AVATAR

Ryszard Makuch

Mutation Testing 101

CAMERA

```
data class Event(
    val timestamp: OffsetDateTime,
    val firstProperty: String,
    val secondProperty: String,
    val thirdProperty: String
)

class EventEmitter(private val clock: Clock) {

    fun emit(): Mono<Event> = Mono.just(
        Event(
            timestamp = OffsetDateTime.now(clock),
            firstProperty = "First",
            secondProperty = "Second",
            thirdProperty = "Third"
        )
    )

}
```

```
class EventEmitter(private val clock: Clock) {

    fun emit(): Mono<Event> = Mono.just(
        Event(
            timestamp = OffsetDateTime.now(clock),
            firstProperty = "First",
            secondProperty = "Second",
            thirdProperty = "Third"
        )
    )

}
```

```
class EventEmitterSpec extends Specification implements
StepVerifierRunner {

    Instant fixedInstant = parse("2021-09-16T10:00:00.00Z")
    Clock fixedClock = fixed(fixedInstant, ZoneId.of("UTC"))

    def "an event emitter emits an event"() {
        given:
            EventEmitter sut = new EventEmitter(fixedClock)

        when:
            Mono<Event> emittedEvent = run(sut.emit())

        then:
            expect(emittedEvent, { it ->
                it.timestamp == fixedInstant.atOffset(ZoneOffset.UTC)
                it.firstProperty == "First"
                it.secondProperty == "Second"
                it.thirdProperty  == "Third"
            })
    }

}
```

```
trait StepVerifierRunner {

    def <T> StepVerifier.FirstStep<T> run(Publisher<T> publisher) {
        StepVerifier.create(publisher)
    }

    def <T> void expect(StepVerifier.FirstStep<T> input,
                        Predicate<T> predicate) {
        input
            .expectNextMatches { predicate.test(it) }
            .verifyComplete()
    }

}

class EventEmitter(private val clock: Clock) {

    fun emit(): Mono<Event> = Mono.just(
        Event(
            timestamp = OffsetDateTime.now(clock),
            firstProperty = "First",
            secondProperty = "Second",
            thirdProperty = "Third"
        )
    )

}
```

```
class EventEmitterSpec extends Specification implements
StepVerifierRunner {

    Instant fixedInstant = parse("2021-09-16T10:00:00.00Z")
    Clock fixedClock = fixed(fixedInstant, ZoneId.of("UTC"))

    def "an event emitter emits an event"() {
        given:
            EventEmitter sut = new EventEmitter(fixedClock)

        when:
            Mono<Event> emittedEvent = run(sut.emit())

        then:
            expect(emittedEvent, { it ->
                it.timestamp == fixedInstant.atOffset(ZoneOffset.UTC)
                it.firstProperty == "First"
                it.secondProperty == "Second"
                it.thirdProperty  == "Third"
            })
    }

}
```

