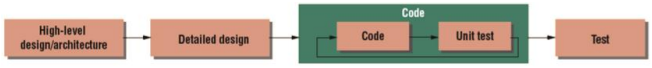
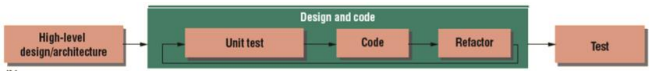
Test Driven Development

Cannot change file attributes.
File exists.

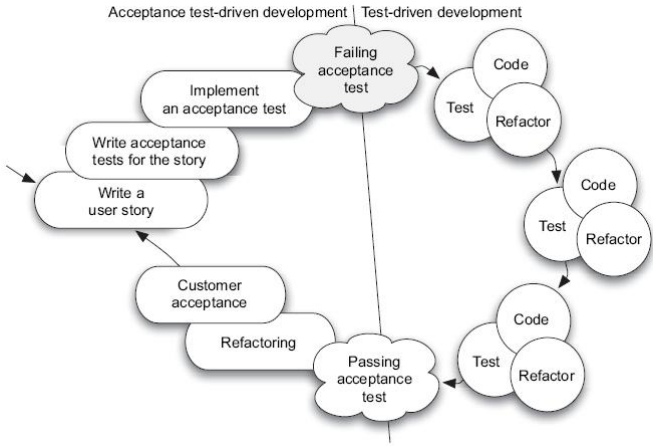
- "Vanha kansanperinne"



- Testilähtöinen / testiohjattu ketterä ohjelmistokehitys



- Laadi oikean toiminnan varmentavat testit ennen toteutuksen kirjoittamista
- Jatkuva integrointi keskeisessä roolissa
  - lisää inkrementti vasta, kun toteutus läpäisee testit
  - varmista, että kaikki aiemmat testit menevät edelleen läpi



Koskela: Test Driven. Manning, 2008.

ATDD ==> TDD

- Käyttäjätarinan hyväksymistesti ei mene läpi
  - koska tarvittavia toimintoja ei ole toteutettu koodina
- Suunnittele yksikkötasolla lisättävä uusi toteutustason ratkaisu
  - oliot ja niiden tarjoamat palvelut (toiminnot)
    - mitä lisättävien metodien tulee tehdä
    - suunnittele metodien kutsurajapinnat
    - kuinka toiminnon lopputulos havainnoidaan
- Toteuta toiminnallinen yksikkö TDD-periaatteen mukaisesti
  a) laadi yksittäistä metodia testaava yksikkötesti
  b) toteuta koodi, jotta testi menee läpi
  c) paranna toteutusta (refaktoroi)
  - ja jatka jälleen kohdasta a)
- Jossain vaiheessa toiminnallisuutta on lisätty riittävästi, jolloin hyväksymiskriteerit täyttyvät eli testi menee läpi
- Inkrementaalisuus

Miksi ATDD?

- Hyväksymiskriteerit/-testit määritellään varhaisessa vaiheessa
  - testit mietitään ennen käyttäjätarinan toteuttavan koodin luontia
  - vaatimukset oltava kirjattu niin täsmällisesti, että ne ovat testattavissa
- Yhteistyökeskeinen lähestymistapa, tiimityötä
  - asiakas mukaan testausprosessiin ennenkuin koodattu pätkääkään
  - tiimi työskentelee yhdessä jokaisessa kehitysprosessin vaiheessa
  - koko tiimi osallistuu jokaiseen keskusteluun tai tapaamiseen, joissa tuotteen ominaisuuksia esitetään, analysoidaan tai arvioidaan
- Lyhyet sprintit ja jatkuva integraatio varmistaa, että laadusta saadaan nopeasti palautetta
  - ohjaa seuraavan vaiheen tulosta (ja sen palautetta)
  - vähentää ylimääräisiä viipeitä
  - lopputulos haluttu / lähempänä haluttua
- Laadusta vastaa koko tiimi
  - siis myös asiakas

Miksi TDD?

- Rajapinnan (API) kuvaus tarkentuu
  - pakottaa ajattelemaan rutiinia kutsujan ja sen toteutuksen termein
  - joutuu aina miettimään esim. luokan ja operaatioiden käyttöä
- Virheiden jäljitys helpompaa
  - jos testi ei mene läpi, syytä helpompi etsiä viimeksi lisätystä osasta
- Saat automatisoidun ja kattavan testipetin
  - jokaista koodikokonaisuutta kohden testejä
  - kaikki testit helppo ajaa uudestaan (regressiotestaus)
  - vaivatonta todeta, että muutokset eivät riko aiemmin kirjoitettua
- TDD kurinalainen tapa suunnitella ja ohjelmoida
  - ohjelmoijat / testaajat etenevät yksi yksityiskohta kerrallaan
  - jokainen uusi toiminnallisuus tulee testattua
- Tavallaan enemmän ohjelmointi- ja suunnittelumenetelmä kuin testausmenetelmä
  - tuloksena virheettömämpi koodi kuin testattaessa jälkikäteen

