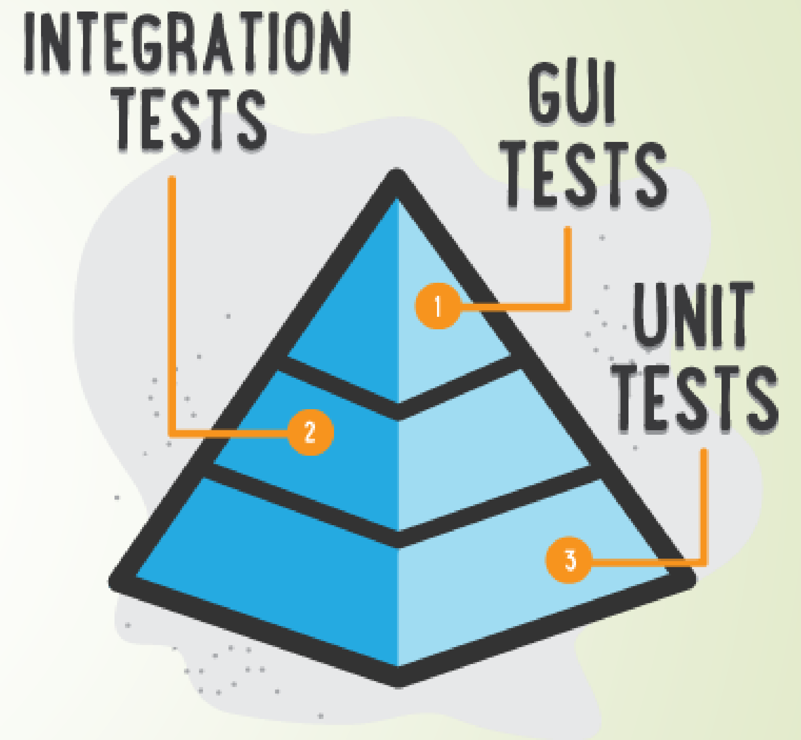
Unit test

- A legalacsonyabb szintű tesztelés. A programot felépítő egységek tesztelése
- Unit: egy rendszer legkisebb önálló egységként tesztlehető része.
- Unit tesztekkel ellenőrizhető, hogy egy unit az elvárásoknak megfelelően működik.
- Egy unit függvényeiről ellenőrizzük, hogy különböző bemetek esetén megfelelő eredményt, vagy hibát produkálnak.
- Az egységeket egymástól izoláltan kell tesztelni.



Előnyei

- A hibák sokkal korábban észlelhetőek.
- Minden komponens legalább egyszer tesztelt.
- Az egységek elkülönítése miatt a hibák helyének meghatározása könnyű.
- A funkciók könnyen módosíthatóak átalakíthatók.
- Dokumentációs szerep: példákat biztosít egyes funkciók használatára.

Elvek

- **Gyors:** A teszteknek gyorsan kell futnia, lassú teszteket senki nem futtatja gyakran, így a hibák nem derülnek ki idejében.
- **Független:** A teszteknek egymástól függetlennek és bármilyen sorrendben végrehajthatónak kell lennie.
- **Megismételhető:** A tesztkenke bármilyen környzetben, hálozat nélkül is végrehajthatónak kell lennie. A különöbző futtatások eredményének meg kell egyeznie.
- **Ön ellenőrző:** A tesztek eredménye egy boolean, futásuk vagy sikeres, vagy sikertelen.
- **Automatikus:** A tesztek automatikusan, interakció nélkül futnak.

Test Driven Development – TDD

- Egy fejlesztési folyamat amelynek lépései:

1. Teszt eset készítése, amely definiálja a szükséges funkciót, annak bemenő adatait és az adatokra adandó választ. (a teszt kezdetben sikertelenül fut)
2. A funkció elkészítése a teszt sikeres futását elérő lehető legkevesebb kóddal.
3. Refactorálás: a kód megváltoztatása, szépítése, duplikátumok eltávolítása (clean code)
4. Tesztesetek készítése a funkció további lehetséges bemenetiehez.

