Developer Survey:
Automated test + mock generation with **RICK**

Deepika Tiwari, Martin Monperrus, Benoit Baudry

Agenda

- Introduction

- Tests generated by **RICK**

- Discussion

Introduction: Mocking

```java
public class ReservationCentre {

    . . .

    public ReservationStatus reserve(int qty, PaymentService paymentService) {

        if (paymentService.checkActiveConnections() > 0) {
            boolean isPaymentSuccessful = paymentService.processPayment(qty);
            . . .
            return ReservationStatus.SUCCESS;
        }

        return ReservationStatus.FAIL;
    }


}
```

```java
public class ReservationCentre {

    . . .

  public ReservationStatus reserve(int qty, PaymentService paymentService) {

     if (paymentService.checkActiveConnections() > 0) {
       boolean isPaymentSuccessful = paymentService.processPayment(qty);

       . . .
       return ReservationStatus.SUCCESS;
     }

     return ReservationStatus.FAIL;
  }

}
```

```java
public class ReservationCentre {

    . . .

    public ReservationStatus reserve(int qty, PaymentService paymentService) {

        if (paymentService.checkActiveConnections() > 0) {
            boolean isPaymentSuccessful = paymentService.processPayment(qty);

            . . .
            return ReservationStatus.SUCCESS;
        }

        return ReservationStatus.FAIL;
    }

}
```

```java
public class ReservationCentre {

    . . .

    public ReservationStatus reserve(int qty, PaymentService paymentService) {

        if (paymentService. checkActiveConnections() > 0) {
            boolean isPaymentSuccessful = paymentService. processPayment(qty);
            . . .
            return ReservationStatus.SUCCESS;
        }

        return ReservationStatus.FAIL;
    }

}
```

MUT

mockable

```java
@Test
public void testThatReservationIsMade() {

    // Arrange
    ReservationCentre centre = new ReservationCentre();

    PaymentService mockPaymentService = mock(PaymentService.class);
    when(mockPaymentService. checkActiveConnections ()).thenReturn(3);
    when(mockPaymentService. processPayment (42.42)).thenReturn(true);

    // Act
    ReservationStatus status = centre. reserve (6, mockPaymentService);

    // Assert
    assertEquals(ReservationStatus.SUCCESS, status);
    verify(mockPaymentService, times(1)).checkActiveConnections();
    verify(mockPaymentService, times(1)).processPayment(anyDouble());

}
```

```java
@Test
public void testThatReservationIsMade() {

    // Arrange
    ReservationCentre centre = new ReservationCentre();

    PaymentService mockPaymentService = mock(PaymentService.class);
    when(mockPaymentService.checkActiveConnections()).thenReturn(3);
    when(mockPaymentService.processPayment(42.42)).thenReturn(true);

    // Act
    ReservationStatus status = centre.reserve(6, mockPaymentService);

    // Assert
    assertEquals(ReservationStatus.SUCCESS, status);
    verify(mockPaymentService, times(1)).checkActiveConnections();
    verify(mockPaymentService, times(1)).processPayment(anyDouble());

}
```

MOCK

```java
@Test
public void testThatReservationIsMade() {

    // Arrange
    ReservationCentre centre = new ReservationCentre();

    PaymentService mockPaymentService = mock(PaymentService.class);
    when(mockPaymentService.checkActiveConnections()).thenReturn(3);
    when(mockPaymentService.processPayment(42.42)).thenReturn(true);

    // Act
    ReservationStatus status = centre.reserve(6, mockPaymentService);

    // Assert
    assertEquals(ReservationStatus.SUCCESS, status);
    verify(mockPaymentService, times(1)).checkActiveConnections();
    verify(mockPaymentService, times(1)).processPayment(anyDouble());

}
```

STUB

```java
@Test
public void testThatReservationIsMade() {

  // Arrange
  ReservationCentre centre = new ReservationCentre();

  PaymentService mockPaymentService = mock(PaymentService.class);
  when(mockPaymentService.checkActiveConnections()).thenReturn(3);
  when(mockPaymentService.processPayment(42.42)).thenReturn(true);

  // Act
  ReservationStatus status = centre.reserve(6, mockPaymentService);

  // Assert
  assertEquals(ReservationStatus.SUCCESS, status);
  verify(mockPaymentService, times(1)).checkActiveConnections();
  verify(mockPaymentService, times(1)).processPayment(anyDouble());

}
```