Käyttäjätarinat

- Käyttäjätarina
  - yksi selkeä toiminnallinen vaatimus, mitä järjestelmän on tuettava
  - pitää käydä ilmi KUKA pystyy tekemään MITÄ ja MIKSI
  - varsinainen teksti pyritään pitämään mahdollisimman lyhyenä
- Keterän projektin ensimmäinen vaihe on määrittelytyöpaja
  - tiimi = "kolmen voima" - asiakkaan edustajat, kehittäjät, testaajat
  - tiimi kirjoittaa tarpeet ja ominaisuudet käyttäjätarinoina
  - tiimi analysoi ja keskustelee niistä
  - tiimi korjaa vaillinaisuudet, tulkinnanvaraisuudet tai virheet

Hyväksymiskriteerit

- Hyväksymiskriteerit
  - kuvaavat kuinka käyttäjätarinassa kuvatun toiminnan toteutuminen voidaan arvioida valmiissa tuotteessa
  - MITÄ asioita tulee testata
- Kirjaa heti sprintin alussa
  - kun priorisoitu mitä käyttäjätarinoita sprintissä toteutetaan
  - asiakkaan, kehittäjän ja testaajan yhdessä laatima
- Asiakas kelpuuttaa, ”asiakas omistaa”
- Hyväksymiskriteerien suunnittelu etukäteen hyödyttää
  - tarjoaa kehittäjille / testaajille ominaisuudesta laajemman vision, ohjaa tuotantoa oikeaan suuntaan
  - selventää ja korostaa vaatimuksia
  - voi paljastaa puutteita toiminnallisessa määrittelyssä
  - varmistaa osaltaan, että toteutuskin tulee tehtyä oikein

Muut tiedot

- Käyttäjätarinoiden ja hyväksymiskriteerien lisäksi myös muu tieto merkityksellistä testaajalle
  - liittymät, joita voidaan käyttää ja joihin voidaan päästä järjestelmän testaamiseksi
  - onko nykyinen työkalutuki riittävää
  - onko testaajalla riittävät tiedot ja taidot suorittaa tarpeelliset testit
- Testaajat huomaavat usein sprintin aikana tarvitsevansa lisätietoa
  - heidän tulee työskennellä muiden ketterän tiimin jäsenten kanssa
  - tiedolla merkitystä, kun määritetään, voidaanko jotakin tiettyä tehtävää pitää valmiina
- Tiimi pitää tehtävää suoritettuna vasta kun hyväksymiskriteerit on täytetty
  - Definition of Done (DoD)

Esimerkki

- Käyttäjätarina
  - Kirjasto-ohjelman käyttäjänä haluan edistyneitä hakutoimintoja, jotta voin helposti ja nopeasti säätää hakuehtoja.
- Hyväksymiskriteeri, MITÄ
  - Voinko rajoittaa hakuja julkaisun tyypin mukaan?
  - Voinko rajoittaa hakuni julkaisun ajankohdan (väli) mukaan?
  - Voinko rajoittaa hakuni julkaisutiedon perusteella (julkaisun nimi, kirjailija, aihe, julkaisija, julkaisupaikka)?
  - Voinko rajoittaa hakuni tiettyyn katalogiin tai kokoelmaan?
  - Voinko rajoittaa hakuni julkaisun saatavuuden mukaan?

Hyväksymistesti

- Kuinka näytät toteen, että hyväksymiskriteerit täyttyvät
- Anna esimerkki käyttötilanteesta
  - kuvaa käyttäjätarinan erityisominaisuudet
- Esitä tavalla, jonka jokainen sidosryhmä pystyy ymmärtämään
  - kirjoita sovellusalueen ja asiakkaan kielellä
  - kuvaa tarvittavat esiehdot sekä syötteet ja niihin liittyvät tulokset
- Ensimmäiset testit ovat tyypillisesti positiivisia testejä ("happy path")
  - toimintosarja, joka suoritetaan, jos kaikki menee odotetusti
  - varmistetaan oikea toiminta ilman poikkeus- tai vikatilanteita
- Laadi lisäksi myös negatiivisia testejä ("failing path")
  - toimintosarja, jossa kokeillaan virhetilanteita
- Ja testejä, jotka kattaa ei-toiminnallisia attribuutteja
  - esim. suorituskyky, käytettävyys

Hyväksymistesti

- On vastaus kysymykseen MITEN testataan
  - ytimekäs ja yksiselitteinen
- Kuvaa miten testaat toiminnallisia kokonaisuuksia, ei aina tarkasti
  - kuvaus testin etenemisestä menee helposti turhan yksityiskohtaiseksi ja olisi ylipäätään vaikea laatia, jos käyttöliittymä ei ole vielä toteutettu
  - testin tulee pysyä samana, vaikka toteutus muuttuisi
- Pyri siihen, että kukin testitapaus mahdollisimman kattava
  - täytyy kattaa kaikki käyttäjätarinan ominaisuudet, mutta niiden ei pitäisi täydentää tarinaa
  - kahden eri esimerkin ei pitäisi kuvata samoja käyttäjätarinan ominaisuuksia
- Tsekkaa jäljitettävyys
  - jokaiselle vaatimukselle tulee löytyä riittävät testitapaukset