- TDD rákényszerít a probléma alaposabb elemzését és az funkcióktól elvárt eredmény jobb megértését.
- A kód így mindig le lesz fedve tesztekkel

Mit kell tesztelni

- Egy osztály minden publikus metódusát tesztelni kell.
- „Triviális” esetek
- Speciális esetek: pl számok esetén: negatív, 0, pozitív, a megengetettnél kisebb, nagyobb, stb..
- Pozitív/Negatív tesztesetek. Negatív teszteset szándékosan hibás paraméterekkel hívja a tesztelt metódust, célja a hibakezelés ellenőrzése.
- Végrehajtási lefedettség: a tesztekből indított hívásoknak a tesztelt metódus lehető legtöbb során végig kell haladnia.
- Egy teszt egyetlen „dolog”

Mit NEM kell tesztelni

- Dolgok, amik nyílvánvalóan működnek: pl.: külső lib-ek, JDK...
- Adatbázis. (feltételezhető, hogy ha elérhető, akkor helyesen működik)
- Triviális metódusok (getter/setterek)
- GUI – a user interface nem tartalmazhat üzleti logikát

Junit (4.x)

- Junit egy unit teszt keretrendszer a Java nyelvhez.
- Annotációkkal konfigurálható.
- Elvárt eredmények ellenörzése Assert-ekkel.

Például

```
public class MyTests {
  @Test
  public void multiplicationOfZeroIntegersShouldReturnZero() {
    // MyClass is tested
    MyClass tester = new MyClass();
    // assert statements
    assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
    assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
    assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
  }
}
```

Teszt osztály felépítése

Annotation	Result
@Test public void method()	Teszt metódus jelölése
@Test (expected = Exception.class)	A teszt sikertelen a megadott típusú kivétel hiányában
@Test(timeout=100)	A teszt sikertelen, ha a metódus mem fejeződik be adott idő alatt
@Before or @After public void method()	Metódus amely minden teszt előtt/ (után) lefut.
@BeforeClass or AfterClass public **static** void method()	Metódus egyszer fut le, az osztályban lévő első teszt metódus indulása előtt
@Ignore or @Ignore("Why disabled")	Adott teszt metódus kihagyása

Junit ellenőrzések Assert – ek

Statement	Description
fail(message)	A hívó teszt sikertelen futását eredményezi
assertTrue([message,] boolean condition) or assertFalse	Ellenörzi, hogy a megadott feltétel igaz (hamis) – e.
assertEquals([message,] expected, actual)	Ellenőrzi, hogy két objektum egyenlő-e.
assertNull([message,] obj) or NotNull	Egy Objektum null (nem null).
assertSame([message,] expected, actual) or assertNotSame	Ellenőrzi, hogy két változó ugyanarra az objektumra mutat (==).

Testing exception

- A @Test (expected = Exception.class) annotáció limitált, csak egyetlen kivételt tud tesztelni. Alternatíva:

```
try {
    mustThrowException();
    fail();
} catch (Exception e) {
    // expected
}
```

Például

```java
public class MyClass {
    public int multiply(int x, int y) {
    // the following is just an example
    if (x > 999) {
      throw new IllegalArgumentException("x>999");
    }
    return x / y;
  }
}
```

```java
public class MyClassTest {
  @Test(expected = IllegalArgumentException.class)
  public void testExceptionIsThrown() {
    MyClass tester = new MyClass();
    tester.multiply(1000, 5);
  }
  @Test
  public void testMultiply() {
    MyClass tester = new MyClass();
    assertEquals(50, tester.multiply(10, 5));
  }
}
```

