Konzept
Informationssysteme GmbH

Clean Property-based Contract Tests

Dr. Frank Raiser
Konzept Informationssysteme GmbH

Overview

Konzept
Informationssysteme GmbH

- Property-based Testing
  - Predicate Logic, Contracts, Invariants
  - How Property-based Testing works
- Property-based Contracts
  - Definition and Application of contract tests
- Quality Assurance
  - Detect untested contracts

Properties – Predicate Logic

Konzept
Informationssysteme GmbH

- `equals` implementation should be
  - Reflexive, Symmetric, Transitive, Consistent, Null-safe
  - F.ex. Symmetry: $\forall x \forall y. (x \neq null \wedge y \neq null) \rightarrow (Equals(x, y) \leftrightarrow Equals(y, x))$

Predicate logic uses ∧ for logical and (&&)

Equals is a predicate that needs to be satisfied.

```
/**
 * Indicates whether some other object is "equal to" this one.
 * <p>
 * The {@code equals} method implements an equivalence relation
 * on non-null object references:
 * <ul>
 * <li>It is <i>reflexive</i>: for any non-null reference value
 *     {@code x}, {@code x.equals(x)} should return
 *     {@code true}.
 * <li>It is <i>symmetric</i>: for any non-null reference values
 *     {@code x} and {@code y}, {@code x.equals(y)}
 *     should return {@code true} if and only if
 *     {@code y.equals(x)} returns {@code true}.
 * <li>It is <i>transitive</i>: for any non-null reference values
 *     {@code x}, {@code y}, and {@code z}, if
 *     {@code x.equals(y)} returns {@code true} and
 *     {@code y.equals(z)} returns {@code true}, then
 *     {@code x.equals(z)} should return {@code true}.
 * <li>It is <i>consistent</i>: for any non-null reference values
 *     {@code x} and {@code y}, multiple invocations of
 *     {@code x.equals(y)} consistently return {@code true}
 *     or consistently return {@code false}, provided no
 *     information used in {@code equals} comparisons on the
 *     objects is modified.
 * <li>For any non-null reference value {@code x},
 *     {@code x.equals(null)} should return {@code false}.
 * </ul>
 * <p>
 * The {@code equals} method for class {@code Object} implements
 * the most discriminating possible equivalence relation on objects;
 * that is, for any non-null reference values {@code x} and
 * {@code y}, this method returns {@code true} if and only
 * if {@code x} and {@code y} refer to the same object
 * ({@code x == y} has the value {@code true}).
 * <p>
 * Note that it is generally necessary to override the {@code hashCode}
 * method whenever this method is overridden, so as to maintain the
 * general contract for the {@code hashCode} method, which states
 * that equal objects must have equal hash codes.
 *
 * @param   obj   the reference object with which to compare.
 * @return  {@code true} if this object is the same as the obj
 *          argument; {@code false} otherwise.
 * @see     #hashCode()
 * @see     java.util.HashMap
 */
public boolean equals(Object obj) {
    return (this == obj);
}
```

Properties – Contracts

- Interfaces define contracts
- Many prominent examples in Java
  - Comparator#compare, Comparable#compareTo
  - List#contains, List#remove
- No choice – you have to agree to that contract
  - `java.lang.IllegalArgumentException: Comparison method violates its general contract!`

How do you test these contracts?