Esimerkki

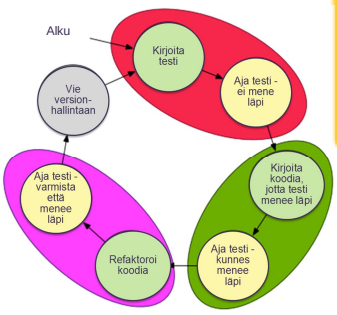
- Käyttäjätarina
  - Kirjaston asiakkaana haluan muokata itseäni koskevia tietoja.
- Hyväksymistestin askeleet, MITEN
  1. Kirjaudu järjestelmään
  2. Valitse sovelluksen alkusivulla ”Omat tiedot”
  3. Tarkasta, että näet tallennetut tiedot
  4. Tarkasta, että voit syöttää tietoa kaikkiin kenttiin
  5. Kokeile kaikissa kentissä sallituilla ja kielletyillä syötteillä
     - numerot vs. teksti, rajakohdat
  6. Talleta muutetut tiedot
  7. Poistu sivulta ja avaa sivu uudelleen
  8. Totea, että tiedot ovat päivittyneet
- Tällainenkin lisäaskel saattaa olla tässä tarpeen
  9. Tarkasta, että ”Tee lainaus”-sivu käyttää yllä muutettuja tietoja

JUnit

JUnit 5

- Open Source -yksikkötestausympäristö Javaa varten
- JUnit-ajuri sisäänrakennettuna IDE-ympäristöissä
- Ajettavissa myös IDE:n ulkopuolelta: ant, maven, gradle
- Vähintään Java 8, eli JDK8 tai uudempi
- JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage
  - JUnit Platform - ajoympäristö
  - JUnit Jupiter – JUnit 5 testien kirjastot
  - JUnit Vintage – vanhojen JUnit 4 testien kirjastot
- JUnit 5:n kirjastot, esim.
  - `import org.junit.jupiter.api.Test;`
  - `import org.junit.jupiter.api.BeforeAll`
  - `import static org.junit.jupiter.api.Assertions.assertEquals;`
- Vastaava työkalu myös muihin kieliin: CppUnit, pyUnit, PHPUnit, ...
- Linkkejä
  - https://junit.org/junit5/docs/snapshot/user-guide/
  - https://www.baeldung.com/junit-5
  - https://blog.codefx.org/libraries/junit-5-basics/

Testilähtöinen kehitys



1) Laadi testattavan luokan tynkä
2) Mieti testitapaus ja laadi testausmetodi
3) Toteuta testattava metodi
4) Aja kaikki testit
5) Korjaa virheet, paranna toteutusta

Toista 2) - 5) kunnes valmis ja testattu

- Tee sama kaikille metodeille / moduuleille
- Kun sovellus valmis, myös automatisoitu testipeti valmis
  - helppo regressiotestata

PUNAISTA - VIHREÄÄ - REFAKTOROI

1) Laadi testattavan metodin tynkä

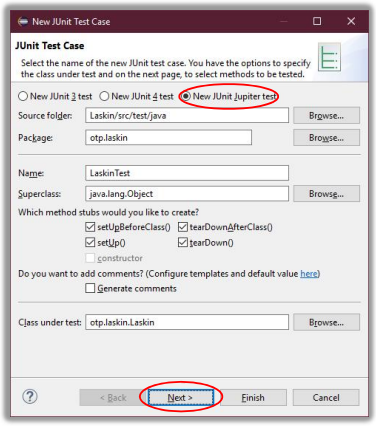
```
package laskin;
public class Laskin {
  private int tulos;

  public int annaTulos(){
     return tulos;
  }
  public void nollaa(){
     // Ei vielä toteutettu
  }
  public void lisää(int n){
     // Ei vielä toteutettu
  }
  public void vähennä(int n){
     // Ei vielä toteutettu
  }
  public void kerro(int n){
     // Ei vielä toteutettu
  }
  public void jaa(int n){
     // Ei vielä toteutettu
  }
}
```

Metodien käyttö suunniteltu (API)
- Metodien koodiosat tyhjiä
- Koodi kääntyy, mutta ei tee mitään

2) Mieti testitapaus ja laadi testi

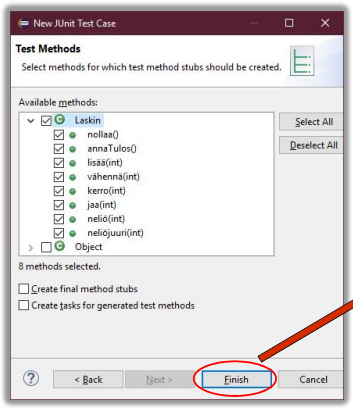
- Luo testattavalle luokalle pariksi erillinen testausluokka
  - varsinaisen sovelluksen koodin sekaan ei ympätä mitään ylimääräistä testaukseen liittyvää
- Kirjoita testausluokkaan metodit, joissa kutsut testattavia metodeja
- Tee kutakin testattavaa metodia kohden yksi tai useampi ajurimetodi
  - esim. yksi per testitapaus
  - ensin TDD:n perusperiaatteen mukaisesti ajo siten, että tulos on epäonnistuminen
- Testauksen tekevän metodin toimintaidea
  - setup tee testin vaatimat alustustoimet
  - act kutsu testattavaa metodia testitapauksen parametreilla
  - assert tarkista saatiinko odotettu lopputulos
- JUnit pitää kirjaa onnistumista / epäonnistumisista