Finished after 0.012 seconds

Runs: 2/2   Errors: 0   Failures: 1

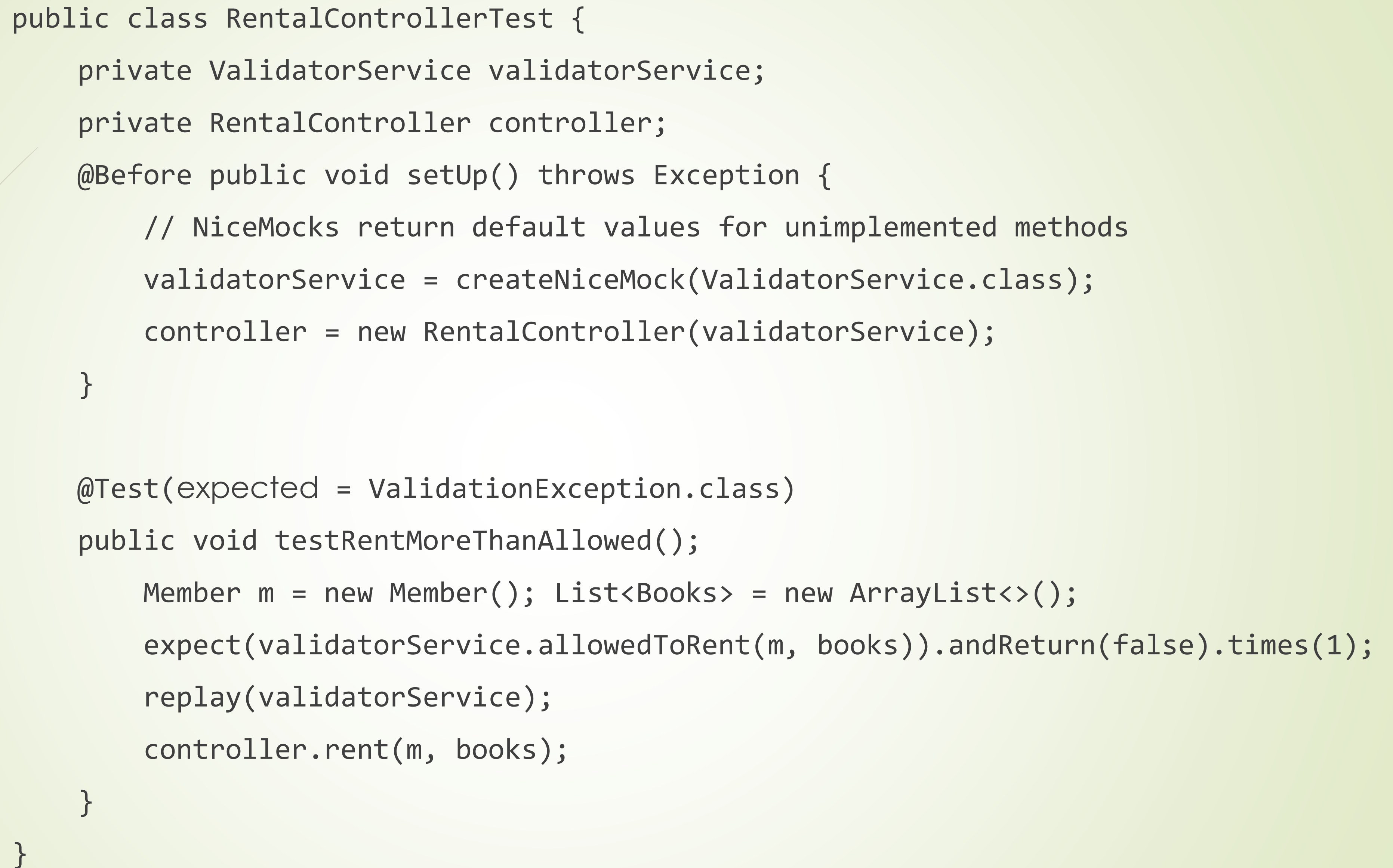
com.vogella.junit.first.MyClassTest [Runner: JUnit 4] (0.000
testExceptionIsThrown (0.000 s)
testMultiply (0.000 s)

Failure Trace
java.lang.AssertionError: Result expected:<50> but was:<2>
at com.vogella.junit.first.MyClassTest.testMultiply(MyClass

- A tesztelt metódusok általában más komponens nyújtotta szolgáltatásokat is használnak.
- Az egyes funkciókat más komponensektől izolációban kell tesztelnünk, a tesztek sikeressége nem függhet a függőségek helyes implementációjától.
- Mock egy olyan objektum, amely egy másik objektum működését szimulálja.
- Egy osztály függőségeinek szimulálására használandó.
- http://easymock.org/

Például

```
public class RentalController {
    private ValidatorService validatorService;
    public RentalController(ValidatorService validatorService) {
        this.validatorService = validatorService;
    }
    public void rent(Member m,  List<Book> books) {
        if(!validatorService.allowedToRent(m, books.size()) {
            throw new ValidationException();
        }
        ...
    }
}
public interface ValidatorService {
    boolean validatorService.allowedToRent(m, numberOfBooksToRent);
}
```



```
public class RentalControllerTest {
    private ValidatorService validatorService;
    private RentalController controller;
    @Before public void setUp() throws Exception {
        // NiceMocks return default values for unimplemented methods
        validatorService = createNiceMock(ValidatorService.class);
        controller = new RentalController(validatorService);
    }

    @Test(expected = ValidationException.class)
    public void testRentMoreThanAllowed();
        Member m = new Member(); List<Books> = new ArrayList<>();
        expect(validatorService.allowedToRent(m, books)).andReturn(false).times(1);
        replay(validatorService);
        controller.rent(m, books);
    }
}
```