Yksikkötestaus

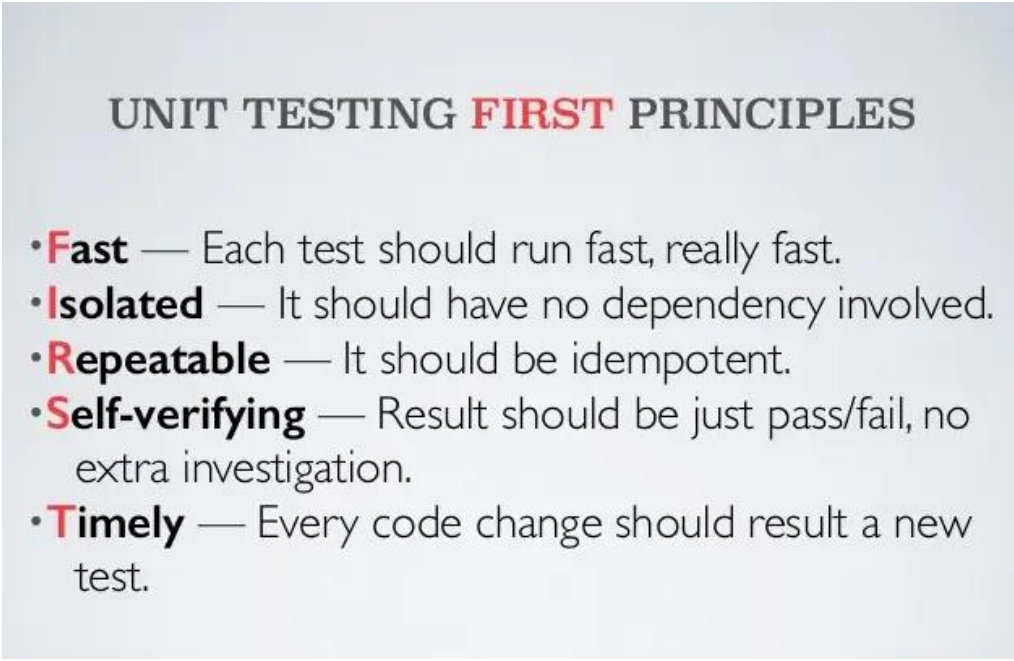
- Mikä, mitä, miten –

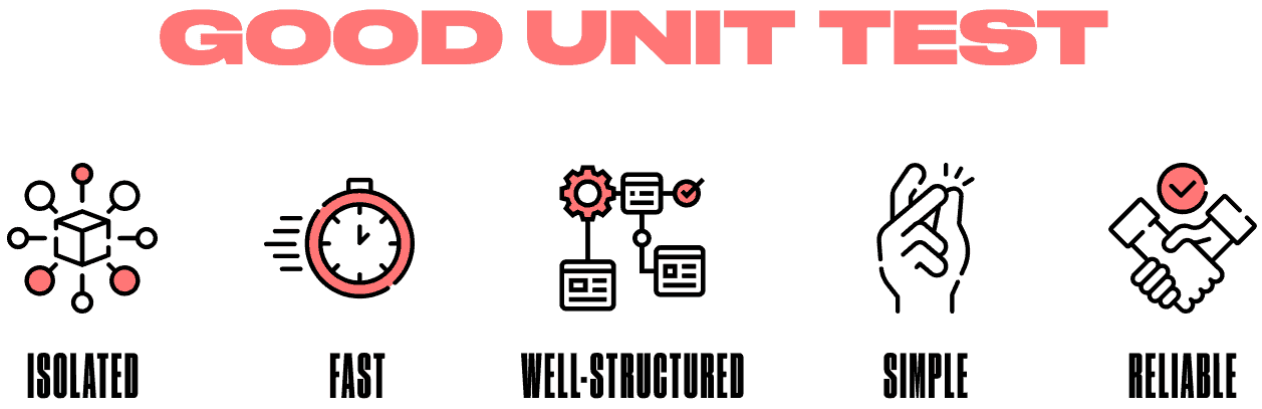
Aino Haikala 2023

Mikä on yksikkötesti?

Rakkaalla lapsella on monta määritelmää.