Eclipse + JUnit

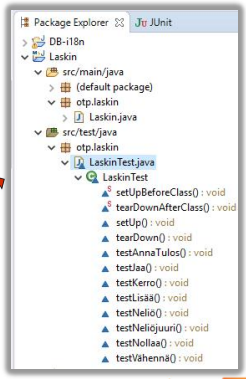


Kun testattavan luokan tynkä tehty, luo sitten testauksen tekevä luokka
- napsauta hiiren oikealla Project Explorerissa luokkaa, jolle haluat luoda testiluokan
- valitse New | JUnit Test Case

Eclipse voi luoda alustus- ja lopetusrutiineille valmiit tyngät (stubs)
- tarpeettomat voi toki jättää pois



Testattaville metodeille on jo tyngät, joten luo kullekin metodille myös testimetodin tynkä




```
package laskin;

import static org.junit.jupiter.api.Assertions
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.Before;
import org.junit.jupiter.api.BeforeClass;
import org.junit.jupiter.api.Test;

public class LaskinTest {

@BeforeAll
    public static void setUpBeforeClass() thro
    }

@AfterAll
    public static void tearDownAfterClass() th
    }

@BeforeEach
    public void setUp() throws Exception {
    }

@AfterEach
    public void tearDown() throws Exception {
    }
// jatkuu
```

```
// jatkoa
@Test
  void testNollaa() {
    fail("Not yet implemented");
  }
@Test
  void testLisää() {
    fail("Not yet implemented");
  }
@Test
  void testVähennä() {
    fail("Not yet implemented");
  }
@Test
  void testKerro() {
    fail("Not yet implemented");
  }
@Test
  void testJaa() {
    fail("Not yet implemented");
  }
@Test
  void testAnnaTulos() {
    fail("Not yet implemented");
  }
}
```

Alustusrutiinien tyngät
Tarpeettomat voi poistaa

Testausrutiinien tyngät

Maven Surefire

- Mavenissa testien ajamisesta huolehtii Surefire-plugin

```
<plugin> <!-- JUnit Jupiteria varten tarvitaan vähintään surefire 2.22.0 -->
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-plugin</artifactId>
   <version>3.0.0-M5</version>
</plugin>
...
<dependency> <!– Testaus JUnit Jupiterilla -->
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-engine</artifactId>
    <version>5.6.2</version>
    <scope>test</scope>
</dependency>                                    pom.xml
```

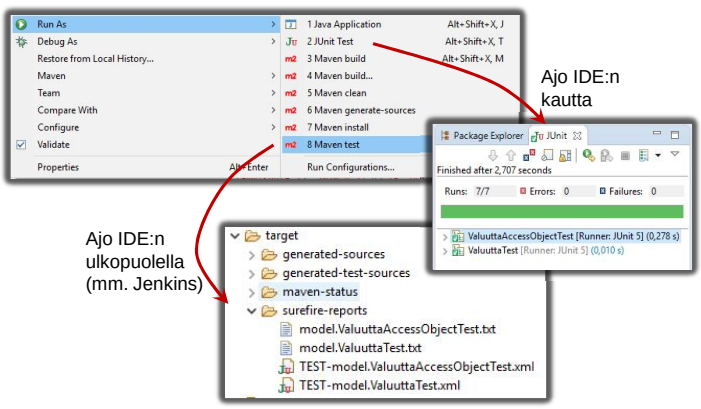
- Oletus: testiluokat hakemistossa src/test/java
- Käy oletuksena läpi alihakemistot ja tiedostot, joiden nimi on muotoa
  **/Test*.java tai **/*Test.java
  **/*Tests.java tai **/*TestCase.java
- Jos poikkeat oletuksista, konfiguroi pom.xml:ään

- @ParameterizedTest vaatii lisäksi riippuvuuden

```
<dependency> <!– Parametrisoitujen JUnit5-testien ajoa varten -->
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-params</artifactId>
   <version>5.8.2</version>
   <scope>test</scope>
</dependency
```

- Huomautus: @DisplayName ei toimi Surefirellä ajettaessa
  - Näyttää testimetodien nimet, valitse ne järkevästi

Testitulokset



Testien alustus

- Kun testattavana on useita tapauksia, testeille yhteiset alustus- ja jälkitoimet (eng. fixture) kannattaa yhdistää erillisiin metodeihin
- @BeforeEach-annotaatio
  - merkitsee metodin, joka suoritetaan ennen jokaista erillistä testiä
  - esim. testimuuttujien alustaminen, tiedoston avaaminen
- @AfterEach-annotaatio
  - merkitsee metodin, joka suoritetaan jokaisen erillisen testin jälkeen
  - esim. tiedoston sulkeminen, tms.
- @BeforeAll-annotaatio
  - merkitsee metodin, joka suoritetaan vain kerran ennen ensimmäistä testiä
  - esim. resurssien allokointia, tiedoston avaaminen, tms.
- @AfterAll-annotaatio
  - merkitsee metodin, joka suoritetaan vain kerran viimeisen testin jälkeen
  - esim. resurssien vapauttamista, tiedoston sulkeminen, tms.
- Koko lista löytyy API:sta
  - https://junit.org/junit5/docs/current/api/

fixture = vakiokalusto

Annotaatiota

- Ajurimetodien määreet
  - otsakkeen eteen annotaatio @Test tai @ParameterizedTest
  - ei palauta koskaan arvoa (void)
- JUnit suorittaa automaattisesti annotaatiolla merkityt metodit
  - ajurimetodien kutsuja ei siis tarvitse ohjelmoida itse
  - ei tarvitse kirjoittaa main()-metodia minnekään
  - suoritusjärjestys ei kiinteä, oleta siis satunnaisessa järjestyksessä
- Testin voi merkitä siten, että sitä ei suoriteta lainkaan
  - @Disabled("Syy miksei suoriteta")
  - JUnit muistuttaa tällaisistakin testeistä
- Testauksen tekevä luokka voi sisältää myös tavallisia metodeja
  - niiden edessä ei ole JUnitin annotaatiota
  - suoritetaan, kun koodissa eksplisiittinen kutsu

Runs: 5/5 (1 skipped)

Assert: Menikö testi läpi?