Coverage: EventEmitterSpec

100% classes, 100% lines covered in package

Element	Class, %	Method, %	Line, %
Event	100% (1/1)	100% (1/1)	100% (5/5)
EventEmitter	100% (1/1)	100% (2/2)	100% (8/8)

```
class EventEmitterSpec extends Specification implements
StepVerifierRunner {

    Instant fixedInstant = parse("2021-09-16T10:00:00.00Z")
    Clock fixedClock = fixed(fixedInstant, ZoneId.of("UTC"))

    def "an event emitter emits an event"() {
        given:
            EventEmitter sut = new EventEmitter(fixedClock)

        when:
            Mono<Event> emittedEvent = run(sut.emit())

        then:
            expect(emittedEvent, { it ->
                it.timestamp == fixedInstant.atOffset(ZoneOffset.UTC)
                it.firstProperty == "First"
                it.secondProperty == "Second"
                it.thirdProperty  == "Third"
            })
    }

}
```

```
class EventEmitter(private val clock: Clock) {

    fun emit(): Mono<Event> = Mono.just(
        Event(
            timestamp = OffsetDateTime.now(clock),
            firstProperty = "First",
            secondProperty = "Second",
            thirdProperty = "Third"
        )
    )

}
```

```
class EventEmitterSpec extends Specification implements
StepVerifierRunner {

    Instant fixedInstant = parse("2021-09-16T10:00:00.00Z")
    Clock fixedClock = fixed(fixedInstant, ZoneId.of("UTC"))

    def "an event emitter emits an event"() {
        given:
            EventEmitter sut = new EventEmitter(fixedClock)

        when:
            Mono<Event> emittedEvent = run(sut.emit())

        then:
            expect(emittedEvent, { it ->
                it.timestamp == fixedInstant.atOffset(ZoneOffset.UTC)
                it.firstProperty == "First"
                it.secondProperty == "Second"
                it.thirdProperty  == "Third"
            })
    }

}
```

```
class EventEmitter(private val clock: Clock) {

    fun emit(): Mono<Event> = Mono.just(
        Event(
            timestamp = OffsetDateTime.now(clock),
            firstProperty = "",
            secondProperty = "Second",
            thirdProperty = "Third"
        )
    )

}
```

```
class EventEmitterSpec extends Specification implements
StepVerifierRunner {

    Instant fixedInstant = parse("2021-09-16T10:00:00.00Z")
    Clock fixedClock = fixed(fixedInstant, ZoneId.of("UTC"))

    def "an event emitter emits an event"() {
        given:
            EventEmitter sut = new EventEmitter(fixedClock)

        when:
            Mono<Event> emittedEvent = run(sut.emit())

        then:
            expect(emittedEvent, { it ->
                it.timestamp == fixedInstant.atOffset(ZoneOffset.UTC)
                it.firstProperty == "First"
                it.secondProperty == "Second"
                it.thirdProperty  == "Third"
            })
    }

}
```

```
class EventEmitter(private val clock: Clock) {

    fun emit(): Mono<Event> = Mono.just(
        Event(
            timestamp = OffsetDateTime.now(clock),
            firstProperty = "",
            secondProperty = "Second",
            thirdProperty = "Third"
        )
    )

}
```

```
class EventEmitterSpec extends Specification implements
StepVerifierRunner {

    Instant fixedInstant = parse("2021-09-16T10:00:00.00Z")
    Clock fixedClock = fixed(fixedInstant, ZoneId.of("UTC"))

    def "an event emitter emits an event"() {
        given:
            EventEmitter sut = new EventEmitter(fixedClock)

        when:
            Mono<Event> emittedEvent = run(sut.emit())

        then:
            expect(emittedEvent, { it ->
                it.timestamp == fixedInstant.atOffset(ZoneOffset.UTC)
                it.firstProperty == "First"
                it.secondProperty == "Second"
                it.thirdProperty == "Third"
            })
    }

}
```

```
class EventEmitter(private val clock: Clock) {

    fun emit(): Mono<Event> = Mono.just(
        Event(
            timestamp = OffsetDateTime.now(clock),
            firstProperty = "",
            secondProperty = "Second",
            thirdProperty = "Third"
        )
    )

}
```

```
class EventEmitterSpec extends Specification implements
StepVerifierRunner {

    Instant fixedInstant = parse("2021-09-16T10:00:00.00Z")
    Clock fixedClock = fixed(fixedInstant, ZoneId.of("UTC"))

    def "an event emitter emits an event"() {
        given:
            EventEmitter sut = new EventEmitter(fixedClock)

        when:
            Mono<Event> emittedEvent = run(sut.emit())

        then:
            expect(emittedEvent, { it ->
                it.timestamp == fixedInstant.atOffset(ZoneOffset.UTC)
                it.firstProperty == "First"
                it.secondProperty == "Second"
                it.thirdProperty == "Third"
            })
    }

}
```

```
class EventEmitter(private val clock: Clock) {

    fun emit(): Mono<Event> = Mono.just(
        Event(
            timestamp = OffsetDateTime.now(clock),
            firstProperty = "",
            secondProperty = "Second",
            thirdProperty = "Third"
        )
    )

}
```

```
class EventEmitterSpec extends Specification implements
StepVerifierRunner {

    Instant fixedInstant = parse("2021-09-16T10:00:00.00Z")
    Clock fixedClock = fixed(fixedInstant, ZoneId.of("UTC"))

    def "an event emitter emits an event"() {
        given:
            EventEmitter sut = new EventEmitter(fixedClock)

        when:
            Mono<Event> emittedEvent = run(sut.emit())

        then:
            expect(emittedEvent, { it ->
                it.timestamp == fixedInstant.atOffset(ZoneOffset.UTC)
                it.firstProperty == "First"
                it.secondProperty == "Second"
                it.thirdProperty == "Third"
            })
    }

}
```

```kotlin
class EventEmitter(private val clock: Clock) {

    fun emit(): Mono<Event> = Mono.just(
        Event(
            timestamp = OffsetDateTime.now(clock),
            firstProperty = "",
            secondProperty = "Second",
            thirdProperty = "Third"
        )
    )

}
```

```kotlin
class EventEmitterSpec extends Specification implements
StepVerifierRunner {

    Instant fixedInstant = parse("2021-09-16T10:00:00.00Z")
    Clock fixedClock = fixed(fixedInstant, ZoneId.of("UTC"))

    def "an event emitter emits an event"() {
        given:
            EventEmitter sut = new EventEmitter(fixedClock)

        when:
            Mono<Event> emittedEvent = run(sut.emit())

        then:
            expect(emittedEvent, { it ->
                it.timestamp == fixedInstant.atOffset(ZoneOffset.UTC)
                it.firstProperty == "First"
                it.secondProperty == "Second"
                return it.thirdProperty  == "Third"
            })
    }

}
```

```kotlin
class EventEmitter(private val clock: Clock) {

    fun emit(): Mono<Event> = Mono.just(
        Event(
            timestamp = OffsetDateTime.now(clock),
            firstProperty = "",
            secondProperty = "Second",
            thirdProperty = "Third"
        )
    )

}
```