VERIFY

```java
@Test
public void testThatReservationIsMade() {

    // Arrange
    ReservationCentre centre = new ReservationCentre();

    PaymentService mockPaymentService = mock(PaymentService.class);
    when(mockPaymentService.checkActiveConnections()).thenReturn(3);
    when(mockPaymentService.processPayment(42.42)).thenReturn(true);

    // Act
    ReservationStatus status = centre.reserve(6, mockPaymentService);

    // Assert
    assertEquals(ReservationStatus.SUCCESS, status);
    verify(mockPaymentService, times(1)).checkActiveConnections();
    verify(mockPaymentService, times(1)).processPayment(anyDouble());

}
```
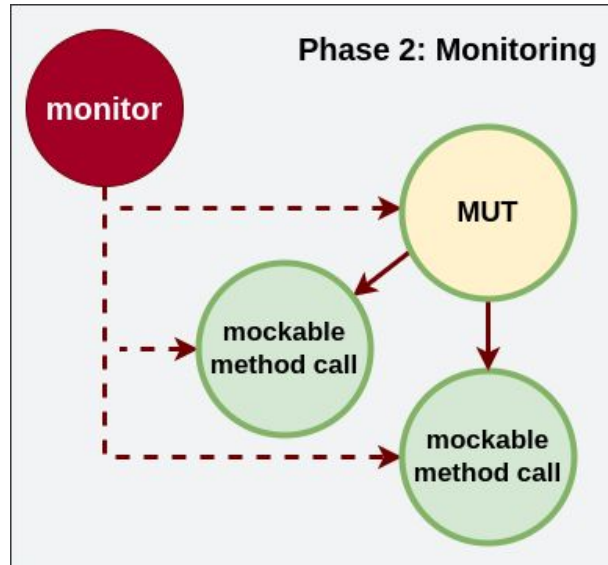
MOCK

STUB

VERIFY

Introduction: RICK

Key phases

Phase 1: Identification
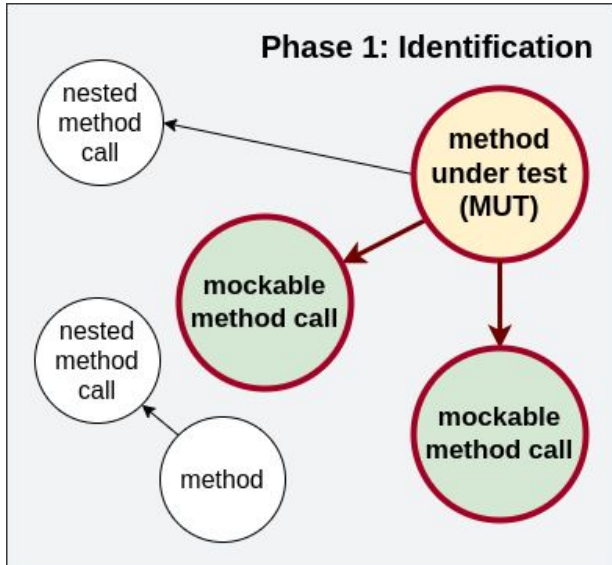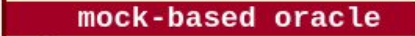
Key phases

Key phases

Phase 3: Generation

```
@Test
void testForMUT() {
  ...
```

mocked method call

mocked method call

`mock-based oracle`

```
}
```

Key phases

Phase 3: Generation

```
@Test
void testForMUT() {
  ...
      mocked
      method call
            mocked
            method call
  mock-based oracle
}
```
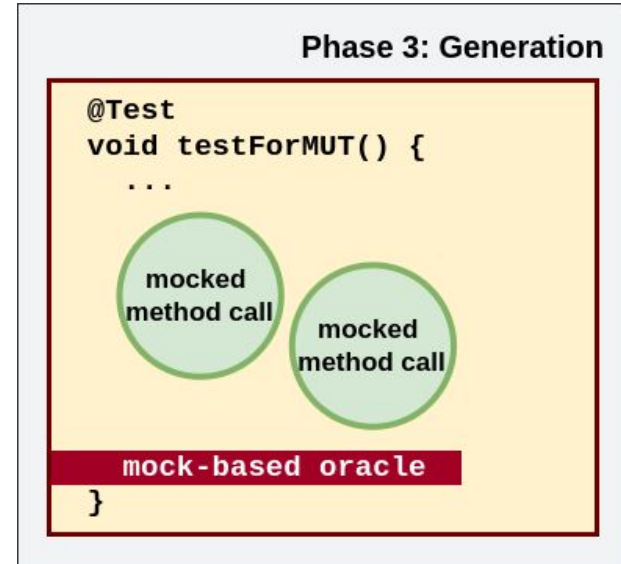
OO: Output oracle
PO: Parameter oracle
CO: Call oracle

