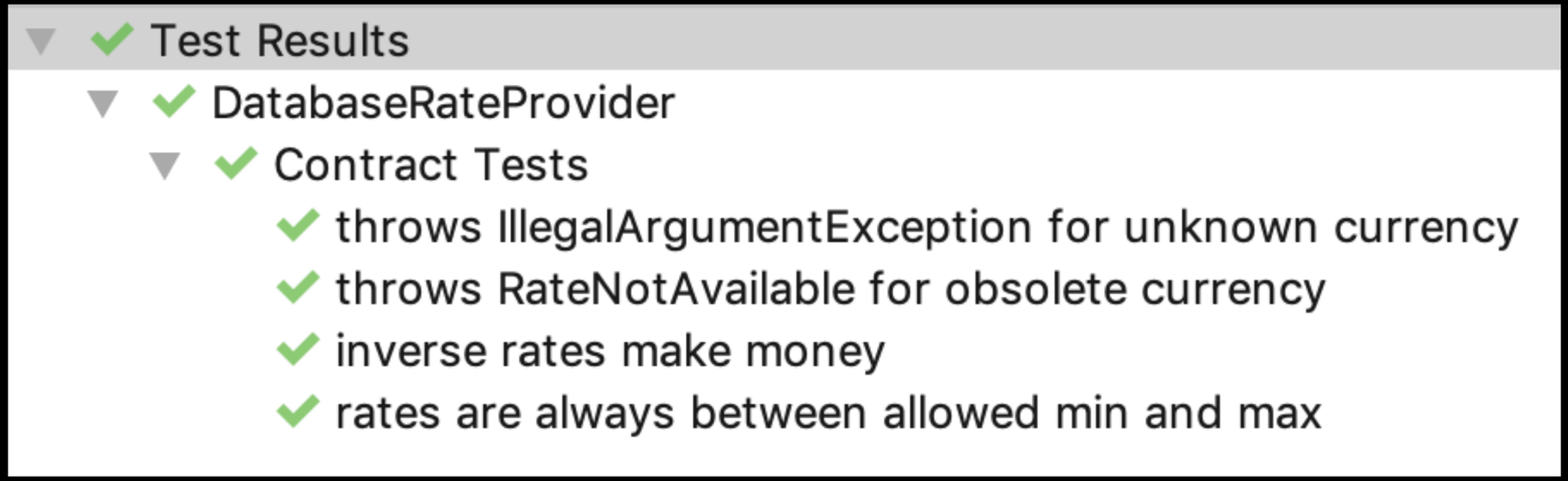
B. Test all implementations of `Supplier`:

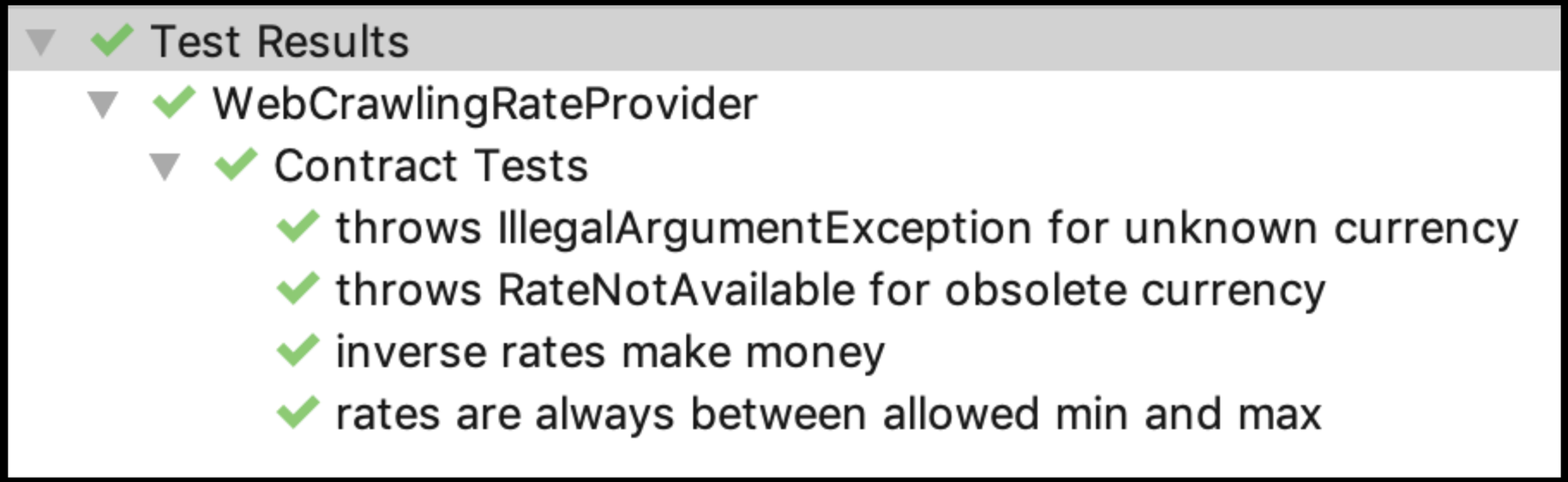
1. Can they handle all questions?
2. Do they come up with the expected answers?

