

# Ohjelmistotekniikan menetelmät

Olli-Pekka Mehtonen, Matti Luukkainen

Avoin Yliopisto, Helsingin Yliopisto, TKTL

Kesä 2017

# Kurssi on johdantoa ohjelmistotuotantoon

## Wikipedian mukaan ohjelmistotuotanto

- ▶ on yhteisnimitys niille työnteon ja työnjohdon menetelmille, joita käytetään, kun tuotetaan tietokoneohjelmia sekä monista tietokoneohjelmista koostuvia tietokoneohjelmistoja.
- ▶ laajasti ymmärrettynä kattaa kaiken tietokoneohjelmistojen valmistukseen liittyvän prosessinhallinnan sekä kaikki erilaiset tietokoneohjelmien valmistamisen menetelmät.
- ▶ kattaa siis kaikki aktiviteetit, jotka tähtäävät tietokoneohjelmien tai -ohjelmistojen valmistukseen.

## Mallintaminen

- ▶ Ohjelmistotuotantoon liittyy **mallintaminen**, eli kyky tuottaa erilaisia kuvauksia, joita tarvitaan ohjelmiston kehittämisen yhteydessä
- ▶ Mallit toimivat kommunikoinnin välineinä
- ▶ *Mitä ollaan tekemässä, miten ollaan tekemässä, mitä tehtiin?*
- ▶ Kurssi oli aiemmin nimeltään *Ohjelmistojen mallintaminen*, painopiste on hieman muuttunut, mutta mallintaminen on edelleen vahvasti mukana

# Kurssista

- ▶ **Aikataulu ja kurssimateriaali:**

- ▶ <https://github.com/Ooppa/OTM-kesa17>

- ▶ **Laskarit:**

- ▶ Aloitetaan jo ensimmäisellä viikolla
  - ▶ Yhteensä 6kpl, 3h per tilaisuus

- ▶ **Arvostelu:**

- ▶ Kurssin kokonaispistemäärä on 36p
  - ▶ Kurssikoe 22p
  - ▶ Laskareista 14p, jotka koostuvat
    - ▶ Paikanpäällä tehtävistä 7p (1p per kerta)
    - ▶ Etukäteen tehtävistä 7p (90% → 7p)
  - ▶ Noin 32p → arvosana 5
  - ▶ Läpikäyty vaatii puolet kurssikokeen pisteistä ja puolet laskaripisteistä, eli ainakin 18p

# Laskarit

- ▶ **Etukäteen tehtävät:**

- ▶ viikoilla 2-7
- ▶ Keskimäärin 6 tehtävää viikossa
- ▶ Tehdään etukäteen niin, että vastauksia voidaan tarkastella ryhmissä, eli tulostettuna, läppärillä tai verkossa

- ▶ **Paikanpäällä tehtävät:**

- ▶ Ryhmätyöskentelyä (myös ensimmäisellä viikolla!)
- ▶ Tehdään niin paljon kuin ehtii, mutta työskennellään aktiivisesti
- ▶ Laskarit eivät ole paja, **paikalla on oltava alusta loppuun**

- ▶ **Laskariajat**

- ▶ **Ryhmä 1:** 1.8. - 17.8. Ti ja To klo 12-15 B221
- ▶ **Ryhmä 2:** 1.8. - 17.8 Ti ja To klo 15-18 B221

# Ohjelmistotuotantoprosessi

## Miksi prosessi kun voi vain tehdä?

- ▶ Pienissä itselle tehtävissä projekteissa voidaan ohjelmoida noudattamatta mitään systematiikkaa
  - ▶ Voidaan helposti häkätä kasaan sovellus, joka *"toimii"*
- ▶ Tämä menetelmä ei toimi isommille, monen hengen projekteissa asiakasta varten tuotetuille ohjelmille
  - ▶ Työn jakaminen tekijätiimiläisten kesken on hankalaa
  - ▶ Jää epäselväksi toimiiko sovellus niin kuin alunperin haluttiin?
  - ▶ Ohjelman rakenteesta tulee epämääräinen ja sen takia laajennettavuus ja ylläpidettävyyys on vaikeaa
- ▶ Ratkaisuksi on kehitelty lukuisia erilaisia menetelmiä ohjelmistotuotantoprosessin systematisoimiseksi <sup>1</sup>
- ▶ Mitä menetelmää tulisi käyttää? Hyvä kysymys!