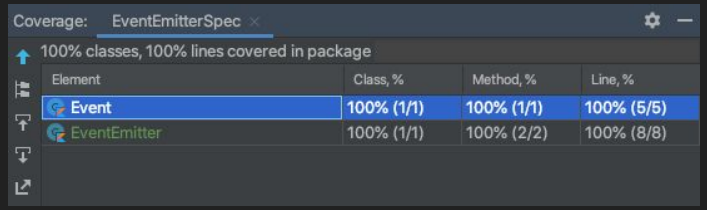
```kotlin
class EventEmitterSpec extends Specification implements
StepVerifierRunner {

    Instant fixedInstant = parse("2021-09-16T10:00:00.00Z")
    Clock fixedClock = fixed(fixedInstant, ZoneId.of("UTC"))

    def "an event emitter emits an event"() {
        given:
            EventEmitter sut = new EventEmitter(fixedClock)

        when:
            Mono<Event> emittedEvent = run(sut.emit())

        then:
            expect(emittedEvent, { it ->
                return it.timestamp == fixedInstant.atOffset(ZoneOffset.UTC) &&
                it.firstProperty == "First" &&
                it.secondProperty == "Second" &&
                it.thirdProperty  == "Third"
            })
    }

}
```

```
class EventEmitter(private val clock: Clock) {

    fun emit(): Mono<Event> = Mono.just(
        Event(
            timestamp = OffsetDateTime.now(clock),
            firstProperty = "First",
            secondProperty = "Second",
            thirdProperty = "Third"
        )
    )

}
```

```
class EventEmitterSpec extends Specification implements
StepVerifierRunner {

    Instant fixedInstant = parse("2021-09-16T10:00:00.00Z")
    Clock fixedClock = fixed(fixedInstant, ZoneId.of("UTC"))

    def "an event emitter emits an event"() {
        given:
            EventEmitter sut = new EventEmitter(fixedClock)

        when:
            Mono<Event> emittedEvent = run(sut.emit())

        then:
            expect(emittedEvent, { it ->
                it.timestamp == fixedInstant.atOffset(ZoneOffset.UTC) &&
                it.firstProperty == "First" &&
                it.secondProperty == "Second" &&
                it.thirdProperty  == "Third"
            })
    }

}
```

```
class EventEmitter(private val clock: Clock) {

    fun emit(): Mono<Event> = Mono.just(
        Event(
            timestamp = OffsetDateTime.now(clock),
            firstProperty = "First",
            secondProperty = "Second",
            thirdProperty = "Third"
        )
    )

}
```

```
class EventEmitterSpec extends Specification implements
StepVerifierRunner {

    Instant fixedInstant = parse("2021-09-16T10:00:00.00Z")
    Clock fixedClock = fixed(fixedInstant, ZoneId.of("UTC"))

    def "an event emitter emits an event"() {
        given:
            EventEmitter sut = new EventEmitter(fixedClock)

        when:
            Mono<Event> emittedEvent = run(sut.emit())

        then:
            expect(emittedEvent, { it ->
                it.timestamp == fixedInstant.atOffset(ZoneOffset.UTC) &&
                it.firstProperty == "First" &&
                it.secondProperty == "Second" &&
                it.thirdProperty  == "Third"
            })
    }

}
```