1. Yksikkötesti testaa yksikköä
2. Yksikkötesti on nopea ajaa
3. Yksikkötesti on eristetty





Kehittäjä 1 sanoo:

1. Yksikkötesti testaa yksikköä
   - Yksikkö on yksikkö koodissa
2. Yksikkötesti on nopea ajaa
3. Yksikkötesti on eristetty
   - Kaikki riippuvuudet korvataan mockeilla.
   - Jos yksikkönä on luokka ja luokastasi on riippuvuus toiseen luokkaan => mockkaat luokan ja laitat sen palauttamaan sopivat arvot.

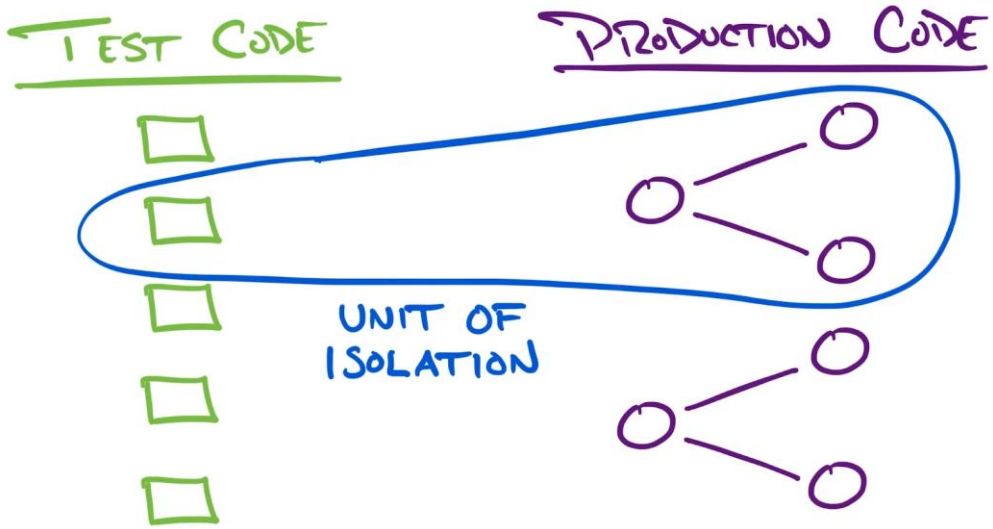
Kehittäjä 2 sanoo:

1. Yksikkötesti testaa yksikköä
   - Yksikkö on luonnollinen toiminnallinen yksikkö
2. Yksikkötesti on nopea ajaa
3. Yksikkötesti on eristetty
   - Jaetut riippuvuudet korvataan mockeilla.
     - Esim tietokannat, levyltä luvut
   - Riippuvuudet, jotka eivät tuota aina samaa tulosta korvataan mockeilla.
     - Esimerkkinä satunnaislukugeneraattori.
   - Jos testillä on riippuvuus, joka ei vaikuta muihin testeihin, se jätetään.

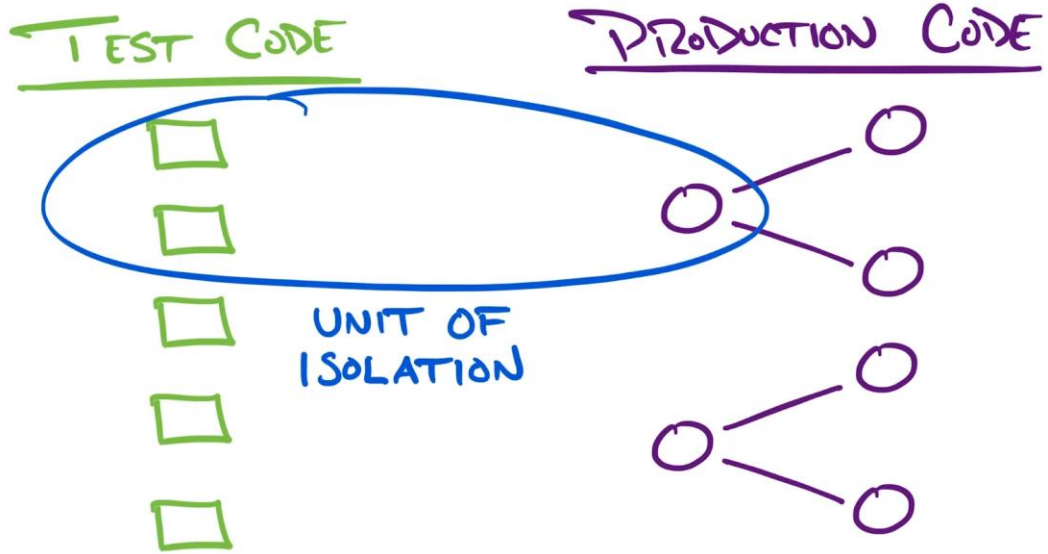
Kuka on oikeassa?