---

<sup>1</sup>[https://en.wikipedia.org/wiki/List\\_of\\_software\\_development\\_philosophies](https://en.wikipedia.org/wiki/List_of_software_development_philosophies)

# Ohjelmistotuotantoprosessin vaiheet

Käytetystä menetelmästä riippumatta ohjelmistotuotantoprosessissa tapahtuu seuraavia aktiviteetteja

## 1. Vaatimusmäärittely

- ▶ Mitä halutaan?

## 2. Suunnittelu

- ▶ Miten tehdään?

## 3. Toteutus

- ▶ Ohjelmoidaan

## 4. Testaus

- ▶ Varmistetaan että toimii niin kuin halutaan

## 5. Ylläpito

- ▶ Korjataan bugeja ja laajennetaan ohjelmistoa

# Ohjelmistotuotantoprosessin vaiheet

## Vaatusmäärittely

Vaatusmäärittelyssä kartoitetaan ja dokumentoidaan **mitä asiakas haluaa**

- ▶ Selvitetään sovelluksen **toiminnalliset vaatimukset**
  - ▶ Mitä toimintoja ohjelmistolla tulisi olla?
- ▶ ja toimintaympäristön asettamat **rajoitteet**
  - ▶ Toteutusympäristö
  - ▶ Suorituskykyvaatimukset
  - ▶ Luotettavuusvaatimukset
  - ▶ Tietoturva
  - ▶ Käytettävyys
- ▶ Ei vielä puututa siihen miten järjestelmä tulisi toteuttaa
- ▶ Ei oteta kantaa ohjelman sisäisiin teknisiin ratkaisuihin, ainoastaan siihen miten toiminta näkyy käyttäjälle



# Ohjelmistotuotantoprosessin vaiheet

## Vaatusmäärittely

Esim: Yliopiston kurssinhallintajärjestelmä

- ▶ Toiminnallisia vaatimuksia:
  - ▶ Opetushallinto voi syöttää kurssin tiedot järjestelmään
  - ▶ Opiskelija voi ilmoittautua valitsemalleen kurssille
  - ▶ Opettaja voi syöttää opiskelijan suoritustiedot
  - ▶ Opettaja voi tulostaa kurssin tulokset
- ▶ Toimintaympäristön rajoitteita:
  - ▶ Kurssien tiedot talletetaan jo olemassa olevaan tietokantaan
  - ▶ Järjestelmää käytetään www-selaimella
  - ▶ Toteutus Javalla
  - ▶ Kyettävä käsittelemään vähintään 100 ilmoittautumista minuutissa

# Ohjelmistotuotantoprosessin vaiheet

## Vaatusmäärittely

### Toinen esimerkki: Reittiopas

- ▶ Toiminnallisia vaatimuksia:
  - ▶ Käyttäjä voi etsiä kahden osoitteen välisiä liikenneyhteyksiä
  - ▶ Käyttäjä voi etsiä nykyisen sijaintinsa ja jonkun osoitteen välisiä liikenneyhteyksiä
  - ▶ Käyttäjä voi valita minkä tyyppisiä kulkuneuvoja haluaa liikkueessaan käyttää
  - ▶ HSL:n virkailija voi ylläpitää kulkuneuvojen aikataulutietoja
- ▶ Toimintaympäristön rajoitteita:
  - ▶ Järjestelmää voi käyttää web-selaimella
  - ▶ Järjestelmää voi käyttää mobiililaitteelle asennetulla applikaatiolla
  - ▶ Mobiiliapplikaatiot on toteutettu natiivitekniikalla
  - ▶ Järjestelmän kyettävä käsittelemään vähintään 10000 kyselyä minuutissa

# Ohjelmistotuotantoprosessin vaiheet

## Vaatusmäärittely

- ▶ Jotta toteuttajat ymmärtäisivät mitä pitää tehdä, joudutaan ongelma-aluetta analysoimaan
  - ▶ Esimerkiksi jäsenetään ongelma-alueen käsitteistöä
  - ▶ Tehdään ongelma-alueesta *malli* eli yksinkertaistettu kuvaus
- ▶ Vaatusmäärittelyn päätteeksi yleensä tuotetaan **määrittelydokumentti**
  - ▶ Kirjaa sen mitä ohjelmalta halutaan
  - ▶ Käytetään ohjeena suunnitteluun ja toteutukseen
  - ▶ Testatessa varmistetaan, että järjestelmä toimii määrittelydokumentin kirjaamalla tavalla
- ▶ Määrittelydokumentin sijaan määrittely (tai ainakin sen osa) voidaan myös ilmaista ns. hyväksymiskriteereinä tai -testeinä.
  - ▶ Tällöin ohjelma toimii "määritelmänsä mukaisesti", jos se läpäisee kaikki määritellyt hyväksymiskriteerit

# Ohjelmistotuotantoprosessin vaiheet

## Ohjelmiston suunnittelu

Miten saadaan toteutettua määrittelydokumentissa vaaditulla tavalla toimiva ohjelma?