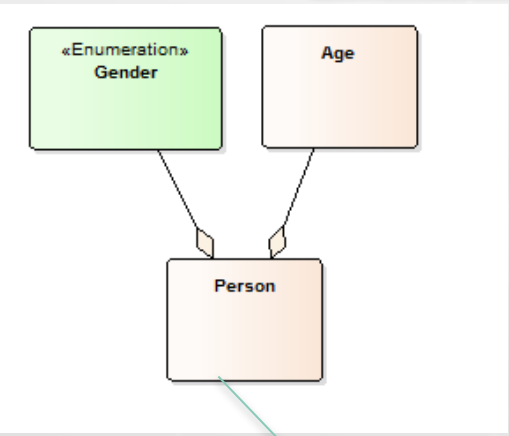
Example Scenario: Premium Calculation

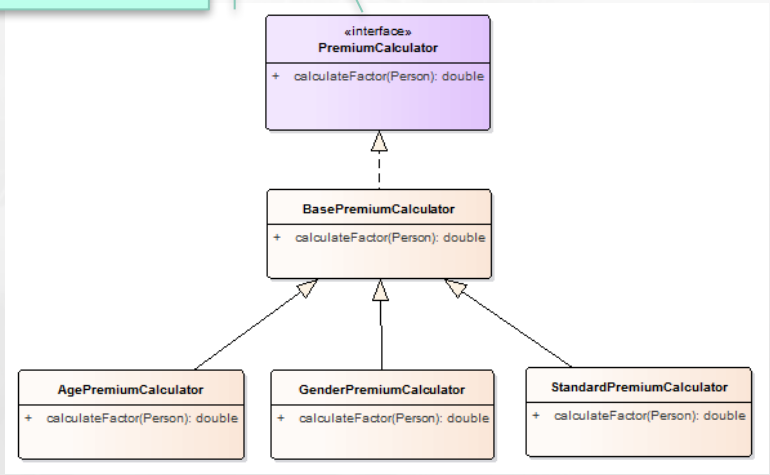
- Calculation of the premium factor for a car insurance
  - Adapted from “Developer Testing” by Alexander Tarlinder [ISBN-13: 978-0-13-429106-2]
- Potential indicators:
  - Younger persons are more likely to have accidents
  - Female drivers are less likely to claim insurance

Example model

Konzept
Informationssysteme GmbH

Interface and different implementations for premium factors





Represents someone who can request an insurance

Standard Unit Testing

Konzept
Informationssysteme GmbH

1. Create a "test" person
2. Run the calculator
3. Verify the correct factor
4. Repeat

AgePremiumCalculator

Factor == 1.75

Factor == 1.0

Factor == 1.35

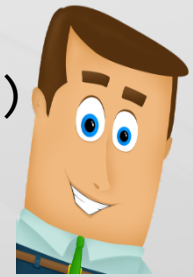
Standard Unit Testing – Limited Sample Space

Konzept
Informationssysteme GmbH

```
new Person(Gender.FEMALE, new Age(19))
```

```
new Person(Gender.FEMALE, new Age(35))
```

```
new Person(Gender.FEMALE, new Age(73))
```