"(...) However, coverage alone might be misleading, as in many cases **where statements are covered but their consequences not asserted upon** [15]. (...)"

State of Mutation Testing at Google (2018) - Goran Petrović, Marko Invanković

[15] *Subsumption of Condition Coverage Techniques by Mutation Testing* (1996) - A. Jefferson Offutt, Jeffrey M. Voas

pitest.org

/src/integration/...
/src/test/...

1010|0000|0000

/src/main/...

1010|0000|0000

Mutation operators:

- 1010 → 0101
- 0000 → 1111

1010|0000|0000

Mutation operators:

- 1010 → 0101
- 0000 → 1111

1010|0000|0000

Mutation operators:

- 1010 → 0101
- 0000 → 1111

0101|0000|0000

Mutation operators:

- 1010 → 0101
- 0000 → 1111

1010|0000|0000

Mutation operators:

- 1010 → 0101
- 0000 → 1111

1010 | 0000 | 0000

Mutation operators:

- 1010 → 0101
- 0000 → 1111

1010 | 1111 | 0000

Mutation operators:

- 1010 → 0101
- 0000 → 1111

1010|0000|0000

Mutation operators:

- 1010 → 0101
- 0000 → 1111

1010|0000|0000

Mutation operators:

- 1010 → 0101
- 0000 → 1111

1010|0000|1111

Mutation operators:

- 1010 → 0101
- 0000 → 1111

0000 | 1111 | 1111

Mutation operators:

- 1010 → 0101
- 0000 → 1111

NO_COVERAGE

SURVIVED

KILLED
TIMED_OUT

0000 | 1111 | 1111

Mutation operators:

- 1010 → 0101
- 0000 → 1111

NO_COVERAGE

SURVIVED

KILLED
TIMED_OUT

0000 | 1111 | 1111

Mutation Coverage = (KILLED + TIMED_OUT) / (SURVIVED + KILLED + TIMED_OUT + NO_COVERAGE) = 1 / 3 = 0.33

Test Strength = (KILLED + TIMED_OUT) / (SURVIVED + KILLED + TIMED_OUT) = 1 / 2 = 0.5

pitest.org

Only **3 changes needed** to start mutation-testing our open-source **Hermes** (https://gallegroithub.com//hermes)

https://gradle-pitest-plugin.solidsoft.info/

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        /**
         * Loading Nexus Publish Plugin from local directory as it doesn't have a stab
         * The plugin publishes packages to Nexus and solves common problems related w
         * such us "split staging repositories".
         * More info about the plugin: https://github.com/gradle-nexus/publish-plugin
         * Issue which it solves: https://groups.google.com/u/0/a/glists.sonatype.com/
         */
        classpath files('./libs/io.github.gradle-nexus.publish-plugin-0.1.0-20200530.1
    }
}
```