Kaksi isompaa koulukuntaa (classical ja mockist), jotka ovat eri mieltä siitä, mitä yksikkötestin eristyksellä tarkoitetaan.

"Unit tests are completely isolated from each other, creating their test fixtures from scratch each time." Beck



"[unit tests] exercise objects, or small clusters of objects, in isolation."



Kumpia minun pitäisi tehdä?

- Oma mielipiteeni molempia.
  - Käyttötarkoitus on erilainen, molemmissa on omat hyvät ja huonot puolensa.
- Mutta:
  - Ymmärtäkää taustat
  - Kannattaa tehdä yksikkötestejä pienille rajatuille kokonaisuuksille, joiden rajat olette määritelleet selkeästi. Se opettaa tekemään testattavaa koodia.
    - Pakottaa miettimään riippuvuussuhteita, rajapintoja…
    - Refaktoroin itse koodia yksikkötestien kirjoutuksen yhteydessä todella usein.

Hyvä yksikkötesti

- Suojaa regressioilta
- Ei hajoa refaktoroinneissa
  - Testaa toimintaa, älä toteutusta.
- Antaa nopeasti palautetta
  - Nopea ajaa
  - Selkeästi nimetty ja näkee helposti mitä testaa.
- Helppo ylläpitää
  - Selkeästi koodattu
  - Yksinkertainen framework
  - Jmockit -> mockito

Yksikkötestin anatomia

```
@Test
  public void doesNotTryToAddIfAllObjectsAreAlreadyAddedIntoDatabase() {                    <= Kuvaava nimi

      List<Long> deletedIds = List.of(123L, 456L, 890L); );                                  <= Arrange
      List<Long> handledIds = List.of(123L, 456L, 890L);
      String rollover = "aaab";

      Mock<WriteStatusQueries> mockedWrite = mock(WriteStatusQueries.class
      Mock<ReadStatusQueries> mockedRead = mock(ReadStatusQueries.class)

    mockedRead.when(() -> ReadStatusQueries.findExistingStatusObjects(jdbcTemplate, deletedIds)).thenReturn(handledIds);

    provider.addDeletedToDatabase(deletedIds, rollover, SanitizedObjectType.CAUTION_INFO_PERSON);     <= Act

    mockedRead.verify(() -> ReadStatusQueries.findExistingStatusObjects(jdbcTemplate, deletedIds),    <= Assert
                    times(1));

  }
```

Arrange-Act-Assert

- Arrange:
  - Luodaan pohja testille
  - Mockien alustukset, muuttujien luonnit
  - Yleensä isoin osa testistä.
- Act
  - Yleensä yksi toiminto.
  - Jos toimintoja kertyy useampi, se voi olla merkki, että rajapinnat eivät ole optimaalisia.
- Assert
  - Varmistetaan, että lopputulos haluttu.
  - Näitä saa olla useampi.

Arrange-Act-Assert-Act-Assert ja muut variaatiot

- Miksi koet tarvetta tehdä näin?
  - Pitäisikö testi jakaa useampaan osaan?
  - Voit luottaa siihen edelliseen testiin kyllä.

- ” The Single Responsibility Principle (SRP) is **the concept that any single object in object-oriented programing (OOP) should be made for one specific function**.” sopii tähänkin yhteyteen

Testin nimeäminen!!!!