- Koodaa vertailuehto, jonka tulee olla totta metodikutsun jälkeen
  - vahvistetaan, että odotettu tulos ja saatu tulos ovat samoja
  - jos ei totta, testi ei mennyt läpi
- fail(String message)
  - ilmoittaa, että testi ei mennyt läpi
- assertTrue / assertFalse(boolean condition, String message)
  - tarkastaa onko parametrina annettu ehto true / false
- assertEquals(Object expected, Object actual) // kuormitettu
- assertEquals(Object expected, Object actual, String message)
  - tarkastaa onko parametrien expected ja actual arvot samoja
  - jos ei, raportoi myös käyttäjän antaman messagen
  - jos verrattavana double-arvot, lisää kolmanneksi parametriksi tarkkuus (delta)
- assertArrayEquals(Object[] expecteds, Object[] actuals)
- assertArrayEquals(Object[] expecteds, Object[] actuals, String msg)
  - taulukoiden kaikkien vastinkenttien oltava samat
- assertTimeout(Duration timeout, ...)
  - testin kestolle voi myös asettaa aikarajan

assert = vakuuttaa, väittää, vahvistaa

junit.org/junit5/docs/5.5.0/api/org/junit/jupiter/api/Assertions.html

- Myös poikkeus voi olla se odotettu tulos
- assertThrows(Class<T> expectedType, Executable executable)
- assertThrows(Class<T> expectedType, Executable executable, String msg)
  - kutsu palauttaa poikkeusolion, expectedType = odotetun poikkeuksen tyyppi

```
// Tässä testin odotettu tulos on, että nollallajako heittää metodin
   kutsujalle poikkeuksen
@Test
@DisplayName("Testaa nollallajako, pitää heittää poikkeus")
public void testJaaNollalla() {
  ArithmeticException poikkeus =
      assertThrows(ArithmeticException.class, () -> laskin.jaa(0),
                   "Piti heittää poikkeus");
  assertEquals("Nollalla ei voi jakaa", poikkeus.getMessage(),
               "Virheellinen teksti");
}
```

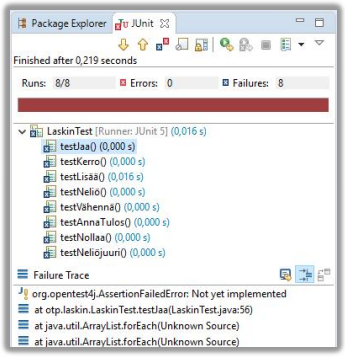
Failure Trace
org.opentest4j.AssertionFailedError: Piti heittää poikkeus ==> Expected java.lang.ArithmeticException to be thrown, but nothing was thrown.
at laskin.LaskinTest.testJaaNollalla(LaskinTest.java:57)

Failure Trace
org.opentest4j.AssertionFailedError: Virheellinen teksti ==> expected: <Nollalla ei voi jakaa> but was: <Jaettu on>
at laskin.LaskinTest.testJaaNollalla(LaskinTest.java:59)

- Kun edellinen testausta tekevä luokka suoritetaan komennolla

  `Run As | JUnit Test` (tai Alt+Shift+X, T)

  niin Eclipse antaa tällaisen yhteenvedon



Mikään testi ei mene läpi, koska on vain tyngät testattavista ja testien tekevistä metodeista

2) Laadi testi ensin

```
public class NelilaskinTest {

  private static Nelilaskin laskin;

@BeforeAll
  static void setUpBeforeClass() throws Exception {
    laskin = new Nelilaskin(); // Käytä kaikissa testeissä samaa
  }
@BeforeEach
  void setUp() {
    laskin.nollaa(); // Nollaa ennen testiä
  }
@Test
  void testNollaa() {
    System.out.println("nollaa");
    laskin.nollaa();
    assertEquals(0, laskin.annaTulos(), "Nollaus ei onnistunut");
  }
@Test
  void testLisää() {
    System.out.println("lisää");
    laskin.lisää(2);
    assertEquals(2, laskin.annaTulos(), "Yhteenlasku 0+2 ei onnistunut");
    laskin.lisää(2);
    assertEquals(4, laskin.annaTulos(), "Yhteenlasku 2+2 ei onnistunut");
}
```

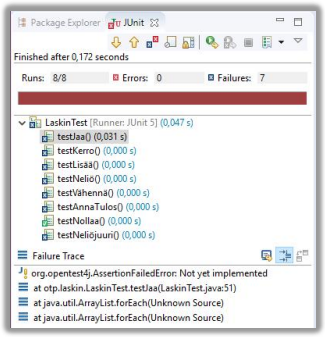
Testien yhteiset alustus- ja lopputoimet

setup – act - assert

Vertaa odotettua tulosta ja saatua tulosta (huomaa järjestys)