```
3   3   buildscript {
4   4       repositories {
5   5           jcenter()
6   6       }
7   7       dependencies {
8   8           /**
9   9            * Loading Nexus Publish Plugin from local directory as it doesn't have a stab
10  10           * The plugin publishes packages to Nexus and solves common problems related w
11  11           * such us "split staging repositories".
12  12           * More info about the plugin: https://github.com/gradle-nexus/publish-plugin
13  13           * Issue which it solves: https://groups.google.com/u/0/a/glists.sonatype.com/
14  14           */
15  15           classpath files('./libs/io.github.gradle-nexus.publish-plugin-0.1.0-20200530.1
16  16           classpath 'info.solidsoft.gradle.pitest:gradle-pitest-plugin:1.6.0'
17  17       }
```

```
subprojects {
    configurations {
        all*.exclude group: "org.slf4j", module: "slf4j-log4j12"
        all*.exclude group: "log4j", module: "log4j"
```

```
160 161 subprojects {
161 162     apply plugin: 'info.solidsoft.pitest'
162 163
163 164     configurations {
```

```
    }

    test {
        reports {
            html.enabled = false
            junitXml.enabled = true
            junitXml.destination = file("$buildDir/test-results/$name")
        }

        testLogging {
```

```
173 174     tasks.withType(JavaCompile) {
174 175         options.compilerArgs << "-Xlint:unchecked,deprecation"
175 176     }
176 177
177 178     pitest {
178 179         threads = 8
179 180         failWhenNoMutations = false
180 181     }
181 182
182 183     test {
```

```
./gradlew pitest
```

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
10	87%	222/255	78%	89/114	92%	89/97

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
pl.allegro.tech.hermes.mock	9	87%	219/251	78%	86/110	91%	86/94
pl.allegro.tech.hermes.mock.matching	1	75%	3/4	75%	3/4	100%	3/3

Report generated by PIT 1.6.3

Mutation Coverage = (KILLED + TIMED_OUT) / (SURVIVED + KILLED + TIMED_OUT + NO_COVERAGE)

Test Strength = (KILLED + TIMED_OUT) / (SURVIVED + KILLED + TIMED_OUT)

StartsWithPattern.java

```
1   package pl.allegro.tech.hermes.mock.matching;
2
3   import com.fasterxml.jackson.annotation.JsonProperty;
4   import com.github.tomakehurst.wiremock.matching.MatchResult;
5   import com.github.tomakehurst.wiremock.matching.StringValuePattern;
6
7   public class StartsWithPattern extends StringValuePattern {
8
9           public StartsWithPattern(@JsonProperty("startsWith") String expectedValue) {
10              super(expectedValue);
11          }
12
13          public String getStartsWith() {
14 1            return expectedValue;
15          }
16
17          @Override
18          public MatchResult match(String value) {
19 3            return MatchResult.of(value != null && value.startsWith(expectedValue));
20          }
21  }
```

Mutations