- Usein aliarvostettu osa testiä.
- Testin nimen perusteella pitäisi pystyä päättelemään lokista, mikä on rikki.
- On olemassa erilaisia käytäntöjä, joilla testejä nimetään
  - Esim testattavan luokan tai metodin nimen lisääminen testiin.
- Pyrin itse suosimaan luettavaa nimeä, joka kertoo mistä on kyse
  - Given-when-then –mallia voi käyttää osviittana.
  - doesNotTryToAddIfAllObjectsAreAlreadyAddedIntoDatabase

Yksikkötestiluokan anatomia

```
public class FooDatabaseProviderTest {

   @BeforeAll                                                    //Setup
   public void setupAll() {}

   @BeforeEach
   public void setup() {}

   @Test
   public void doesNotTryToAddIfAllObjectsAreAlreadyAddedItemsIntoDatabase() {      // Tests
   }

   @AfterEach                                                                      //Teardown
   public void tearDown()  {}

   @After all
   public void tearDownAll()  {}
}
```

Testisetin anatomia

- Setup all => vain ne mitä voi käyttää kaikille testeille yhdessä.
- Setup each => yhteinen alustus yksittäisille testeille
- Testit => Eivät saa riippua toisistaan
  - Suosi factoryitä.
  - Testit ajettavissa missä vain järjestyksessä ja useita kertoja.
- Teardown => huolehdittava jälkitöistä.

Mitä mun pitäis testata?

- Lyhyt vastaus: Luultavasti enemmän kuin miltä devatessa tuntuisi.

- Meidän työ on tuottaa maksimaalinen hyöty asiakkaalle suhteessa resursseihin, joita meillä on.

- Mikä aikajänne? POC vs vuosikymmeniä käytössä oleva järjestelmä.
- Riski = Riskin todennäköisyys * riskin vaikutus toteutuessaan

Code coverage

- Managerit tykkää mitattavista arvoista:
  - Meillä on vaatimuksena x % code coverage.

- Miksi seniorikehittäjä irvistää tässä vaiheessa?
- Koodikattavuus on hyvä työkalu, mutta huono mittausväline, koska se antaa vain viitteitä asioiden toimivuudesta.

Line coverage vs branch coverage

- Line coverage = Lines of code excecuted / total number of lines

```
public Boolean isEven (int num) {
   return ( num % 2  == 0);
}
```

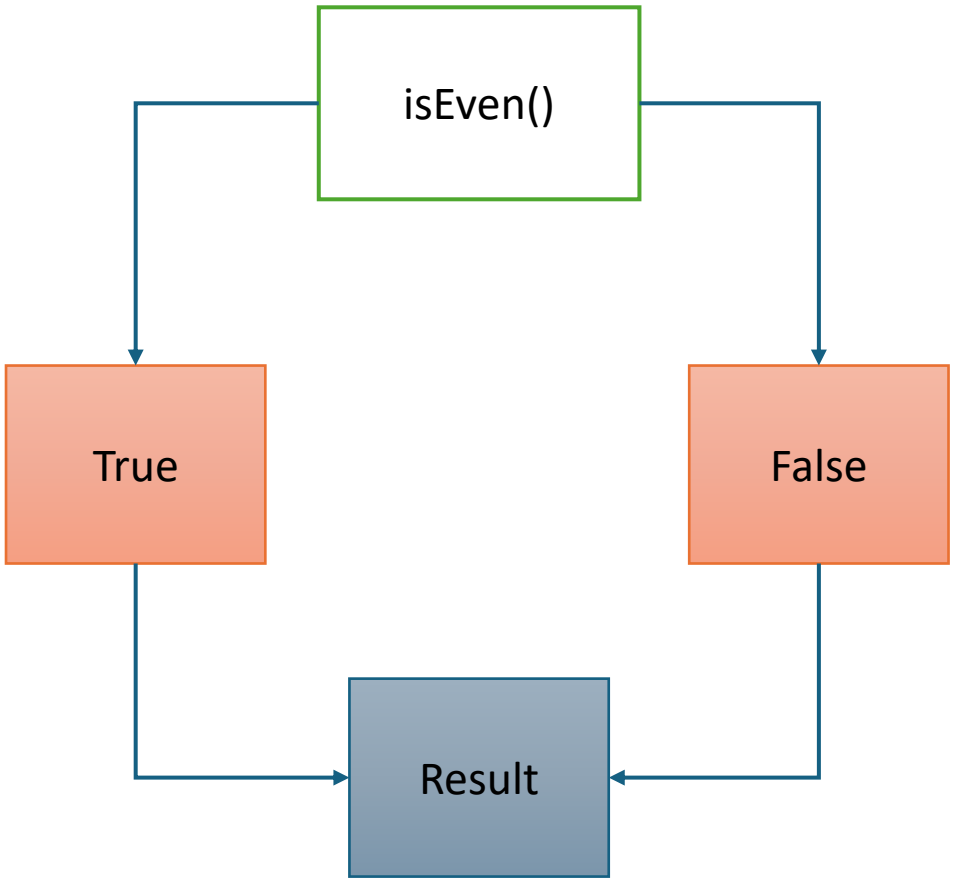
- Branch coverage = Code branches walked through / total number of branches