Suunnittelussa on useimmiten kaksi vaihetta

### 1. Arkkitehtuurisuunnittelu

- ▶ Määritellään ohjelman rakenne karkealla tasolla
- ▶ Mistä suuremmista rakennekomponenteista ohjelma koostuu?
- ▶ Miten komponentit yhdistetään, eli komponenttien väliset rajapinnat
- ▶ Mitä riippuvuuksia ohjelmalla on esim. tietokantoihin ja ulkoisiin rajapintoihin

### 2. Oliosuunnittelu

- ▶ yksittäisten komponenttien suunnittelu
- ▶ minkälaisista luokista komponentti koostuu
- ▶ miten luokat kutsuvat toistensa metodeja
- ▶ mitä apukirjastoja luokat käyttävät

Suunnittelun lopputuloksena on yleensä suunnitteludokumentti

# Ohjelmistotuotantoprosessin vaiheet

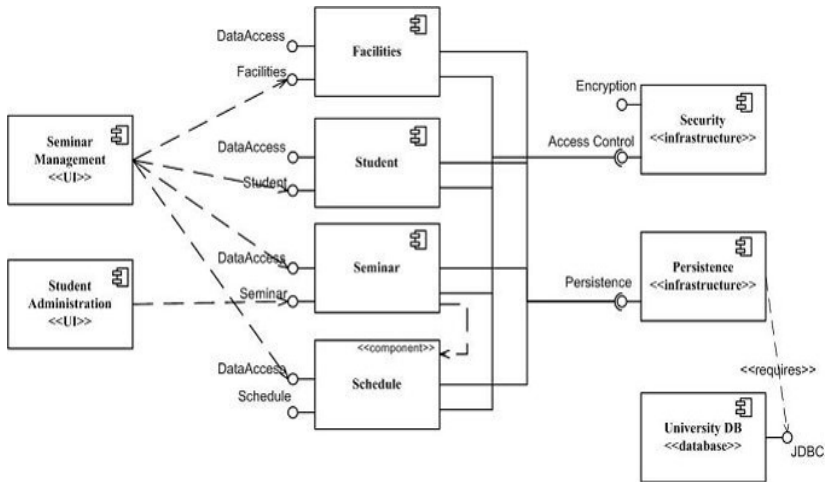
## Ohjelmiston suunnittelu

### ► Suunnitteludokumentti

- Ohje toteuttajille
- Joskus/usein suunnittelu- ja ohjelmointivaihe ovat niin kiinteästi sidottuna toisiinsa, että tarkkaa suunnitteludokumenttia ei tehdä
- Joskus koodi toimii dokumenttina

### ► Mallit liittyvät vahvasti suunnitteluun!

- Seuraavalla sivulla erään ohjelman *arkkitehtuurikuvaus*, joka visualisoi
  - järjestelmän alikomponentit ja
  - komponenttien väliset rajapinnat



# Ohjelmistotuotantoprosessin vaiheet

## Toteutus, testaus ja ylläpito

- ▶ Suunnittelun mukainen järjestelmä toteutetaan valittuja tekniikoita käyttäen
- ▶ Toteutuksen yhteydessä ja sen jälkeen testataan että järjestelmä on riittävän virheetön ja sisältää käyttäjän haluaman toiminnallisuuden
- ▶ Sen jälkeen kun ohjelma on saatu käyttöön, alkaa ylläpitovaihe
  - ▶ korjataan käytössä löydettyjä virheitä ja laajennetaan ohjelmaa uusilla toiminnallisuuksilla
  - ▶ on tyypillistä että ylläpitovaihe kestää vuosia, varsinkin nykyään, kun trendinä on saada ohjelmasta nopeasti käyttöön perustoiminnallisuus

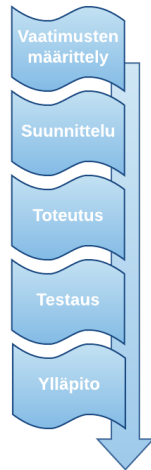
Tämän kurssin painopiste on vaatimusmäärittelyssä, suunnittelussa ja testaamisessa

# Ohjelmistotuotantoprosessi käytännössä

## Vesiputousmalli

Vesiputousmalli on vaiheittainen ohjelmistotuotantoprosessi, jossa suunnittelu- ja toteutusprosessi etenee vaihe vaiheelta alaspäin kuin vesiputouksessa.