Contract Tests




Test Results
DatabaseRateProvider
Contract Tests
throws IllegalArgumentException for unknown currency
throws RateNotAvailable for obsolete currency
inverse rates make money
rates are always between allowed min and max


Test Results
DatabaseRateProvider
Contract Tests
throws IllegalArgumentException for unknown currency
throws RateNotAvailable for obsolete currency
inverse rates make money
rates are always between allowed min and max


Test Results
WebCrawlingRateProvider
Contract Tests
throws IllegalArgumentException for unknown currency
throws RateNotAvailable for obsolete currency
inverse rates make money
rates are always between allowed min and max

```java
interface RateProviderContractTests {
    RateProvider createProvider();

    @Test
    default void throws_IllegalArgumentException_for_unknown_currency() {
        RateProvider provider = createProvider();
        Assertions.assertThatThrownBy(() -> provider.rate("XYZ", "USD"))
                  .isInstanceOf(IllegalArgumentException.class);
    }

    @Test
    default void throws_RateNotAvailable_for_obsolete_currency() {
        RateProvider provider = createProvider();
        Assertions.assertThatThrownBy(() -> provider.rate("DEM", "EUR"))
                  .isInstanceOf(RateNotAvailable.class);
    }

    @Test
    default void rates_are_always_between_allowed_min_and_max() throws RateNotAvailable {
        assertRateWithinBounds("USD", "EUR");
        assertRateWithinBounds("EUR", "USD");
        assertRateWithinBounds("CAD", "CHF");
    }

    @Test
    default void inverse_rates_make_money() throws RateNotAvailable {
        assertInverseRatesMakeMoney("USD", "EUR");
        assertInverseRatesMakeMoney("CAD", "CHF");
    }
}
```

```
class DatabaseRateProviderTests {
    @Nested
    class Contract_Tests implements RateProviderContractTests {
        @Override
        public RateProvider createProvider() {
            return new DatabaseRateProvider();
        }
    }

    @Nested
    class Functional_Tests {
        @Test void rates_data_is_loaded_on_startup() {…}

        @Test void rates_data_can_be_refreshed() {…}

        @Test void supported_rates_can_be_configured() {…}
    }
}
```

```
class WebCrawlingRateProviderTests {
    @Nested
    class Contract_Tests implements RateProviderContractTests {
        @Override
        public RateProvider createProvider() {
            return new WebCrawlingRateProvider();
        }
    }
}
```

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

Benötigen wir integrierte Tests?

- Collaboration und Contract Tests sind Microtests:
  Sie skalieren besser, laufen schneller
  und sind präziser
- Wir benötigen weiterhin integrierte Tests
  - um technische Komplikationen zu verifizieren,
    z.B. Nebenläufigkeit, Parallelität, Netzwerkverkehr
  - um die Integration mit externen Komponenten zu überprüfen,
    z.B. Datenbanken, Webservices, Devices

Wie können wir Contract Testing vereinfachen?

- Aufwand und Disziplin nötig, um Contracts als Tests zu formulieren
  - Grenzwerte
  - Kombinatorik
  - Manuelle Mocks

Beispiel-basierte Tests

Ein *Beispiel* zeigt, dass unser Code
bei ganz konkreten Eingaben
ein ganz konkretes Ergebnis liefert.

