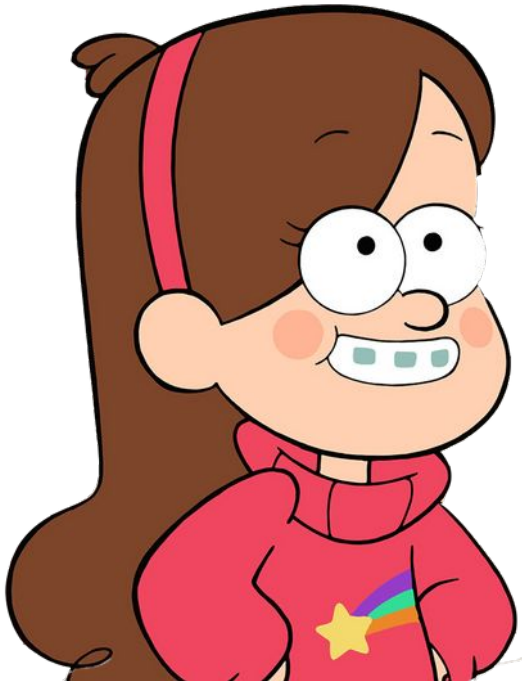


Unit testing

And the importance of it

Днес ще говорим за

- ★ Какво са тестове
- ★ Защо и кога трябва да пишем тестове
- ★ Дефиниции, свързани с тестовете
- ★ Какво е Unit Test и как се пише
- ★ Какво е JUnit и за какво се използва
- ★ Добри практики за писане и именуване на тестове
- ★ Класът Assert
- ★ Задачи

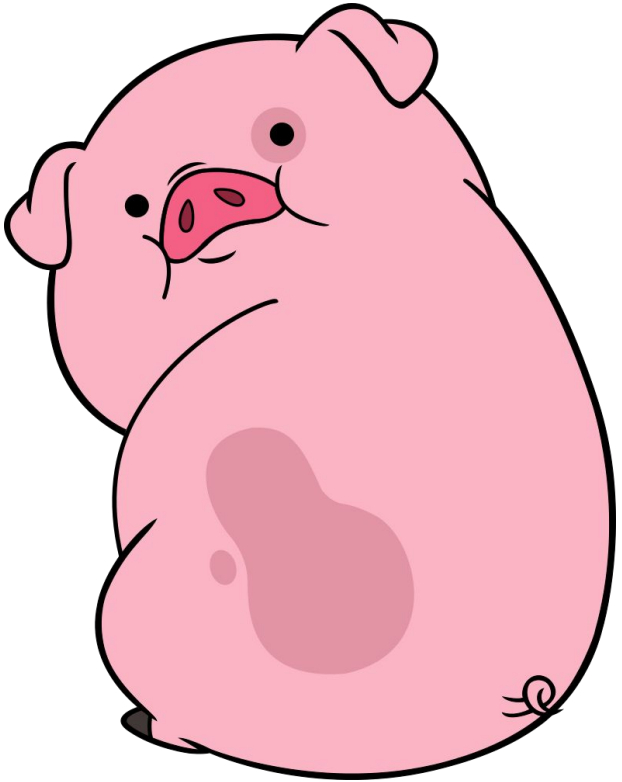
Защо трябва да тестваме софтуера?

- ★ За да не произвеждаме код с бъгове
- ★ За да не изгубим много пари и време
- ★ За да се възприема кодът ни като надежден
- ★ За да си спестим трудности при поддръжката

Важно е да подсигурим код с минимален брой грешки

Какво са тестове?