- ▶ Tuotantoprosessin vaiheet etenevät peräkkäin
- ▶ Vaiheen valmistuttua siirrytään seuraavaan vaiheeseen
- ▶ Eri vaiheet ovat tyypillisesti eri henkilöiden tai tiimien suorittamia
- ▶ Jokaisen vaiheen lopputulos dokumentoidaan tyypillisesti erittäin tarkasti
- ▶ **Perinteinen tapa tehdä ohjelmistoja**





# Ohjelmistotuotantoprosessi käytännössä

## Vesiputousmalli

Vesiputousmallissa on kuitenkin ongelmia:

- ▶ Järjestelmää testataan kokonaisuudessaan vasta kun "kaikki" on valmiina
  - ▶ Suunnitteluvaiheen virheet saattavat paljastua vasta testauksessa
- ▶ Perustuu oletukselle, että käyttäjä pystyy määrittelemään ohjelmalta halutun toiminnallisuuden heti projektin alussa
  - ▶ Näin ei yleensä kuitenkaan tapahdu, asiakas osaa sanoa mitä todella tarvitsee vasta kun näkee lopputuotteen
  - ▶ ja koska projekti voi kestää pitkään, vaatimuksien voivat muuttua projektin aikana

Ohjelmistotuotannon yksi perustavanlaatuisimmista ongelmista on asiakkaan ja toteuttajien välinen kommunikointi!

# Ohjelmistotuotantoprosessi käytännössä

## Ketterä ohjelmistokehitys (Agile software development)

- ▶ Lähdetään olettamuksesta, että asiakkaan vaatimukset muuttuvat ja tarkentuvat projektin kuluessa
  - ▶ Ei siis yritetäkään kirjoittaa alussa määrittelydokumenttia, jossa on kirjattuna tyhjentävästi kaikki järjestelmältä haluttu toiminnallisuus
- ▶ Tuotetaan järjestelmä **iteratiivisesti**, eli pienissä paloissa
  - ▶ Ensimmäisen iteraation aikana tuotetaan pieni osa järjestelmän toiminnallisuutta
    - ▶ määritellään vähän, suunnitellaan vähän, toteutetaan ja testataan
  - ▶ Lyhyitä iteraatioita - tyypillisesti muutaman viikon
    - ▶ iteraatioita kutsutaan usein *sprinteiksi*
  - ▶ Asiakas antaa palautetta jokaisen iteraation päätteeksi → suuntaa on mahdollista korjata
  - ▶ Seuraavassa iteraatiossa toteutetaan taas hiukan uutta toiminnallisuutta asiakkaan toiveiden mukaan

Jokaisen iteraation päätteeksi lopputuloksena ohjelmisto, jossa on mukana toiminnallisuutta.

# Ohjelmistotuotantoprosessi käytännössä

## Ketterä ohjelmistokehitys (Agile software development)

Asiakkaan kanssa jatkuva kommunikaatio on olennainen osa ketteryyttä

- ▶ Asiakkaan palaute on välitön
  - ▶ Vaatimuksia voidaan tarkentaa ja muuttaa
- ▶ Asiakas valitsee jokaisen iteraation aikana toteutettavat lisäominaisuudet
  - Todennäköisempää että aikaansaannos on asiakkaan toiveiden mukainen
- ▶ Iteraation sisällä määrittely, suunnittelu, toteutus ja testaus eivät välttämättä etene peräkkäin (usein jatkuvaa!)
  - ▶ Ketterissä menetelmissä dokumentoinnin rooli on yleensä kevyempi kun vesiputousmallissa
- ▶ Ketterässä ohjelmistotuotannossa periaattena on se, että sama tiimi vastaa määrittelystä, suunnittelusta, toteutuksesta ja testaamisesta
  - ▶ Tämä vähentää dokumentaation tarvetta