```
14  1. replaced return value with "" for pl/allegro/tech/hermes/mock/matching/StartsWithPattern::getStartsWith → NO_COVERAGE
    1. negated conditional → KILLED
19  2. negated conditional → KILLED
    3. replaced return value with null for pl/allegro/tech/hermes/mock/matching/StartsWithPattern::match → TIMED_OUT
```

Active mutators

- BOOLEAN_FALSE_RETURN
- BOOLEAN_TRUE_RETURN
- CONDITIONALS_BOUNDARY_MUTATOR
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- NULL_RETURN_VALUES
- PRIMITIVE_RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

Tests examined

- pl.allegro.tech.hermes.mock.HermesMockRuleTest (141 ms)
- pl.allegro.tech.hermes.mock.HermesMockJsonTest (771 ms)
- pl.allegro.tech.hermes.mock.HermesMockTest (6009 ms)
- pl.allegro.tech.hermes.mock.HermesMockAvroTest (2733 ms)

	:test	:pitest	:pitest / :test	mutacje
hermes-management	0:03:46.185	0:34:12.000	9.07	1644
hermes-consumers	0:00:42.698	0:06:21.000	8.92	2959
hermes-frontend	0:00:22.773	0:03:22.000	8.87	807
hermes-tracker-elasticsearch	0:01:12.950	0:02:35.000	2.12	71
hermes-mock	0:00:14.246	0:02:11.000	9.20	114
hermes-common	0:00:17.444	0:01:20.000	4.59	1251
hermes-client	0:00:07.302	0:01:11.000	9.72	283
hermes-schema	0:00:04.508	0:00:28.000	6.21	216
hermes-api	0:00:02.944	0:00:13.000	4.42	929
hermes-test-helper	0:00:02.224	0:00:07.000	3.15	496
hermes-tracker-mongo	0:00:00.086	0:00:02.000	23.26	16
hermes-metrics	0:00:01.331	0:00:02.000	1.50	38
hermes-tracker	0:00:01.091	0:00:01.000	0.92	40
	0:06:56	**0:52:05**	**7.52**	

Incremental analysis comes to the rescue:
https://pitest.org/quickstart/incremental_analysis/

In a brief summary: based on historical data, create mutations only when it makes sense.

Incremental analysis comes to the rescue:
https://pitest.org/quickstart/incremental_analysis/

In a brief summary: based on historical data, create mutations only when it makes sense.

Pitest can be plugged into the CI/CD process with Github Actions:
https://github.com/GroupCDG-Labs/pitest-github-demo/pull/1/files

Incremental analysis comes to the rescue:
https://pitest.org/quickstart/incremental_analysis/

In a brief summary: based on historical data, create mutations only when it makes sense.

Pitest can be plugged into the CI/CD process with Github Actions:
https://github.com/GroupCDG-Labs/pitest-github-demo/pull/1/files

Or can be used as well as part of the static code analysis with SonarQube:
https://github.com/VinodAnandan/sonar-pitest

Just tell me the truth.
Does anyone use mutation testing in **real projects**?

Implement tests verifying that async internal dispatchers shutdown after processing all elements #552

Open with

Merged pivovarit merged 1 commit into `master` from `improving-tests` on 14 Nov 2020

Conversation 0 | Commits 1 | Checks 4 | Files changed 2 | +69 −64

pivovarit commented on 14 Nov 2020 • edited — Owner

Despite working fine, PIT reported that a crucial part of the implementation is not tested correctly due to JUnit calling `System.exit()` which results in an undesired slaughter of all non-daemon threads. This makes tests pass even when the internal dispatcher is still waiting for a poison pill.

Before:

```
65      void stop() {
66          try {
67 1            workingQueue.put(POISON_PILL);
68          } catch (InterruptedException e) {
69              completionSignaller.completeExceptionally(e);
70          } finally {
71 1            dispatcher.shutdown();
72          }
73      }
74
```

After:

```
    void stop() {
        try {
1           workingQueue.put(POISON_PILL);
        } catch (InterruptedException e) {
            completionSignaller.completeExceptionally(e);
        } finally {
1           dispatcher.shutdown();
        }
    }
```

2

pivovarit force-pushed the `improving-tests` branch from **1dd8947** to **ba4dde0** 11 months ago

Reviewers
No reviews

Assignees
pivovarit

Labels
internal

Projects
None yet

Milestone
No milestone

Linked issues
Successfully merging this pull request may close these issues.
None yet

Notifications Customize
Subscribe
You're not receiving notifications from this thread.

1 participant

https://github.com/pivovarit/**parallel-collectors**/pull/552

"(...) User feedback is gathered via Critique where each surfaced code findings displays "Please fix" and "Not useful" links. **75% of all findings with feedback were found useful by developers**. (...)"

State of Mutation Testing at Google (2018) - Goran Petrović, Marko Invanković

Is it possible to detect **real bugs** using mutation testing?

"RQ3 Fault coupling. **Are reported mutants coupled with real software faults?** Can tests written based on mutants improve test effectiveness for real software faults?"

"(...) We found that for **1043 (70%) of the bugs, mutation testing would have reported a fault-coupled mutant in the bug-introducing change**. **Recall that each bug-introducing change was covered by the existing tests**, suggesting that code coverage had exhausted its usefulness. (...)"

Does mutation testing improve testing practices? (2021) - Goran Petrović, Marko Ivanković, Gordon Fraser, René Just


atm
ALLEGRO
TECH
MEETING
14
Thank you!