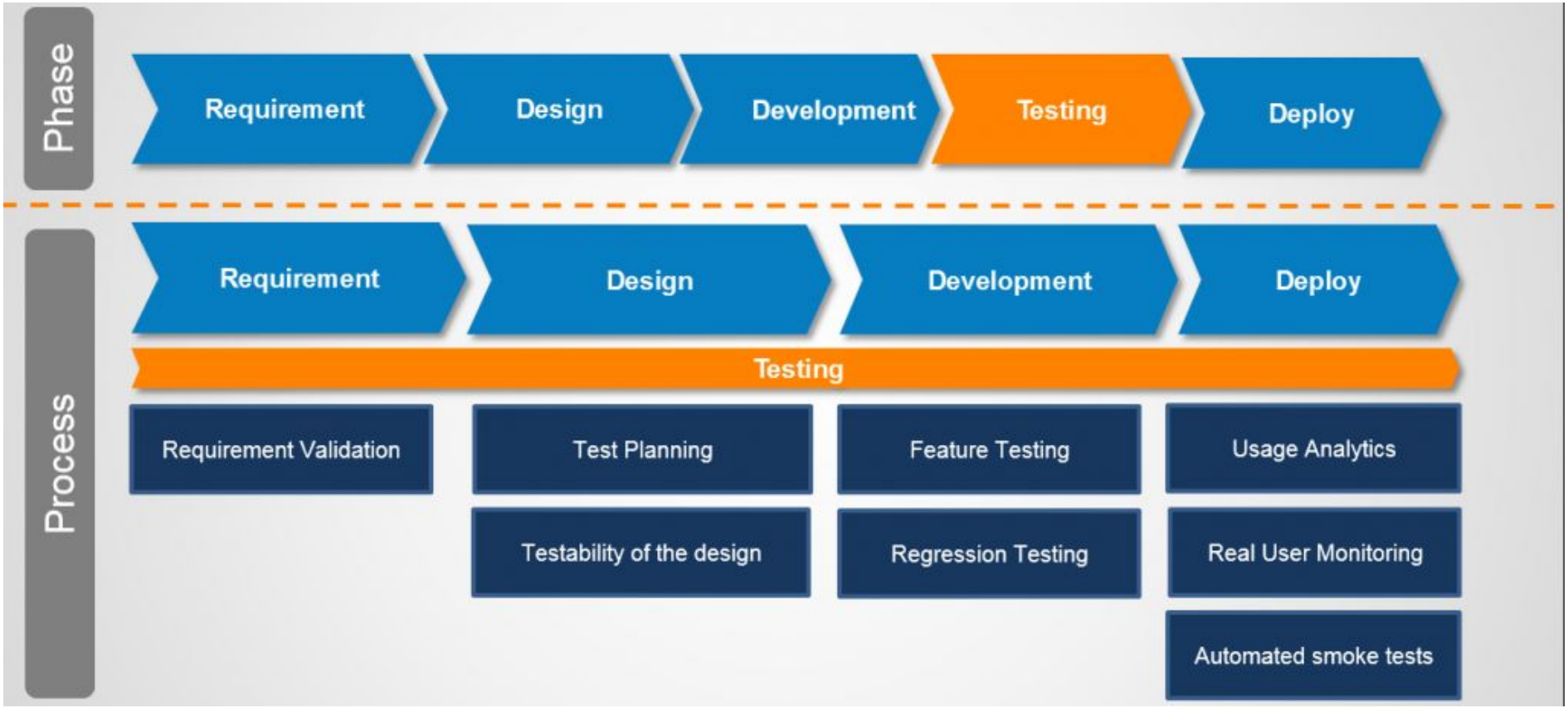
- Последователност от действия, която подсигурява правилното изпълнение на дадена функционалност
- Тестовете се създават по определен тестов случай
- Сравнение на очакваният и реалният резултат от тестваното парче код



```
@Test
public void testLoginWhenPasswordIsIncorrectThenReturnFalse(){
    //test Login(String password) method here
}
```