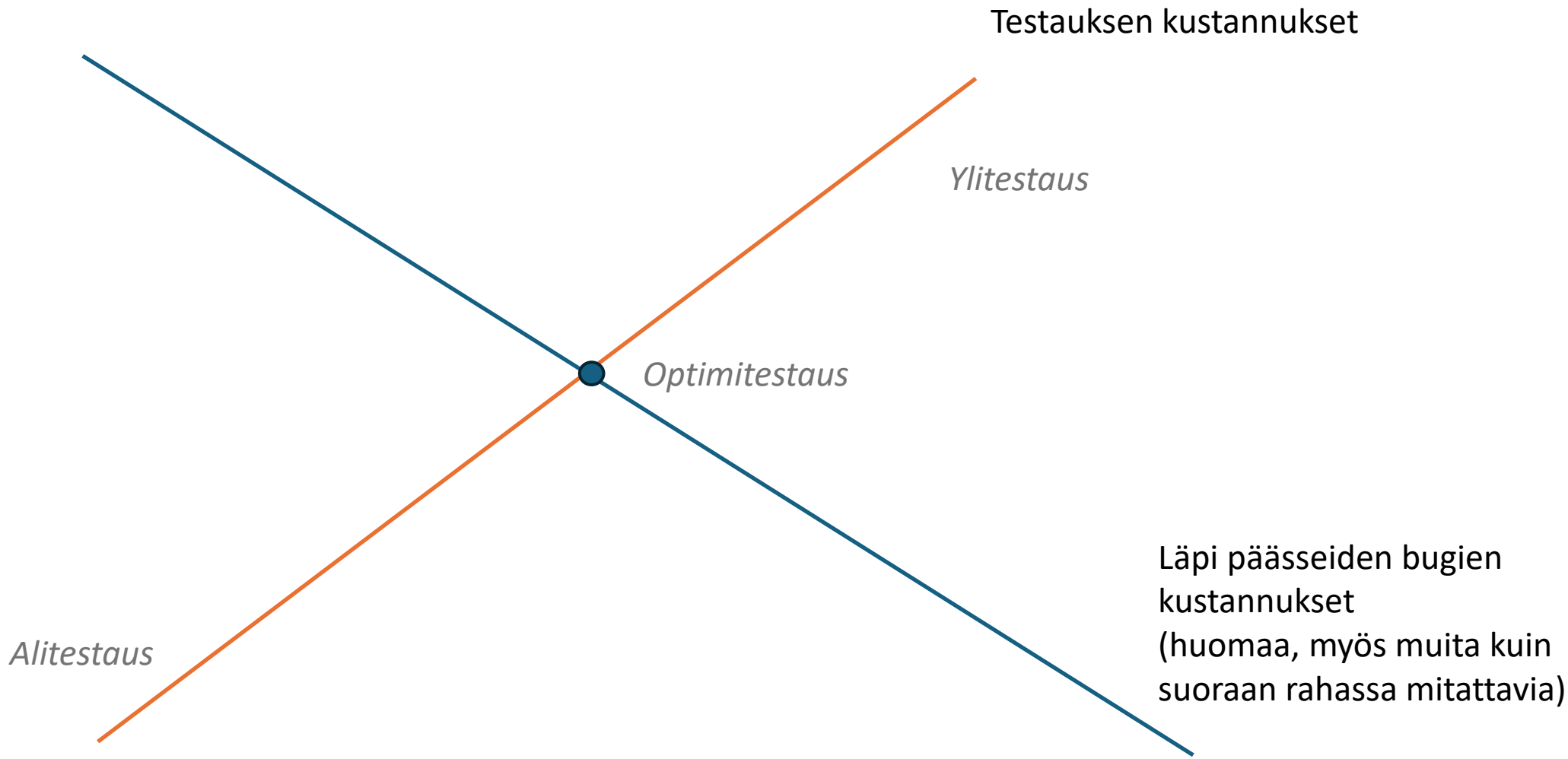
Coverage

```
public Boolean isEven (int num) {
   if ( num % 2  == 0) { return true; }
   return false;
}
```

```
// test
public twoIsEven () {
   Assert.equal(true, isEven(2))
}
```