Aja testi

- Onko testi tehty oikein
  - ei saa mennä läpi, jos koodia ei ole vielä toteutettu



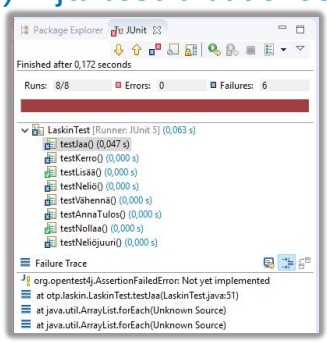
testNollaa()-metodissa oleva testi, meni nyt läpi
- sillä instanssimuuttujilla on aina oletuksena nolla-arvo

testLisää()-metodissa oleva testi epäonnistui
- sillä toteutus ei ole vielä kunnossa

3) Toteuta testattava piirre

```
package laskin;

public class Laskin {
  private int tulos;

  public void nollaa(){
     tulos = 0;
  }
  public void lisää(int n){
     tulos = tulos + n;
  }
  ...
}
```

4) Aja testit uudelleen



Molemmat testit menivät läpi

5) Korjaa virheet, paranna toteutusta

- Se ensimmäisenä mieleen tullut tapa ei kenties ollutkaan se paras
- Muista myös miettiä moduulien uudelleenkäyttöä

= Refaktoroi

- Miten refaktoroida, ks. esim. http://sourcemaking.com/refactoring
- Testeihin voi joutua toki palaamaan myöhemminkin
  - lisää uusia testejä
  - refaktoroi uudestaan vanhoja testejä

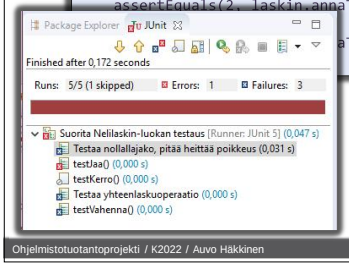
6) Toista vaiheita 2-5, kunnes moduuli valmis

TÄRKEÄÄ
- Muista tutkia testien tulokset huolella
- Varmista, että testaat oikeita asioita ja oikealla tavalla
- Testi voi mennä läpi, koska se on ohjelmoitu väärin
- Testi voi mennä läpi, koska on testattu väärää asiaa

@DisplayName

- JUnit tulostaa oletuksena suorittamiensa luokkien ja metodien nimet
- JUnit 5:ssä voi määritellä itse mitä niiden tilalle tulostuu

```
@DisplayName("Suorita Nelilaskin-luokan testaus")
public class NelilaskinTest
  @Test
  @DisplayName("Testaa yhteenlaskuoperaatio")
  void testLisää() {
    System.out.println("lisää");
    laskin.lisää(2);
    assertEquals(2, laskin.annaTulos(), "Yhteenlasku ei onnistunut");
```



Ei skulaa Mavenin surefirellä ajettaessa.

@TestMethodOrder, @Order

- Perusoletus, että testit suoritetaan "jossain" järjestyksessä
  - kukin testi lähtee ikään kuin tyhjästä pöydältä liikkeelle
- Testien suoritusjärjestyksen voi myös määrätä
  - Alphanumeerinen, numeroitu, random tai kustomoitu

```
@TestMethodOrder(MethodOrderer.Alphanumeric.class)
public class MethodAlphanumericTest {
    @Test
    void testZ() {
        assertEquals(2, 1 + 1);
    }
    @Test
    void testA() {
        assertEquals(2, 1 + 1);
    }
    @Test
    void testB() {
        assertEquals(2, 1 + 1);
    }
}
```

Järjestys: testA, testB, testZ

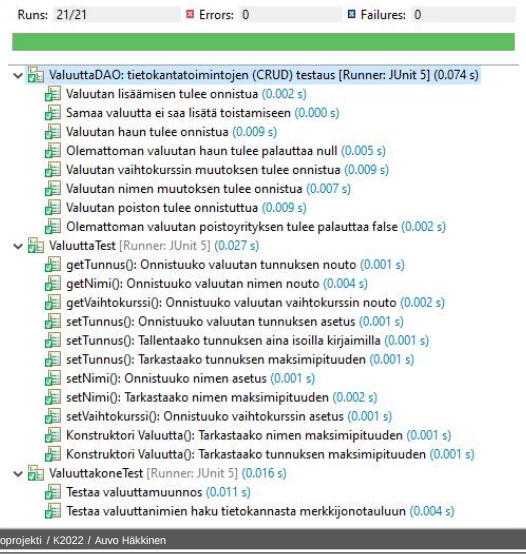
JUnit 5.4:stä alkaen

```
import org.junit.jupiter.api.MethodOrderer;
import org.junit.jupiter.api.TestMethodOrder;
import org.junit.jupiter.api.Order;

@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class MethodAlphanumericTest {
    @Test
    @Order(2)
    void testZ() {
        assertEquals(2, 1 + 1);
    }
    @Test
    @Order(1)
    void testA() {
        assertEquals(2, 1 + 1);
    }}
```