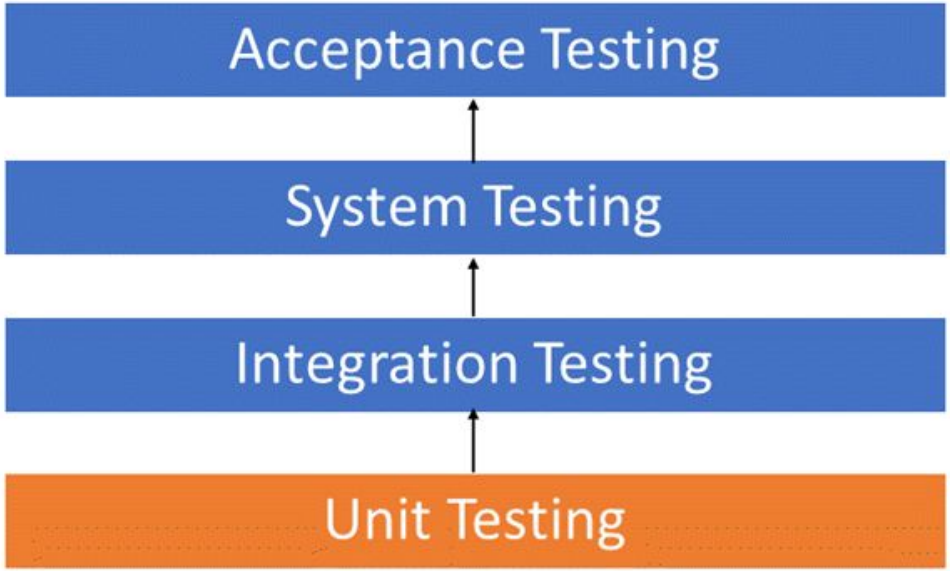
Кога трябва да пишем тестове?

- ★ Тестовете не са етап от създаването на програма, те са процес
- ★ Тестове трябва да се пишат успоредно с останалият код
- ★ Не е добра практика всички тестове да се напишат накрая
- ★ Ако пишем тестовете си навреме, ще си спестим много проблеми в бъдещето


Phase
Requirement
Design
Development
Testing
Deploy
Process
Requirement
Design
Development
Deploy
Testing
Requirement Validation
Test Planning
Feature Testing
Usage Analytics
Testability of the design
Regression Testing
Real User Monitoring
Automated smoke tests

Видове автоматични тестове

- ★ Функционални тестове
  - unit
  - integration
- ★ Нефункционални тестове
  - performance
  - stress
  - crash
  - security
  - usability