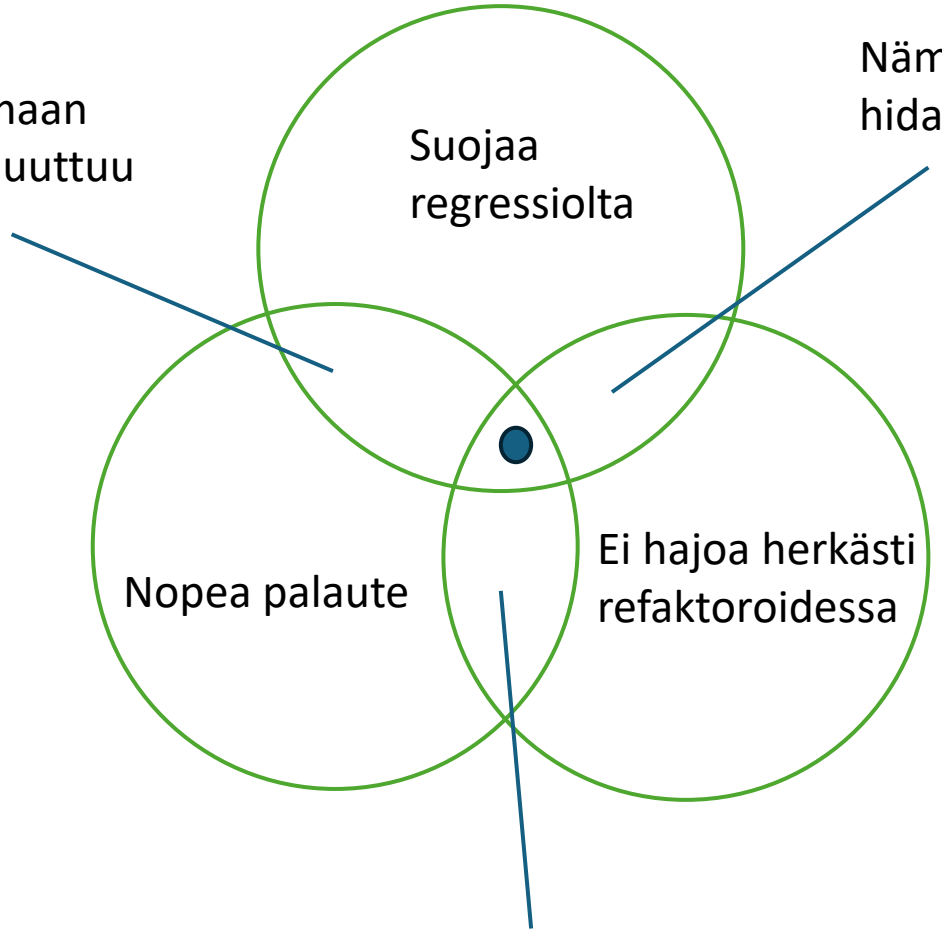
Code coverage?
Branch coverage?





Huomioi myös se, kasvaessaan projekti tulee hankalammaksi hallita (Monimutkaistuminen, software entropy).

Nämä testit joutuu kirjoittamaan uudestaan aina kun jotain muuttuu



Nämä testit ovat niin hitaita ajaa, että hidastavat koko ajoa.

Turhaa koodia. Mitä näillä tekee, jos ne ei auta?

Kysymyksiä?