Järjestys: testA, testZ

@TestMethodOrder

- `@TestMethodOrder(MethodOrderer.Random.class)`
  - suorittaa testit satunnaisessa järjestyksessä s.e. järjestys on erilainen eri suorituskerroilla
  - jos halua, että seuraava koonti tuottaa täsmälleen saman järjestyksen kuin edellinen, voi random-generaattorille konfiguroida kiinteän siemenluvun
  - random-generaattorin siemenlukuna oletuksena aika
- `@TestMethodOrder(MyOwnOrderer.class)`
  - ohjelmoija voi kustomoida oman MethodOrderer-rajapinnan toteuttavan luokan, jossa voi itse järjestellä ajonaikaisesti testimetodit jollain kriteerillä, esim.
    - parametrien lukumäärän perusteella
    - metodin nimen perusteella
    - ...
- Kun testit on järjestetty loogisesti, niin on helpompi todeta onko oleelliset testattu tai puuttuuko joku testi



@RepeatedTest, @Timeout

- Sama testimetodi suoritetaan useita kertoja
  - Jos jotain valitaan randomilla, niin ...

```
@RepeatedTest(10)       // ei tarvita erikseen @Test
void testValinta() {
    valinta = pelaaja.valitseRandomilla();
    assertTrue(valinta == KIVI ||
               valinta == PAPERI ||
               valinta == SAKSET,
               "Virheellinen valinta");
}
```

- Riittääkö 10 kokeilua? 100? Testaako silti oikein?
  - no ei, aina voi tulla KIVI eikä nuo kaksi muuta koskaan
- Testin kestolle voi asettaa aikarajan, esim.
  - `@Timeout(value=2,unit=TimeUnit.SECONDS)`

@ParameterizedTest

- JUnit-ajuri kutsuu testausmetodia useita kertoja, eri parametreilla

```
@ParameterizedTest  // ei erikseen @Test
@ValueSource(strings = { "racecar", "radar", "Madam I am Adam" })
void palindromes(String candidate) {
  assertTrue(isPalindrome(candidate));
}
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void test_ValueSource(int i) {
    System.out.println(i);
}
@ParameterizedTest
@EnumSource(value = TimeUnit.class, names = { "DAYS", "HOURS" })
void testWithEnumSourceInclude(TimeUnit timeUnit) {
  assertTrue(EnumSet.of(TimeUnit.DAYS, TimeUnit.HOURS).contains(timeUnit));
}
@ParameterizedTest
@MethodSource("stringProvider")
void testWithSimpleMethodSource(String argument) {
  assertNotNull(argument);
}
static Stream<String> stringProvider() {
  return Stream.of("ananasakäämä", "simsalabim", "hokkuspokkus");
}
```



```
@ParameterizedTest
@CsvSource({ "Suomi, 1", "Viro, 2", "'La, paz', 3" })
void testWithCsvSource(String first, int second) {
    assertNotNull(first);
    assertNotEquals(0, second);
}

@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSource(String first, int second) {
    assertNotNull(first);
    assertNotEquals(0, second);
}
```

```
Valtio, viite
Suomi, 1
Viro, 2
"Ameriikan Yhdysvallat", 3
```

two-column.csv

- https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests
- https://www.journaldev.com/21639/junit-parameterized-tests

Asserteista

- Jos testi ei mene läpi, testimetodin suoritus päättyy heti

```
@DisplayName("only allows bridge crew to access the the phasers")
void canAccessPhasers() {
   assertTrue(testee.canAccessPhasers(picard),
            "Bridge crew should have access");
   assertFalse(testee.canAccessPhasers(barclay),
            "Crew should not have access");
   assertFalse(testee.canAccessPhasers(lwaxana),
            "Non crew should not have access");
}
```

Ei suoriteta, jos edeltävä ei mene läpi

- JUnit 5:ssä testit voi ryhmitellä kokonaisuuksiksi
  - kaikki erilliset testit suoritetaan huolimatta meneekö läpi vai ei

```
@DisplayName("only allows bridge crew to access the the phasers")
void canAccessPhasers() {
  assertAll(
    () -> assertTrue(testee.canAccessPhasers(picard),
            "Bridge crew should have access"),
    () -> assertFalse(testee.canAccessPhasers(barclay),
            "Crew should not have access"),
    () -> assertFalse(testee.canAccessPhasers(lwaxana),
            "Non crew should not have access")
  );
}
```

Nyt erottimena pilkku

Raportointi?

- Nimeä JUnit-testimetodit testin tarkoitusta kuvaavalla tunnuksella
  - luokkaa Laskin testaa luokka LaskinTest, luokkaa Maa MaaTest, jne.
- Nimeä metodit tekemistä / käyttäytymistä kuvaavalla verbillä
- Käytä @DisplayName –annotaatiota

```
@DisplayName("Saavutetaanko yläraja 100")
```

- Käytä assertXXX()-metodista muotoa, jossa mukana merkkijono, joka tulostuu, kun testi ei mene läpi

```
assertEquals(100, result, 0.0, "Ylärajaa ei saavutettu");
```

- Kommentoi testiluokan koodia riittävästi
  - mitä testaat milläkin tapauksella, varmista kattavuus
  - erikoisuudet
- Laita testiluokan alkuun yleiskommentti, josta käy ilmi testitapaukset
- Pidä testitapaukset jossain loogisessa järjestyksessä
  - helpompi huomata puuttuuko joku testitapaus