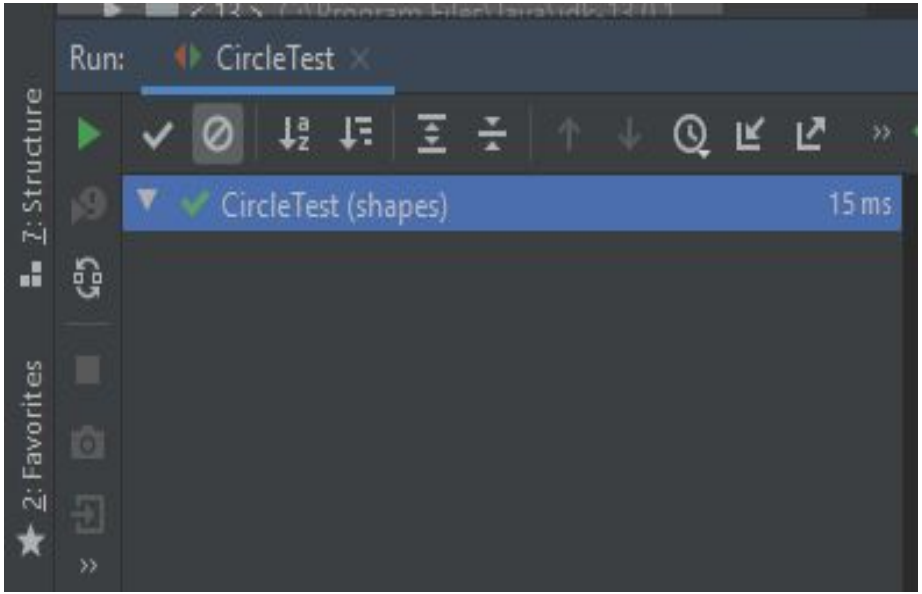
Дефиниции

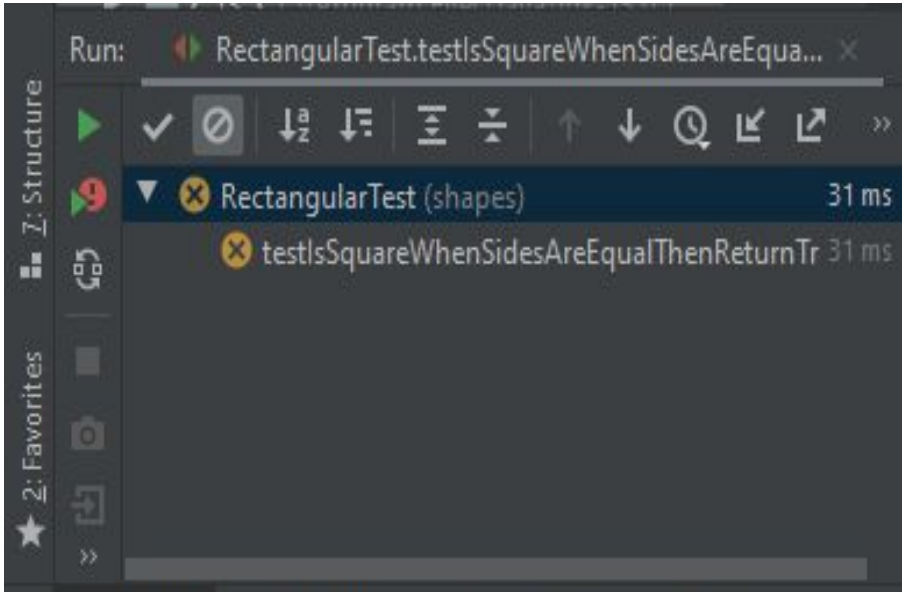
- ★ **Продуктивен код (code under test)** - кодът, който реализира потребителските изисквания
- ★ ***test coverage* или *code coverage*** - *процентът на продуктивния код, който се тества от автоматични тестове*

Test Driven Development (TDD)

- ★ Test driven development (TDD) е методология, при която тестовете се пишат преди продуктивния код
- ★ Подсигуряваме, че когато тестът мине, кодът, който сме написали работи по начина, по който искаме
- ★ Не се изкушаваме да нагласим теста, така че да минава
- ★ щом даден тест бъде удовлетворен (т.е. минава успешно, стане „зелен“), съответният use-case е реализиран („done“)

Какво означава тест да “мине” или “фейлне”?





Unit tests

- ★ Код, който изпълнява специфична, „атомарна“ функционалност на кода, която да бъде тествана
- ★ Тества се малък фрагмент код (метод или най-много клас)
- ★ Гаранция, че кодът ще работи както искаме
- ★ Подсигурява работата на кода, дори когато го модифицираме

Анотации

- ★ @Test
  - обозначава метод като тестов метод
- ★ @Test(timeout = 100)
  - фейлва, ако изпълнението на метода продължи повече от 100 милисекунди

JUnit

- ★ Най-популярният framework за писане на тестове в Java
- ★ Базира се на **анотации**
- ★ Всеки тест е метод с **анотация** @Test
- ★ Тестовете се намират в клас, който се използва **само** за тестване (test case)

JUnit setup

Best practices

- ★ Тривиален код като гетъри и сетъри няма нужда от тестове
- ★ Тестовете трябва да са кратки, точни и ясни
- ★ 70-80% code coverage
- ★ Един тест трябва да покрива само един сценарий
- ★ Тестовете трябва да са независими
- ★ Тестовете са безполезни, ако кодът ни не е добре разделен на **методи!**

Конвенции за именуване

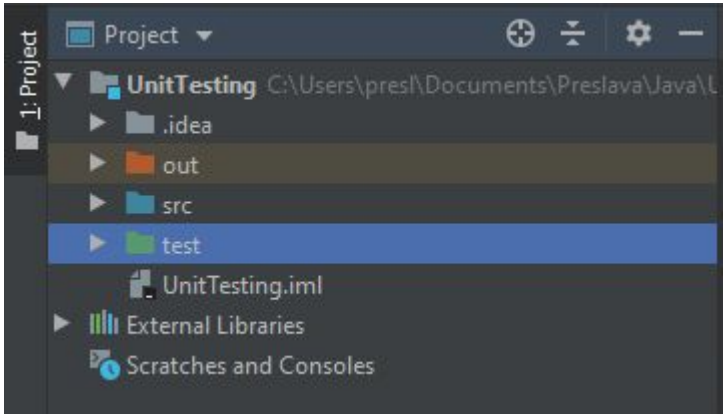
- Името на тестовия клас = Името на тестваният клас + Test
  - пр: Login
    LoginTest
- Имената на тестовете започват с test, след това следва името на тестваният метод и кратко описание на тестовия сценарий
  - test<methodName>When<scenario>Then<expectedResult>
  - пр: testLoginWhenPasswordIsIncorrectThenReturnFalse