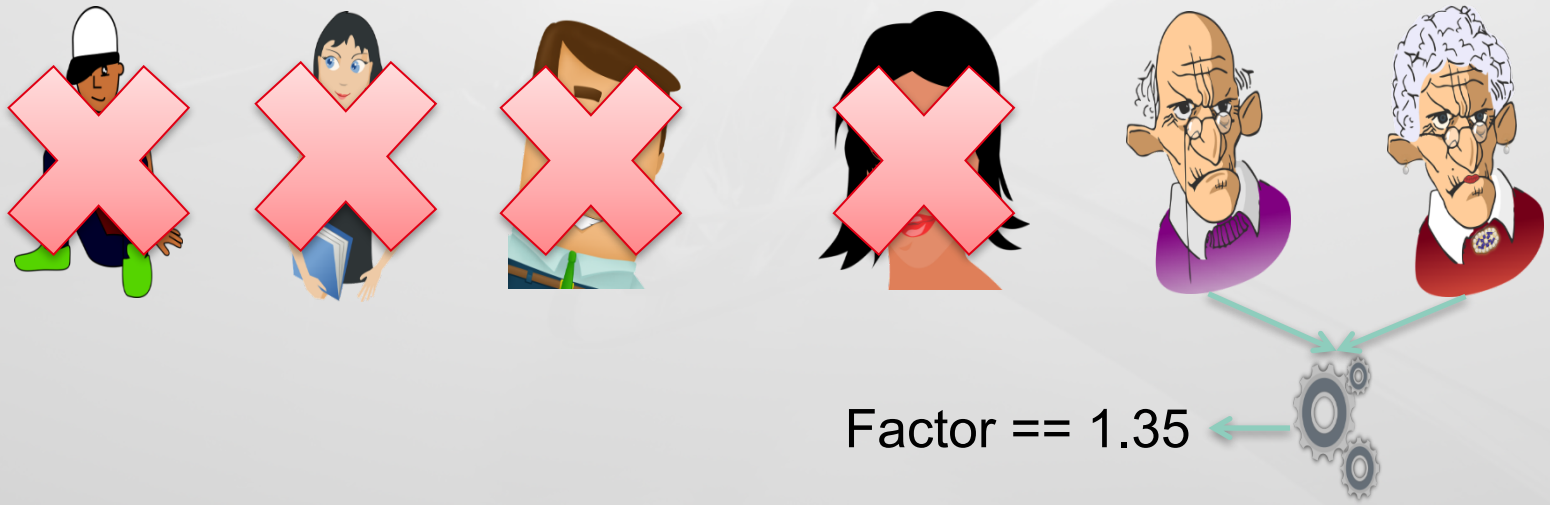
Property-Based Testing with junit-quickcheck

Konzept
Informationssysteme GmbH

- Elderly drivers should get a premium factor of 1.35

$$\forall x.(x\ is\ a\ Person) \wedge (x.age \geq 60) \rightarrow calculateFactor(x) = 1.35$$

```
@Property
public void elderlyDriverFactor(Person driver) {
    assumeThat(driver.getAge().getYears(), greaterThanOrEqualTo( value: 60));
    assertThat(calculator.calculateFactor(driver), is( value: 1.35));
}
```



Generators

Konzept
Informationssysteme GmbH

```
@Property
public void elderlyDriverFactor(Person driver) {
    assumeThat(driver.getAge().getYears(), greaterThanOrEqualTo( value: 60));
    assertThat(calculator.calculateFactor(driver), is( value: 1.35));
}
```

- How to get a `Person` ?
- Generators do that
  - Configurable selection of appropriate generator
  - Configuration of generator possible as well (f.ex. only female persons)

```
public class PersonGenerator extends Generator<Person> {

    public PersonGenerator() {
        super(Person.class);
    }

    public Person generate(
            SourceOfRandomness sourceOfRandomness,
            GenerationStatus generationStatus) {
        return new Person(
                gen().type(Gender.class).generate(
                        sourceOfRandomness, generationStatus),
                gen().type(Age.class).generate(
                        sourceOfRandomness, generationStatus));
    }
}
```

Generators can be straightforward

junit-quickcheck – Background

Konzept
Informationssysteme GmbH

Language	Library	License
C	Theft https://github.com/silentbicycle/theft	ISC License
Clojure	test.check https://github.com/clojure/test.check	Eclipse Public License
Erlang	Triq – Trifork QuickCheck for Erlang https://github.com/krestenkrab/triq	Apache 2.0
Haskell	QuickCheck https://hackage.haskell.org/package/QuickCheck	BSD 3
Java	junit-quickcheck https://github.com/pholser/junit-quickcheck	MIT
Javascript	qc.js https://bitbucket.org/darrint/qc.js	Revised BSD
.NET	FsCheck https://github.com/fscheck/FsCheck	Revised BSD
Python	factcheck https://github.com/npryce/python-factcheck	Apache 2.0
Scala	ScalaCheck http://scalacheck.org/	Revised BSD

Possibilities and Limitations

Konzept
Informationssysteme GmbH

- Many tests with little code
- Better readability
  - Property explicitly formulated
  - Fewer tests needed
  - No object construction code
- Reusable generators