Beispiel-basierte Tests

Ein *Beispiel* zeigt, dass unser Code
bei ganz konkreten Eingaben
ein ganz konkretes Ergebnis liefert.

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

Property-based Testing

Eine *Property* zeigt, dass unser Code
für eine Klasse von Eingaben (Vorbedingung)
bestimmte allgemeine Eigenschaften (Invarianten)
erfüllt.

```
Collections.reverse(List aList):
    // Vorbedingungen?
    // Nachbedingungen und Invarianten?
```

```
Collections.reverse(List aList):
    // Vorbedingungen?
    // Nachbedingungen und Invarianten?
```

```
Collections.reverse(List aList):
   // Vorbedingungen?
   // Nachbedingungen und Invarianten?
```

Vorbedinungen

- Beliebige Liste, die nicht null ist

```
Collections.reverse(List aList):
    // Vorbedingungen?
    // Nachbedingungen und Invarianten?
```

Vorbedinungen

- Beliebige Liste, die nicht null ist

Invariants

- Größe der Liste bleibt gleich
- Alle Elemente bleiben erhalten
- Beim Reversieren wird das erste zum letzten Element
- Doppeltes Reversieren führt zur Ausgangsliste

Eine Property als 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();
}
```

RateProvider: Contract Properties

```
interface RateProviderContractProperties <E extends RateProvider> {

    E createProvider();

    @Property
    default void throws_IllegalArgumentException_for_unknown_currency(
            @ForAll("unknownCurrencies") String unknownCurrency,
            @ForAll("knownCurrencies") String knownCurrency
    ) {
        RateProvider provider = createProvider();
        Assertions.assertThatThrownBy(() -> provider.rate(unknownCurrency, knownCurrency))
                .isInstanceOf(IllegalArgumentException.class);
        Assertions.assertThatThrownBy(() -> provider.rate(knownCurrency, unknownCurrency))
                .isInstanceOf(IllegalArgumentException.class);
    }

    @Provide
    default Arbitrary<String> unknownCurrencies() {
        return Arbitraries.strings().alpha().ofLength(3);
    }

    @Provide
    default Arbitrary<String> knownCurrencies() {
        return Arbitraries.of("USD", "EUR", "CHF", "CAN", "DEM", "FRF");
    }
}
```

```
class DatabaseRateProviderProperties {
    @Group
    class Contract_Properties implements RateProviderContractProperties<DatabaseRateProvider> {
        @Override
        public DatabaseRateProvider createProvider() {
            return new DatabaseRateProvider();
        }
    }
}
```

```java
class DatabaseRateProviderProperties {
    @Group
    class Contract_Properties implements RateProviderContractProperties<DatabaseRateProvider> {
        @Override
        public DatabaseRateProvider createProvider() {
            return new DatabaseRateProvider();
        }
    }
}
```

```
java.lang.AssertionError:
 Expecting:
   <net.johanneslink.eurocalc.RateNotAvailable: Currency [DEM] is unknown>
 to be an instance of:
   <java.lang.IllegalArgumentException>
```

```
class DatabaseRateProviderProperties {
    @Group
    class Contract_Properties implements RateProviderContractProperties<DatabaseRateProvider> {
        @Override
        public DatabaseRateProvider createProvider() {
            return new DatabaseRateProvider();
        }
    }
}
```

```
java.lang.AssertionError:
 Expecting:
   <net.johanneslink.eurocalc.RateNotAvailable: Currency [DEM] is unknown>
 to be an instance of:
   <java.lang.IllegalArgumentException>
```

```
private void checkCurrencyValid(String currency) throws RateNotAvailable {
    if (supportedCurrencies().contains(currency)) {
        return;
    }
    if (obsoleteCurrencies().contains(currency)) {
        throw new RateNotAvailable(currency);
    }
    throw new IllegalArgumentException(currency);
}
```

```java
@Property
default void rates_are_always_between_allowed_min_and_max(
        @ForAll("supportedCurrencies") String currency1,
        @ForAll("supportedCurrencies") String currency2
) throws RateNotAvailable {
    assertRateWithinBounds(currency1, currency2);
}

@Provide
default Arbitrary<String> supportedCurrencies() {
    return Arbitraries.of("USD", "EUR", "CHF", "CAD");
}
```