Key phases

Phase 1: Identification

nested method call

method under test (MUT)

mockable method call

nested method call

method

mockable method call

Phase 2: Monitoring

monitor

MUT

mockable method call

mockable method call

Phase 3: Generation

```
@Test
void testForMUT() {
    ...
```

mocked method call

mocked method call

`mock-based oracle`

```
}
```
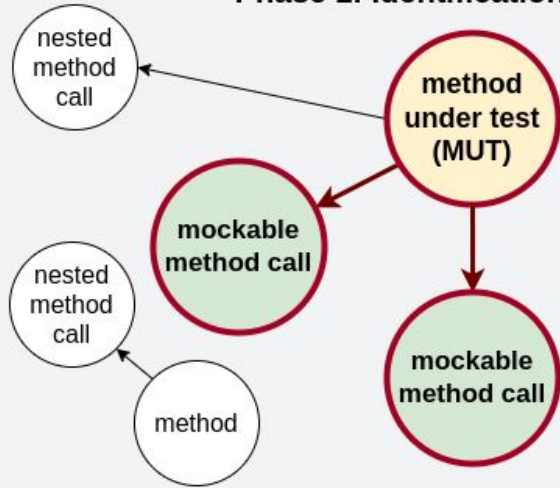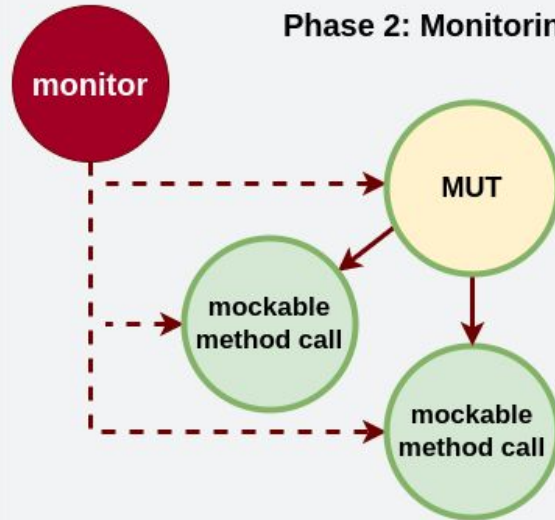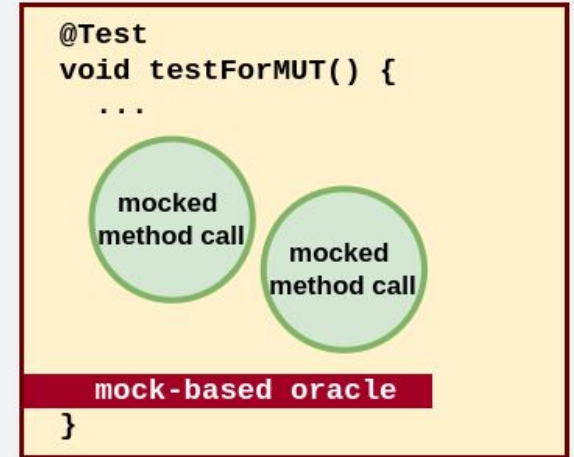
OO, **PO**, **CO**

Introduction: graphhopper maps

- **https://www.graphhopper.com/**

- Routing: Point A to B (to C...)

- On bike, foot, car, (mountain bike, ...)

- Elevation, layers, ...

- **https://graphhopper.com/maps/**

Experiment

Generated tests

Clone

```
git clone git@github.com:Deee92/graphhopper.git --branch 5.3-rick
```

Browse on

https://github.com/Deee92/graphhopper/tree/5.3-rick

MUT #1

MUT:

`com.graphhopper.util.Instruction.getTurnDescription(Translation)`

Mockable:

`com.graphhopper.util.Translation.tr(String, Object…)`

Generated tests:

`com.graphhopper.util.TestInstructionRickGen`

MUT #2

MUT:

com.graphhopper.storage.StorableProperties.loadExisting()

Mockable:

com.graphhopper.storage.DataAccess.loadExisting()

com.graphhopper.storage.DataAccess.getCapacity()

com.graphhopper.storage.DataAccess.getBytes(long, byte[], int)

Generated tests:

com.graphhopper.storage.TestStorablePropertiesRickGen

Discussion

Mocking effectiveness: P1

Statement	Response
I would mock the same objects myself	yes
I would write similar oracles myself	Yes, but all oracles in one test
The oracles represent realistic values which would be useful to developers	Represent actual production behavior but more obvious to developers (e.g., "continue onto Kungsgatan" in MUT1)
The use of mocks allows for more focus on the MUT	yes

Mocking effectiveness: P2

Statement	Response
I would mock the same objects myself	For the muts, would probably mock; Won't mock package-internal or legacy, depends on domain knowledge (how big class is)
I would write similar oracles myself	Probably, but depends on the data, not sure about CO (if performance critical)
The oracles represent realistic values which would be useful to developers	Realistic data, saves time, deciding inputs, corner cases - combinations of parameters, Would typically use my own data
The use of mocks allows for more focus on the MUT	yes