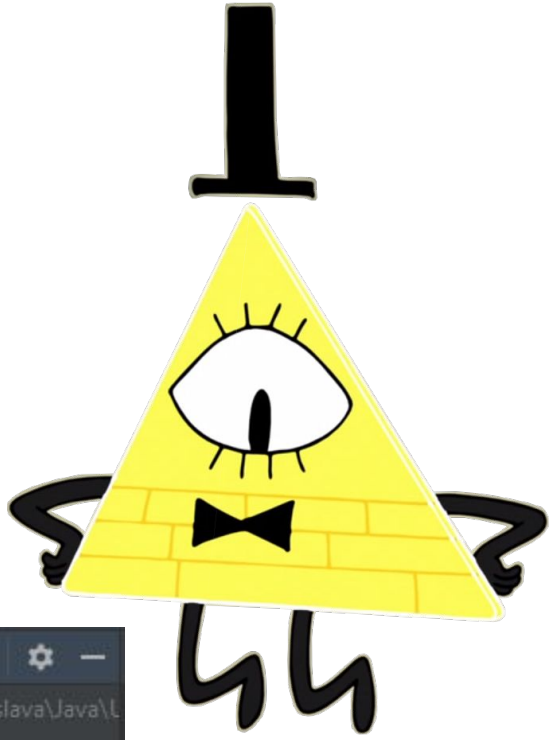
Ред на изпълнение

- ★ По default , тестовете се изпълняват в случаен ред
- ★ Добре написаните тестове са независими и би трябвало да работят без значение от реда им на изпълнение
- ★ От JUnit 4.11 натам тестовете може да се подредят, но не е препоръчително!

Къде “живеят” тестовете

Обикновено unit тестовете се разполагат в отделен проект или в отделна source директория, за да са отделени от продуктивния код





Класът Assert

`org.junit.Assert`

assertEquals(String message, expected, actual)

```
public class HelloWorld{
     public String sayHello(){
          return “Hello world!”;
     }
}
```

```
public class HelloWorldTest {

     @Test
     public void testSayHello() {
     HelloWorld hello = new HelloWorld();
          Assert.assertEquals(“Method doesn’t return
               expected value”, “Hello world!”,
               hello.sayHello());
     }
}
```

`assertArrayEquals(String message, expected, actual)` - за сравнение на масиви по дължина и съдържание

`assertEquals(String message, expected, actual, delta)` - за сравнение на числа с плаваща запетая. Delta определя точността на сравнението

assertTrue(String message, boolean condition)

```
public class Calculator {
      public boolean areEqual(int a, int b){
            return a == b;
      }
}
```

```
public class CalculatorTest {
      private Calculator calc = new Calculator();

      //in this case test passes
      @Test
      public void testAreEqualWhenInputNumbersAreSame(){
            Assert.assertTrue("Method doesn't
                  return expected value", calc.areEqual(1, 1));
      }

      //in this case test fails
      @Test
      public void testAreEqualWhenInputNumbersAreDifferent(){
            Assert.assertTrue("Method doesn't
                  return expected value", calc.areEqual(1, 2));
      }
}
```

`assertFalse(String message, boolean condition)` - за проверка дали дадено условие е лъжа

`assertNull(String message, Object o)` - за проверка дали даден обект **е** Null

`assertNotNull(String message, Object o)` - за проверка дали даден обект **не е** Null

@Before & @After

```
public class NumberListTest {

    private NumberList numList;

     @Before
     public void initialize() {
          numList = new NumberList();
     }

     @After
     public void initialize() {
          numList.clear();
     }

     @Test
     public void testFindSizeWhenNoElementsThenReturnZero() {
          //test here
     }
}
```

@BeforeClass & @AfterClass

```
@BeforeClass
public static void setup() {
    LOG.info("startup - creating DB connection");
}

@AfterClass
public static void tearDown() {
    LOG.info("closing DB connection");
}
```

Let’s dive in!