```java
@Property
default void rates_are_always_between_allowed_min_and_max(
        @ForAll("supportedCurrencies") String currency1,
        @ForAll("supportedCurrencies") String currency2
) throws RateNotAvailable {
    assertRateWithinBounds(currency1, currency2);
}

@Provide
default Arbitrary<String> supportedCurrencies() {
    return Arbitraries.of("USD", "EUR", "CHF", "CAD");
}
```

```
java.lang.NullPointerException: null
sample = ["USD", "USD"]
```

```java
@Property
default void rates_are_always_between_allowed_min_and_max(
        @ForAll("supportedCurrencies") String currency1,
        @ForAll("supportedCurrencies") String currency2
) throws RateNotAvailable {
    Assume.that(!currency1.equals(currency2));
    assertRateWithinBounds(currency1, currency2);
}
```

```java
@Property
default void rates_are_always_between_allowed_min_and_max(
        @ForAll("supportedCurrencies") String currency1,
        @ForAll("supportedCurrencies") String currency2
) throws RateNotAvailable {
    Assume.that(!currency1.equals(currency2));
    assertRateWithinBounds(currency1, currency2);
}
```

```
Contract Properties:rates are always between allowed min and max =
                                |------------------jqwik------------------
tries = 16                      | # of calls to property
checks = 12                     | # of not rejected calls
```

```java
@Property
default void inverse_rates_make_money(
        @ForAll("supportedCurrencies") String currency1,
        @ForAll("supportedCurrencies") String currency2
) throws RateNotAvailable {
    Assume.that(!currency1.equals(currency2));
    assertInverseRatesMakeMoney(currency1, currency2);
}
```

```java
@Property
default void inverse_rates_make_money(
        @ForAll("supportedCurrencies") String currency1,
        @ForAll("supportedCurrencies") String currency2
) throws RateNotAvailable {
    Assume.that(!currency1.equals(currency2));
    assertInverseRatesMakeMoney(currency1, currency2);
}
```

```
Contract Properties:inverse rates make money =
 java.lang.AssertionError:
 Expecting:
  <1.04>
 to be less than:
  <1.0>

                                |-------------------jqwik-------------------
tries = 8                       | # of calls to property
checks = 6                      | # of not rejected calls
sample = ["EUR", "CAD"]
```

EuroConverter: Collaboration Properties

```
@Property
void illegal_currencies_are_not_handed_to_rate_provider(
        @ForAll("illegalCurrencies") String illegalCurrency
) throws RateNotAvailable {
    RateProvider provider = Mockito.mock(RateProvider.class);

    Assertions.assertThatThrownBy(
        () -> new EuroConverter(provider).convert(8.0, illegalCurrency))
        .isInstanceOf(IllegalArgumentException.class);

    Mockito.verify(provider, Mockito.never()).rate(Mockito.anyString(), Mockito.anyString());
}

@Provide
Arbitrary<String> illegalCurrencies() {
    return Arbitraries.strings().alpha()
                      .ofMaxLength(10)
                      .map(String::toUpperCase)
                      .filter(currency -> currency.length() != 3);
}
```

```java
@Property
void can_handle_any_exchange_rate(
        @ForAll @DoubleRange(min = MINIMUM_RATE, max = MAXIMUM_RATE) double exchangeRate,
        @ForAll @DoubleRange(min = 0.01, max = 1_000_000.0) double amount
) {



    RateProvider provider = (fromCurrency, toCurrency) -> exchangeRate;
    double euroAmount = new EuroConverter(provider).convert(amount, "USD");
    Assertions.assertThat(euroAmount).isGreaterThan(0.0);
}
```

```java
@Property
void can_handle_any_exchange_rate(
        @ForAll @DoubleRange(min = MINIMUM_RATE, max = MAXIMUM_RATE) double exchangeRate,
        @ForAll @DoubleRange(min = 0.01, max = 1_000_000.0) double amount
) {

    String classifier = exchangeRate == MINIMUM_RATE ? "min"
                      : exchangeRate == MAXIMUM_RATE ? "max" : "other";
    Statistics.collect(classifier);

    RateProvider provider = (fromCurrency, toCurrency) -> exchangeRate;
    double euroAmount = new EuroConverter(provider).convert(amount, "USD");
    Assertions.assertThat(euroAmount).isGreaterThan(0.0);
}
```