# Ohjelmistotuotantoprosessi käytännössä

## Ketterä ohjelmistokehitys (Agile software development)

Virheellinen johtopäätös on ajatella, että kaikki ei-perinteinen tapa tuottaa ohjelmistoja on ketterien menetelmien mukainen

- ▶ Häkkerointi siis ei ole ketterä menetelmä
- ▶ Ketteryys ei tarkoita ettei ole sääntöjä
- ▶ Monissa ketterissä menetelmissä (kuten eXtreme Programming) on päinvastoin erittäin tarkasti määritelty miten ohjelmien laatua hallitaan
  - ▶ Pariohjelmointi, jatkuva integraatio, automatisoitu testaus, Testaus ensin -lähestymistapa (TDD, Test Driven Development), ...
- ▶ Eli myös ketteryys vaatii kurinalaisuutta, joskus jopa enemmän kuin perinteinen vesiputousmalli

# Testaus

- ▶ Tarkastellaan seuraavaksi tarkemmin ohjelmien testaamista
- ▶ Testausvaiheessa siis tarkoitus varmistaa, että järjestelmä on riittävän laadukas käytettäväksi
- ▶ Testauksella on kaksi hieman erilaista tavoitetta
  - ▶ osoittaa, että järjestelmällä on käyttäjän haluamat vaatimukset
  - ▶ käytännössä tämä tarkoittaa vaatimusdokumenttiin/vaatimuksiin kirjattujen asioiden toteutumisen demonstroimista
  - ▶ osoittaa, että järjestelmä on riittävän virheetön

# Testaustasot

Testaus jakautuu useisiin erilaisiin aktiviteetteihin tai *testaustasoihin* sen mukaan mihin testaus kohdistuu

- ▶ **Yksikkötestaus**

- ▶ Toimivatko yksittäiset metodit ja luokat kuten halutaan?

- ▶ **Integraatiotestaus**

- ▶ Varmistetaan komponenttien eli yksittäisten luokkien ja niiden muodostavien kokonaisuuksien yhteentoimivuus
- ▶ Jos sovellus käyttää tietokantaa tai verkossa olevia rajapintoja, voi niiden käytön oikeellisuuden varmistaminen kuulua integraatiotestauksen piiriin

- ▶ **Järjestelmä/hyväksymistestaus**

- ▶ Toimiiko kokonaisuus niin kuin vaatimusdokumentissa sanotaan?
- ▶ Testataan sovellusta sen käyttöliittymän kautta
- ▶ Kutsutaan hyväksymätestaukseksi jos loppukäyttäjän suorittama

# Testaustasot

## ► Regressiotestaus

- *regressio*  $\approx$  palautuminen, taantuminen, takautuminen
  - järjestelmälle muutosten ja bugikorjausten jälkeen ajettavia testejä, jotka varmistavat että muutokset eivät riko mitään
  - regressiotestit koostuvat yleensä yksikkö-, integraatio- ja järjestelmätesteistä
  - tärkeässä roolissa ketterissä menetelmissä, missä ohjelmistoa rakennetaan valmiiksi pala palalta
- 
- Manuaalinen testaus on aikaavievää, nykyinen trendi onkin tehdä testeistä automaattisesti suoritettavia
  - Yksikkö- ja integraatiotestit ja jossain määrin myös järjestelmätestit ovat yleensä sovelluskehittäjien tekemiä
    - Testaamisesta hyötyvät loppukäyttäjän lisäksi myös sovelluskehittäjät, ilman kattavia, automatisoituja testejä, ohjelman jatkokehittäminen muuttuu virhealttiiksi, hitaaksi ja stressaavaksi

# Yksikkötestaus - JUnit

- ▶ Yksikkötesteillä siis testataan toimivatko ohjelman yksittäiset luokat ja niiden metodit oikein
- ▶ Yksikkötestit ovat nykyään aina automatisoituja, sovelluskehittäjän kirjoittamia testejä
- ▶ Javalla ohjelmoitaessa testit tehdään useimmiten *JUnit*-testauskirjaston avulla
- ▶ Yhtä loogista kokonaisuutta (esim. luokkaa) testaavat testitapaukset sijoitetaan yhden testiluokan sisälle
- ▶ Yksittäiset testit eli testitapaukset toteutetaan testiluokan sisälle metodeina, jotka on annotoitu eli merkitty avainsanalla `@Test` avainasanalla
- ▶ Yksittäisessä testitapauksessa kannattaa yleensä testata ainoastaan yhtä "asiaa", esim:
  - ▶ saako maksukortti alussa oikean saldon
  - ▶ väheneekö maksukortin saldo oikein ostettaessa edullinen lounas
  - ▶ maksukortin saldo ei voi mennä negatiiviseksi