Muita huomioita

- Entä, jos metodi ei palauta arvoa (void), jota voisi tutkia assertissä?
- Jos metodi muuttaa jonkun muuttujan arvoa, tutki sitä
  - ns. sivuvaikutus
  - esim. lisäys kasvattaa listan kokoa, poisto pienentää -> tutki kokoa
  - esim. kun listaan viedään uusi alkio, sen haku pitäisi onnistua
- Laita metodit palauttamaan jotain merkityksellistä, vaikka kutsuja ei sitä käyttäisikään
  - tällöin voit tutkia testeissä assertilla
  - ei vaikuta kutsuihin, sillä Javassa kutsujan ei ole pakko ottaa paluuarvoa vastaan
- Tarvittaessa luokkaan voi tehdä testausta varten ekstrametodin esim. isValid(), jolla voi tutkia onko kaikki OK
  - tyyliin assertTrue(olio.isValid())

Muita huomioita

- Entä, jos olio käyttää toisen olion palvelua, eikä luokkaa ole vielä toteutettu?
- Priorisoi
  - toteuta ja testaa se toinen luokka ensin
- Käytä sijaista (test double, stub, fake)
  - on olevinaan alkuperäinen
  - tekee hommansa niin, ettei kutsuja huomaa eroa
  - palauttaa kutsujan kannalta oikeita arvoja
    - koodissa vain kovakoodattu return-lause
  - korvaa sitten aikanaan oikealla luokalla
- Käytä matkija-oliota (mock)
  - tuota oliota vastaava toiminnallisuus mock-kirjastolla testin aikana
  - saatavilla valmiita mock-kirjastoja, esim. Mockito, JMockit, EasyMock, jMock...

Kattavuus-analyysi

Kattavuusanalyysi

- Varmista, että testitapaukset suorittamalla ohjelmiston käskyt käyty riittävällä tarkkuudella läpi
- Kattavuusmittoja
  - Aliohjelmakattavuus (function coverage)
    - Onko jokaisessa aliohjelmassa (~metodi, funktio) käyty testien aikana?
  - Lausekattavuus (statement coverage)
    - Onko jokaisessa lauseessa käyty?
  - Päätöskattavuus (branch coverage)
    - Onko jokaisessa kontrollirakenteen haarassa käyty?
  - Ehtokattavuus (condition coverage)
    - Onko jokainen boolean lausekkeen yksittäinen ehto saanut sekä arvon true että false?
- Tavoite
  - Testeissä käyty vähintään kerran jokaisessa sovelluksen lauseessa

- Prosenttilukujen tulkinnassa kannattaa aina olla varovainen
  - suuri kattavuus ei välttämättä tarkoita hyvää testausta
- Kattavuusmitat eivät kerro mitään testauksen laadusta
- Kattavuusmittojen mukaan 100% testattu ei ole kattavasti testattu
  - kaikissa lausehaaroissa käyty, mutta...
  - mitat eivät huomioi puuttuvaa koodia
  - tekeekö olemassa oleva koodi juuri sen mitä asiakas halusi
- IDE-kehitysympäristöissä mukana myös kattavuustyökalut
  - NetBeans, Eclipse, IntelliJ, PhpStorm, ...
- Maven
  - tarvitsee jacoco-maven-plugin -lisäosan
  - tulokset: target/site -hakemistossa

JaCoCoverage analysis of project "Laskin" (powered by JaCoCo from EclEmma) > mockesimerkki — Source Files — Sessions

mockesimerkki

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed
Tuote		80%		n/a	1	3	1	6	1	3	0
Hinnoittelija		62%		n/a	2	5	2	5	2	5	0
TilaustenKäsittely		100%		100%	0	4	0	14	0	3	0
Tilaus		100%		n/a	0	3	0	6	0	3	0
Asiakas		100%		n/a	0	3	0	6	0	3	0
Total	6 of 110	95%	0 of 2	100%	3	18	3	37	3	17	0

```
1.  package mockesimerkki;
2.
3.  public class Hinnoittelija {
4.
5.      public float getAlennusP
6.          return 0.0f;
7.      }
8.
9.      public void aloita(){
10.
11.     }
12.
13.     public void lopeta(){
14.
15.     }
16.
17.     public void setAlennusPr
18.
19.     }
20. }
```

```
3.  public class TilaustenKäsittely {
4.
5.      private Hinnoittelija hinnoittelija;
6.
7.      public void setHinnoittelija(Hinnoittelija hinnoittelija) {
8.          this.hinnoittelija = hinnoittelija;
9.      }
10.
11.     public void käsittele(Tilaus tilaus) {
12.         Asiakas asiakas = tilaus.getAsiakas();
13.         Tuote tuote = tilaus.getTuote();
14.
15.         hinnoittelija.aloita();
16.         float prosentti = hinnoittelija.getAlennusProsentti(asiakas, tuote);
17.
18.         if(tuote.getHinta() >= 100){
19.             hinnoittelija.setAlennusProsentti(asiakas, prosentti + 5);
20.         }
21.
22.         prosentti = hinnoittelija.getAlennusProsentti(asiakas, tuote);
23.         float alennusHinta = tuote.getHinta() * (1 - (prosentti/100));
24.         asiakas.setSaldo(asiakas.getSaldo() - alennusHinta);
25.         hinnoittelija.lopeta();
```