Какво научихме?

- ★ Тестовете са важни!
- ★ Винаги трябва да пишем тестове
- ★ Тестовете гарантират качеството на кода
- ★ Unit тестовете трябва да са точни и ясни
- ★ Как да пишем и именуваме тестове
- ★ За да има полза от тестовете, кодът ни трябва да е добре разделен на методи

задачи

задачи

задачи

задачи

задачи

задачи

задачи

задачи

задачи

1. Създадена е програма, която ни помага да разбираме различните фигури по-добре. Тя намира обиколките и лицата им, както и някои специфични за фигурата особености. Имплементирани са класове `Circle`, `Rectangular` и `Triangle`.
Кода може да откриете на https://github.com/VratsaSoftware/java19/tree/master/UnitTesting/src/shapes.
Тестовете на https://github.com/VratsaSoftware/java19/tree/master/UnitTesting/test/shapes

1.1. Нека да разгледаме и да коментираме:

- имплементацията на кода
- вече написаните тестове

за класовете `Circle` и `Rectangular`.

1.2. По подобен начин нека сега разгледаме логиката на класа `Triangle` и да напишем подходящи тестове, с които да тестваме поведението на кода. При нужда да се поправят грешките.

2. В тази задача ще влезем в ролята на `QA`. Това, което те правят ежедневно, е да се грижат, че кодът, написан от програмистите, работи правилно. Тестовете, които те пишат, трябва да покриват колкото може по-голям процент от сценариите, които могат да се разиграят, и да поправят откритите бъгове. Кода може да откриете на https://github.com/VratsaSoftware/java19/tree/master/UnitTesting/src/highway.
Тестовете на https://github.com/VratsaSoftware/java19/tree/master/UnitTesting/test/highway

Създадено е приложение, което цели събирането на такса за магистрала. На магистралата може да има два типа превозни средства - `Cars` и `Motorbikes`. Всеки от двата класа има член-данни и имплементирани методи. Програмата пази данните за всички превозни средства, които се движат по пътя в момента. Превозно средство може да се качва или да слиза от магистралата.

2.1. За целта нека първо заедно да разгледаме следния код и да коментираме какви тестове могат да се напишат, за да се тества функционалността на готовото приложение.

2.2. Както видяхме, понякога даже и програмистите допускат пропуски/грешки, които в някои случаи могат да се окажат фатални. Затова е нужно да се пишат достатъчно много тестове, които да откриват възможни проблемите. Нека сега заедно да поправим откритите проблеми.

Домашно

Домашно

Домашно

Домашно

Домашно

Домашно

Домашно

Домашно

Домашно

Домашно

Домашно

1.1. Довършете писането на тестове за задачата за фигурите. Тествайте напълно класовете `Rectangular` и `Triangle`. Тествайте всички възможни случаи, за които се сетите, и подсигурете, че кода работи правилно. Например методът `isIsosceles()`, който проверява дали даден триъгълник е равнобедрен, трябва да се тества поне три пъти - всяка страна с всяка друга

1.2. Довършете писането на тестове за задачата за магистралата. Разгледахме защо родителски клас `VehicleBase` е по-удачен от интерфейса `Vehicle`. Рефакторирайте класа `Motorbike` по същия начин както направихме с `Car`. Помислете как да си подсигурите, че няма как едно и също превозно средство да се движи два пъти на магистралата. Например проверете за наличие на такъв `registrationNumber` в листа.

Забележка: не е нужно да добавяте нова функционалност за тези задачи, тествайте само вече имплементираната, но ако решите да го направите не забравяйте прилежащите тестове. При наличие на грешка в логиката или структурата (разделяне по класове, наследяване и т.н.) на приложението - поправете я.

2. (Бонус задача) Подобно на последната задача, която направихме заедно, напишете подходящи тестове за предоставения код (чието условие може да намерите https://github.com/VratsaSoftware/java19/blob/master/UnitTesting/src/cart/Shopping%20Cart%20Exercise.pdf , а имплементация - https://github.com/VratsaSoftware/java19/tree/master/UnitTesting/src/cart) и поправете грешките. Имайте предвид, че е възможно те да не са само в логиката на приложението, а може и някои добри практики да са нарушени.