- Random input data
  - Can be reproduced (seed)
  - no guarantee for edge-cases
- Generators needed
  - can also get complex
- Runtime penalty
  - Falsifications tried 100 times

Example: combining calculators

Konzept
Informationssysteme GmbH

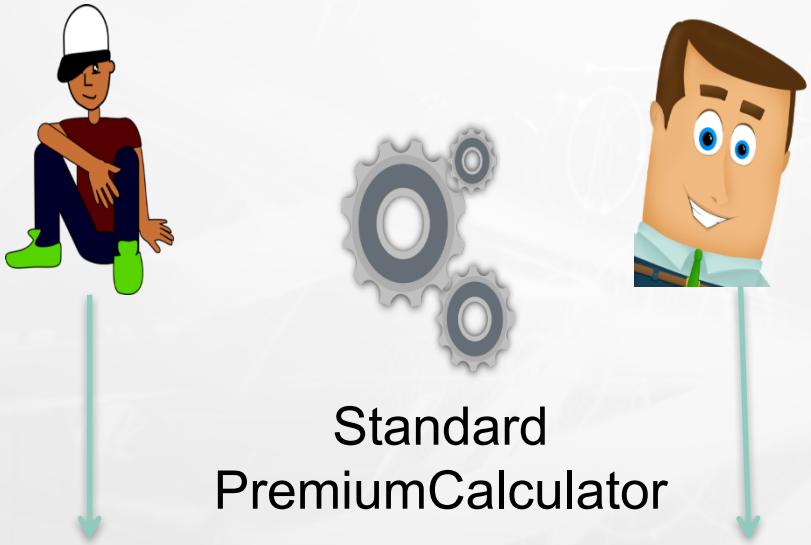
```
public class StandardPremiumCalculator extends BasePremiumCalculator {

    private final List<PremiumCalculator> calculators = Arrays.asList(
            new AgePremiumCalculator(),
            new GenderPremiumCalculator()
    );

    public double calculateFactor(Person person) {
        double combinedFactor = calculators.stream()
                .mapToDouble(calculator -> calculator.calculateFactor(person))
                .reduce( identity: 1.0, (x,y) -> x*y);
        return super.calculateFactor(person) * combinedFactor;
    }
}
```

Product of base class factor and other calculators' factors

Contracts



- Various factors involved, then a new business rule:
  - All factors must be within the range of 0.5 and 2.0
    - Essentially a contract on the `PremiumCalculator` interface

Property-based Contracts

Konzept
Informationssysteme GmbH

```java
public interface PremiumCalculatorContract {

    PremiumCalculator subject();

    @Property
    default void premiumFactorRemainsInValidRange(Person driver) {
        double factor = subject().calculateFactor(driver);
        assertThat(factor, is(both(
                greaterThan( value: 0.5)).and(lessThan( value: 2.0))));
    }
}
```

Codifies contractual agreement for all interface implementations

Clean Property-based Contracts

Konzept
Informationssysteme GmbH

How much effort and code is it to verify the contract for an implementation class?

Explicit adherence to contract

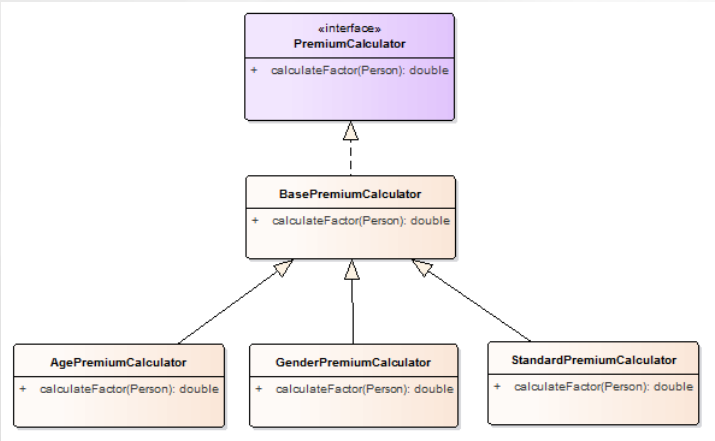
```
public class StandardPremiumCalculatorTest implements PremiumCalculatorContract {

    @Override
    public PremiumCalculator subject() { return new StandardPremiumCalculator(); }
```

Provides test subject

Sufficient for 100 test cases against the contract!

