andrena OBJECTS

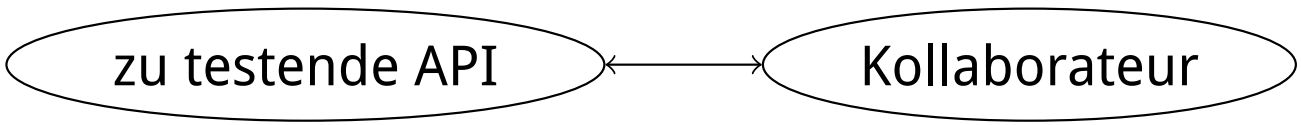
Experts in agile software engineering

Wer braucht eigentlich ein Mocking-Framework?

Simon Wagner Urs Metz
andrena objects ag

Entwicklertag Karlsruhe 2016, 16.06.2016

Ausgangslage



Gründe warum Test Doubles verwendet werden

echter Kollaborateur schlecht für isolierten Unit-Test geeignet weil,

- er langsam ist
- er schwer in gewünschten Zustand zu bringen ist
- er Benutzerinteraktion erfordert
- er sich nicht deterministisch verhält

Arten von Test Doubles

- Dummy
- Stub
- Fake
- Spy
- Mock

Klassifikation ist an Meszaros, *XUnitPatterns.com*, Fowler, *Mocks Aren't Stubs* und Martin, *The Little Mocker* angelehnt.

Beispiel (nach Martin, *The Little Mocker*)

```
public interface Authorizer {
  boolean isAuthorized(User user);
}
```

selbst geschriebene Test Doubles – Dummy

```
public interface Authorizer {
  boolean isAuthorized(User user);
}
```

```
public class AuthorizerDummy implements Authorizer {
  boolean isAuthorized(User user) {
    throw NotImplementException();
  }
}
```

selbst geschriebene Test Doubles – Stub

```
public interface Authorizer {
  boolean isAuthorized(User user);
}
```

```
public class AuthorizerEveryUserAuthorizedStub implements Authorizer {
  boolean isAuthorized(User user) {
    return true;
  }
}
```

```
public class AuthorizerNoUserAuthorizedStub implements Authorizer {
  boolean isAuthorized(User user) {
    return false;
  }
}
```

selbst geschriebene Test Doubles – Fake

```
public interface Authorizer {
  boolean isAuthorized(User user);
}
```

```
public class AuthorizerFake implements Authorizer {
  boolean isAuthorized(User user) {
    if ("Bob".equals(user.getName())
      return true;
    return false;
  }
}
```

selbst geschriebene Test Doubles – Spy

```
public interface Authorizer {
  boolean isAuthorized(User user);
}
```

```
public class AuthorizerSpy implements Authorizer {
  private User suppliedUser;

  public boolean isAuthorized(User user) {
    suppliedUser = user;
    return true;
  }

  public User getSuppliedUser() { return suppliedUser; }
}
```

selbst geschriebene Test Doubles – Mock

```
public interface Authorizer {
  boolean isAuthorized(User user);
}
```

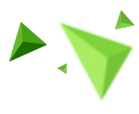
```
public class AuthorizerMock implements Authorizer {
  private User suppliedUser;

  public boolean isAuthorized(User user) {
    suppliedUser = user;
    return true;
  }

  public void verify(User expectedSuppliedUser) {
    assertEquals(expectedSuppliedUser, suppliedUser);
  }
}
```

Vorteile von selbst geschriebenen Test-Doubles

DEMO: aus TDD by Example von Kent Beck



Probleme beim Einsatz eines Mocking-Frameworks

- bei „enger“ Kollaboration aufwändig und führt zu Überspezifikation
- API-Änderung müssen in allen Tests nachgezogen werden
- Unit-Test sagt wenig über Korrektheit des Codes aus

Mocking-Frameworks helfen manchmal doch!

DEMO: angelehnt an Beispiel aus Growing Object-Oriented Software Guided by Tests von S. Freeman & N. Pryce

Vorteile eines Mocking-Frameworks

- echte Mocks aufwändig per Hand zu implementieren und zu verwenden
- einfache Überpüfung welche Nachrichten an Kollaborateur gesendet wurden
- eignet sich gut für Event-APIs

Fazit

Für ein Test-Double benötigt man kein Mocking-Framework!

Charakter des Kommunikationsprotokolls entscheidet,
ob sich Verwendung eines Mocking-Framework anbietet

Fazit

Für ein Test-Double benötigt man kein Mocking-Framework!

Charakter des Kommunikationsprotokolls entscheidet,
ob sich Verwendung eines Mocking-Framework anbietet

Bibliographie

- Martin Fowler. *Mocks Aren't Stubs*. 2004. URL: http://martinfowler.com/articles/mocksArentStubs.html.
- Robert C. Martin. *The Little Mocker*. 2014. URL: http://blog.8thlight.com/uncle-bob/2014/05/14/TheLittleMocker.html.
- Gerard Meszaros. *XUnitPatterns.com*. 2009. URL: http://xunitpatterns.com/.