# JUnit-esimerkki

```
public class MaksukorttiTest {  
  
    @Test  
    public void konstruktoriAsettaaSaldonOikein() {  
        Maksukortti kortti = new Maksukortti(10);  
  
        String vastaus = kortti.toString();  
  
        assertEquals("Kortilla on rahaa 10.0 euroa", vastaus);  
    }  
  
    @Test  
    public void syoEdullisestiVahentaaSaldoaOikein() {  
        Maksukortti kortti = new Maksukortti(10);  
  
        kortti.syoEdullisesti();  
  
        assertEquals("Kortilla on rahaa 7.5 euroa", kortti.toString());  
    }  
}
```

# JUnit

- ▶ Yksittäinen testitapaus luo ensin testattavan olion, kutsuu oliolle testattavaa metodia ja lopulta varmistaa että metodi toimii halutulla tavalla
- ▶ Jokainen yksittäinen testimetodi suoritetaan toisistaan täysin riippumattomana. Testien kirjoitusjärjestyksellä ei siis ole mitään merkitystä. Jokainen testi on kuin pieni oma pääohjelma, joka testaa luokan toimintaa yksittäisessä tilanteessa
- ▶ Yksittäiset testimetodit sisältävät usein toistoa, esim. testattavan olion luominen voi tapahtua monessa testissä samalla tavalla
- ▶ Useiden testien yhteiset osat voidaan eristää, siirtämällä yhteiset osat avainsanalla *@Before* annotoituun metodiin
  - ▶ @Before-metodi suoritetaan ennen jokaisen metodin suoritusta
  - ▶ testattavat oliot tulee sijoittaa testausluokan oliometodeihin, jotta ne näkyvät @Before:n ja testimetodien sisällä

# JUnit-esimerkki

```
public class MaksukorttiTest {  
  
    Maksukortti kortti;  
  
    @Before  
    public void setUp() {  
        kortti = new Maksukortti(10);  
    }  
  
    @Test  
    public void konstruktoriAsettaaSaldonOikein() {  
        assertEquals("Kortilla on rahaa 10.0 euroa", kortti.toString());  
    }  
  
    @Test  
    public void syoEdullisestiVahentaaSaldoaOikein() {  
        kortti.syoEdullisesti();  
        assertEquals("Kortilla on rahaa 7.5 euroa", kortti.toString());  
    }  
  
    @Test  
    public void syoMaukkaastiVahentaaSaldoaOikein() {  
        kortti.syoMaukkaasti();  
        assertEquals("Kortilla on rahaa 6.0 euroa", kortti.toString());  
    }  
}
```

# Hyvät testit

- ▶ Hyvät testit ovat kattavat eli testaavat kaiken koodin monipuolisesti
- ▶ Normaalien syötteiden lisäksi on testeissä erityisen tärkeää tutkia testattavien metodien toimintaa virheellisillä syötteillä
  - ▶ mitä tapahtuu jos kortin saldo ei riitä ostokseen
  - ▶ mitä tapahtuu jos kortille yrittää ladata negatiivisen määrän rahaa
- ▶ Erityisen tärkeää on testata koodin toimintaa *raja-arvoilla*
  - ▶ voiko maksukortilla ostaa lounaan jos kortilla on rahaa täsmälleen lounaan verran
- ▶ Testien kattavuutta voidaan myös mitata erilaisten työkalujen avulla
  - ▶ *rivikattavuus* (line coverage) kuvaa sitä, kuinka monta prosenttia luokan koodista testit suorittavat

Palaamme yksikkötesteihin laskareissa ja muihin testauksen tasoihin (integraatiotestaus ja järjestelmätestaus) myöhemmin kurssilla

# Hyvät testit

Ohjelmoija kävelee baariin ja...

- ▶ tilaa oluen
- ▶ tilaa 0 olutta
- ▶ tilaa -1 olutta
- ▶ tilaa 2147483648 olutta
- ▶ tilaa *asdasdasdas*