Liskov Substitution Principle



- `BasePremiumCalculator` should satisfy the contract
- LSP requires derived classes to adhere to contract
  - Can be easily tested now
  - **But:** What if a developer forgot to add the contract test?

jQAssistant

Konzept
Informationssysteme GmbH

Define contracts

Validate contract adherence

jQAssistant is a QA tool which allows the definition and validation of project specific rules on a structural level. [Source: jqAssistant docs]

Define LSP rules

Taking quality assurance even further

jqassistant

Jenkins

any Continuous Integration system

Analyze source

Detect contracts

Detect classes for contracts

Ensure Test classes exist

Ensure contract is implemented

Execute tests to verify all contracts for all implementations

Konzept
Informationssysteme GmbH

Contract Definition

Konzept
Informationssysteme GmbH

Neo4j's Cypher query syntax

```
MATCH
  (i:Interface)-[:DEPENDS_ON]->(t:Type),
  (i:Interface)-[:DEPENDS_ON]->(s:Type)
WHERE
  i.name =~ ".*Contract" AND
  t.fqn = "com.pholser.junit.quickcheck.Property" AND
  i.name = s.name + "Contract"
WITH
  i :Contract
CREATE
  (i)-[:APPLIES_TO]->(s)
RETURN
  i, s
```

Interface must end in "Contract" and have some @Property

Mark as a contract

Create relationship to the type the contract applies to

* The precise rules may need to be tweaked in project-specific ways (f.ex. If you want a base class for contracts instead of interface w/ default methods)

Contract Adherence Constraint

Konzept
Informationssysteme GmbH

Query types for which contract applies and their test classes

```
MATCH
 (t:Type)-[:DEPENDS_ON]->(i:Interface)<-[:APPLIES_TO]-(c:Contract),
 (test:Class)-[:DEPENDS_ON]->(t)
WHERE
 test.name =~ ".*Test" AND
 NOT (test)-[:IMPLEMENTS]->(c)
RETURN
 test,c
```

Make sure test class adheres to the contract

Contract Testing

Does not implement `PremiumCalculatorConcept`

```
public class AgePremiumCalculatorTest {
```

#	Constraint Name	Count	Severity	Duration (in ms)
1	contract:TestWithoutContract ✗	1	minor	325

Verifies that a test class for a contract-relevant class does implement the contract correctly.

test	c
impl.AgePremiumCalculatorTest	api.PremiumCalculatorContract

Summary

Konzept
Informationssysteme GmbH

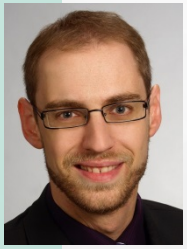
- Explicit contract definitions
- Tiny amount of code
- Reusable generators
- Safety-net against accidentally ignoring contracts

- Effort to create generators
- Large amount of tests being executed
- Quality (dependent on generators and properties) may be misleading

Clean Property-based Contract Tests

Konzept
Informationssysteme GmbH

Konzept Informationssysteme GmbH
Pfarrer-Weiss-Weg 12
89073 Ulm



Dr. Frank Raiser
Software Entwicklungsingenieur
Tel.: +49 731 1403434 – 51
Fax.: +49 731 1403434 – 34
frank.raiser@konzept-is.de
www.konzept-is.de

Thank you

Special thanks to pholser for extending junit-quickcheck to make this work

Paul Holser

[All images Public Domain.]