Mocking effectiveness: P3

Statement	Response
I would mock the same objects myself	Prefer to define test doubles manually, not use mocking frameworks, if possible real collaborator, depends on complexity of the collaborator (IO, networking), more control over tests
I would write similar oracles myself	OO is strongest, PO, CO are verifications on different levels, PO stronger than CO
The oracles represent realistic values which would be useful to developers	Make sense for examples (MUTs), I try to avoid primitive types, use complex types
The use of mocks allows for more focus on the MUT	Rigid snapshot is useful, but would complicate refactoring, Not for me, would make test doubles myself

Mocking effectiveness: P4

Statement	Response
I would mock the same objects myself	MUTs can be trivial, but good examples to understand RICK
I would write similar oracles myself	The different oracles make sense, hybrid = manual + RICK, Maybe put in the same test
The oracles represent realistic values which would be useful to developers	Yes, realistic
The use of mocks allows for more focus on the MUT	Mockable without reflection: code is good

Structure: P1

Statement	Response
The tests have a clear structure	AAA is very clear, display info
I would follow a similar structure	Maybe, but Input(AA)-oracle(A) path can also be followed
Suggestions for improvement: name of the test (snake case), display info can also specify the return type, flag for kind of oracle or merge them	

Structure: P2

Statement	Response
The tests have a clear structure	Yes, comprehensible, display name is rare and useful
I would follow a similar structure	Yes, Setup teardown cleanup, Null values and corner cases, crashes
Suggestions for improvement: not at the moment, before after	

Structure: P3

Statement	Response
The tests have a clear structure	Yes, very clear structure
I would follow a similar structure	Yes, use AAA
Suggestions for improvement: extract similar statements in a setup phase / delegated to other method	

Structure: P4

Statement	Response
The tests have a clear structure	Yes, Maybe remove the comment
I would follow a similar structure	Setup etc important, good for generated tests Maybe not constrained for all manually written tests
Suggestions for improvement: displayname could be long for more mocked methods	

Understandability: P1

Statement	Response
The tests are generally intuitive / understandable	Yes, easy to read because of structure
The intention of each test is clear	Yes, PO and CO are different, contribute differently to verification
Suggestions for improvement: N/A	

Understandability: P2

Statement	Response
The tests are generally intuitive / understandable	Yes, very, Data variations, more data for method with branches
The intention of each test is clear	Yes from display name, pattern AAA
Suggestions for improvement: not at the moment	

Understandability: P3

Statement	Response
The tests are generally intuitive / understandable	Yes, due to AAA
The intention of each test is clear	Extract shared code into a method and call that. Assertion phase: CO vs PO is not clear (Java / Python), types
Suggestions for improvement: Comments not necessary - line break suffices for AAA	

Understandability: P4

Statement	Response
The tests are generally intuitive / understandable	yes
The intention of each test is clear	yes
Suggestions for improvement: this is fine / good	

Profile: P1

Years of experience with software development	6 years
Position in team / organisation	Researcher, software that consumes less energy, exploiting software tests as production traffic
Size of team / organisation	4

Mocking practices: P1

- How often do you write tests?
  - **Often, everyday, testing is life**
  - Sometimes
  - Rarely

- How often do you create / use mocks?
  - Often
  - **Sometimes**
  - Rarely

- How familiar are you with Java + JUnit + Mockito?
  - **Proficient**
  - **Comfortable** - Mockito
  - Novice

Profile: P2

Years of experience with software development	30-40 (1982 first computer)
Position in team / organisation	Test environment architect
Size of team / organisation	22 in department, 8k in bank

Mocking practices: P2

- How often do you write tests?
  - **Often**
  - Sometimes
  - Rarely

- How often do you create / use mocks?
  - Often
  - **Sometimes**
  - Rarely

- How familiar are you with Java + JUnit + Mockito?
  - **Proficient**
  - Comfortable
  - Novice

Profile: P3

Years of experience with software development	7 years
Position in team / organisation	Developer, technical coach
Size of team / organisation	Team: 6, org: 220k

Mocking practices: P3

- How often do you write tests?
    - **Often - TDD**
    - Sometimes
    - Rarely

- How often do you create / use mocks?
    - Often: test doubles, fake collaborators
    - Sometimes
    - Rarely: mocks

- How familiar are you with Java + JUnit + Mockito?
    - Proficient
    - Comfortable
    - Novice
    - pytest (mocks included)

Profile: P4

Years of experience with software development	~10 (5 with Graphhoppper)
Position in team / organisation	Developer
Size of team / organisation	3 (6)

Mocking practices: P4

- How often do you write tests?
  - **Often, all the time**
  - Sometimes
  - Rarely

- How often do you create / use mocks?
  - Often
  - **Sometimes, define mock classes / dummy implementation**
  - **Rarely, Graphhopper does not use mockito**

- How familiar are you with Java + JUnit + Mockito?
  - **Proficient**
  - Comfortable
  - Novice

Thanks!

deepikat@kth.se