```java
@Property
void can_handle_any_exchange_rate(
        @ForAll @DoubleRange(min = MINIMUM_RATE, max = MAXIMUM_RATE) double exchangeRate,
        @ForAll @DoubleRange(min = 0.01, max = 1_000_000.0) double amount
) {

    String classifier = exchangeRate == MINIMUM_RATE ? "min"
                      : exchangeRate == MAXIMUM_RATE ? "max" : "other";
    Statistics.collect(classifier);

    RateProvider provider = (fromCurrency, toCurrency) -> exchangeRate;
    double euroAmount = new EuroConverter(provider).convert(amount, "USD");
    Assertions.assertThat(euroAmount).isGreaterThan(0.0);
}
```

```
Collaboration Tests:can handle any exchange rate =
                              |------------------jqwik------------------
tries = 1000                  | # of calls to property
checks = 1000                 | # of not rejected calls

statistics for [Collaboration Tests:can handle any exchange rate] =
    other : 98.5 %
    max   : 0.8 %
    min   : 0.7 %
```

Problem:
Vertrag ist in den Tests versteckt

- Externalisieren des Vertrags
  - Über Annotationen am Supplier-Interface
    @NotNull, @Pure etc.
  - Über Contract-Library
    - C4J
    - jqwik-Erweiterung

Externalisierter RateProvider-Contract

```
public class RateProviderSupplierContract implements SupplierContract<RateProvider> {
    @Require
    public boolean rate(
            @ConstrainedBy(CurrencyConstraint.class) String fromCurrency,
            @ConstrainedBy(CurrencyConstraint.class) String toCurrency
    ) {
        return !fromCurrency.equals(toCurrency);
    }

    @Ensure
    public void rate(String fromCurrency, String toCurrency, Result<Double> result) {
        result.onValue(value -> {
            assertThat(value).isGreaterThanOrEqualTo(RateProvider.MINIMUM_RATE);
            assertThat(value).isLessThanOrEqualTo(RateProvider.MAXIMUM_RATE);
        }).onThrowable(throwable -> {
            assertThat(throwable).isInstanceOfAny(
                    RateNotAvailable.class,
                    IllegalArgumentException.class
            );
        });
    }
}
```

Manche Vor- und Nachbedingungen lassen sich durch Constraints und Annotationen ausdrücken

```
class CurrencyConstraint implements Constraint<String> {
   @Override
   public boolean isValid(@NonNull String value) {
      return value.length() == 3;
   }
}
```

```java
interface RateProviderContractProperties<E extends RateProvider> {

    @Property
    default void rate_provider_contract_is_obeyed(
        @ForAll("currencies") String currency1,
        @ForAll("currencies") String currency2,
        @ForAll("rateProvider") @Contract(RateProviderSupplierContract.class) E provider
    ) {
        try {
            provider.rate(currency1, currency2);
        } catch (PreconditionViolation ignored) {
        } catch (IllegalArgumentException | RateNotAvailable ignored) { }
    }

    @Provide
    default Arbitrary<String> currencies() {
        return Arbitraries.oneOf(
            supportedCurrencies(), obsoleteCurrencies(), unknownCurrencies());
    }

    @Provide
    default Arbitrary<RateProvider> rateProvider() {
        return Arbitraries.create(this::createProvider);
    }

    abstract E createProvider();
}
```

Weitere Möglichkeiten

- Zustandsabhängige Verträge können durch State-Machines abgebildet werden
  - Unterstützung in jqwik
- Test-Stubs und Mocks können (teilweise) aus dem Vertrag generiert werden
- Anwendung auf Consumer-Driven Contracts

Zusammenfassung

- Collaboration und Contract Tests sind eine Alternative zu (vielen) integrierten Tests
- Property-based Testing gibt Collaboration und Contract Tests mehr Power
- Externalisierte Verträge helfen dabei, Wiederholungen in den Tests und Properties zu vermeiden

Code:

https://github.com/jlink/contract-testing

Slides:

https://johanneslink.net/downloads/contract-testing.pdf

Sources

- Betrand Meyer: Object Oriented Software Construction
- JB Rainsberger:
  Integrated Tests are a Scam
  https://blog.thecodewhisperer.com/permalink/integrated-tests-are-a-scam