Testing

PA165
Dec 9, 2014
Petr Adámek, Tomáš Pitner

Testing of Applications

- Testing verifies compliance with the *specification* and implementation of *customer expectations*.
- It is an important part of the quality management in software development.
- Unlike formal verification does not allow to detect *all* potential errors.

Základní pravidla

- **Tests should be *reproducible*.**
  - Somebody else should be able to perform the same tests with the same results.
- **Tests should be *deterministic*, i.e. they should have the same input conditions at the beginning.**
- **Tests should be *independent*, i.e. not to be influenced by each other.**
  - Usually by setting the same input conditions for each test
- **Tests should be *repeatable cheaply*.**
  - It usually means running in an automated way.

Modes of Testing

- **Manual testing:**
  - low entry costs;
  - expensive repetitions;
  - difficult to ensure reproducibility, determinism and independence
- **Automated testing:**
  - high input costs;
  - cheap repetition;
  - easy to ensure reproducibility, determinism and independence.

Types of Testing According to Goals

- **Unit testing**
  - Does the unit work independently on the context?
- **Integration testing**
  - Does the component work integrated in its environment?
- **Functional testing**
  - Does it fulfill the functional requirements?
- **Acceptance Testing**
  - Is it good for the customers? Will they accept it?

Types of Testing According to Goals

- Performance and scalability testing
- Testing the user-friendliness
- Security testing

Unit Testing

- In unit testing we try to test the individual components of the system being developed at the lowest level.
- Individual test components should be isolated from its surroundings to avoid the influence of the surroundings on the test component.
- Interaction with the environment is simulated using mock-objects that simulate the behavior of the neighborhood in a particular test scenario.
- The better the decomposition is done, the easier the unit testing is.

Tools for Unit Testing

- JUnit
- TestNG

Example

```
public class CalculatorTest {

    private Calculator c;

    @Before
    public void setUp() {
        c = new Calculator();
    }


    @Test
    public void testDivide() {
        assertEquals( 9, c.divide( 99, 10));
        assertEquals(10, c.divide(100, 10));
    }


    @Test(expected = IllegalArgumentException.class)
    public void testDivideByZero() {
        c.divide(100, 0);
    }
}
```

Basic Rules

- The test outputs are always *Yes / No* (Boolean)
- *First test, then code* (see XP and TDD)
- When the error is to be corrected: *first test, then fix* (protection against regression)
- *Trivial* get / set methods are not tested
- Test all *non-standard* situations and *limit* values
- Error messages and comments *not always needed*
- Tests runs after *every change*

Interactions with Environment

- Components should be tested in isolation.
- But it is necessary to simulate the kind of interaction with the environment.
- That is what Mock objects do.
- These objects must be type compatible with simulated component:
  - Inheritance
  - Implementing an interface (preferable)
- Mock objects can be created manually (tedious), or through tooling:
  - Mockito, EasyMock, JMock

Example (manually created Mock objects)

```
public class CurrencyConvertorTest {
  @Test
  public void testConvert() {
    ExchangeRateTable exchangeRateTable = new ExchangeRateTable() {

      public void setExchangeRate(Currency currency, BigDecimal exchangeRate) {
        throw new UnsupportedOperationException("Not supported yet.");
      }
      public BigDecimal getExchangeRate(Currency currency) {
        return BigDecimal.valueOf(28.2);
      }
    };

    CurrencyConvertor convertor = new CurrencyConvertor(exchangeRateTable);
    Currency czk = Currency.getInstance("CZK");
    BigDecimal actualResult = convertor.convert(czk, BigDecimal.valueOf(10));
    BigDecimal expectedResult = BigDecimal.valueOf(282.0);
    assertEquals(expectedResult, actualResult)
  }
}
```

Example (Mockito)

```
@RunWith(MockitoJUnitRunner.class)
public class CurrencyConvertorTest {

  @Mock
  ExchangeRateTable exchangeRateTable;

  @Test
  public void testConvert() {

    when(exchangeRateTable.getExchangeRate(czk)).
      thenReturn(BigDecimal.valueOf(28.2));

    CurrencyConvertor convertor = new CurrencyConvertor(exchangeRateTable);
    Currency czk = Currency.getInstance("CZK");
    BigDecimal actualResult = convertor.convert(czk, BigDecimal.valueOf(10));
    BigDecimal expectedResult = BigDecimal.valueOf(282.0);
    assertEquals(expectedResult, actualResult)
  }
}
```

Unit Testing in Java EE

- For Java EE applications, it is necessary to take into account the existence of the container.
  - Tests outside the container - test only business logic, not behavior depending on the container (such as transaction management, authorization, etc.)
  - Tests in a container - will test everything, but this kind of testing for unit tests not fit.
- In testing outside of the container concept is used mock objects which simulate the behavior of the container.

Unit Testing - Data

- How to test data persistence layer:
  - Mock objects (easy with JPA or other libraries and frameworks, complicated by the low-level JDBC).
  - Database is stored in memory (easy for JPA, with low-level JDBC may be a problem with the SQL dialect).
- Do not forget to provide the same initial conditions (state database is always the same initial state).
- What can help
  - *DBUnit*
  - *Abstract DAO*

What else can help?

- **Tools for measuring *test coverage***
  - Line Coverage
  - Branch Coverage
- **Tools for generating *test data***
- **Extended set of *assert methods***
- **Etc.**

Integration Testing

- *Integration testing* is used to verify the correct interaction of individual components that are assembled and the system behaves as expected in its specification.
- See also *continuous integration*

Functional Testing

- Functional testing is used to verify the functionality of the end-user perspective.
- Mostly performed at the user interface level
- Rational Functional Tester - web GUI+
  - `http://www-01.ibm.com/software/awdtools/tester/functional/index.html`
- Selenium IDE – web
  - `http://selenium.openqa.org/`
- Marathon – GUI
  - `http://marathonman.sourceforge.net/`
- Rational Robot - GUI(for legacy applications), *Rational Quality Manager*, *JWebUnit*

Acceptance Testing

- Customer acceptance testing verifies that the application meets customer's requirements and expectations.
- Absence of acceptance testing (or its underestimation and lack of design) almost always leads to future disputes and problems.
- Customers unfortunately have a tendency to underestimate it. The non-compliance of the implementation with the customer's requirements so often comes at the moment of production deployment :-(.

Performance and Scalability Testing

- Performance testing verifies system throughput and response time at high loads.
- Part of the specification should be the definition of the throughput and response times of the prescribed load.
- *Rational Performance Tester* (+ extensions)
- `http://www-01.ibm.com/software/awdtools/tester/performance/index.htm`
- *Rational Service Tester* for SOA Quality (functional testing, performance testing +)
- JMeter - `http://jakarta.apache.org/jmeter/`

Usability Testing

- In the USA a common thing, in Europe still not so obvious and Asia is likely to overtake Europe in this.
- The definition of the prototype of the target user.
- Select a group of test users (test sample).
- Test user is given a list of tasks that are trying to solve without the help of someone else.
- His/her behavior is monitored and evaluated.
- See Štefkovič, M.: Usability of Web applications. https://is.muni.cz/auth/th/166042/fi_b/ (Bc. Thesis)

Security Testing

- Security testing checks resistance against various security attacks.
- Tools:
- Rational *AppScan* - web app security testing
- `http://www-01.ibm.com/software/awdtools/appscan/`

Questions